Thesis for the degree of Master of Science

Constraints for Type Class Extensions

Gerrit van den Geest

Supervisor: prof. dr. S. D. Swierstra
Daily supervisors: dr. B. J. Heeren and dr. A. Dijkstra
Utrecht, February 2007
INF/SCR-06-36

Department of Information and Computing Sciences
Universiteit Utrecht
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Universiteit Utrecht

Abstract

Type classes are perhaps the most exciting feature of Haskell's type system. Although type classes were only proposed as a solution for overloading identifiers, they have become an active research topic where experimental type class extensions are still being proposed. On top of that, the underlying principles are also used to encode various other extensions, such as extensible records, implicit parameters, and subtyping. However, implementing type classes and type class extensions is not a trivial task. It is not only a matter of type checking to resolve overloading, but also evidence for overloaded identifiers has to be inserted. A uniform approach to easily formulate type class extensions side by side is missing. In addition, error messages concerning type classes are difficult to understand or sometimes not present at all.

We propose a constraint-based framework for the resolution of overloading. Assumptions and proof obligations are explicitly encoded into the constraint language of this framework. Furthermore, type class extensions can be easily formulated using Constraint Handling Rules (CHRs). The confluence requirement is circumvented by using only propagation CHRs, and by specifying design decisions in the form of heuristics. Using the resulting framework, we show how various context reduction strategies can be specified side by side. Furthermore, we explain how scoped instances and overlapping instances are naturally supported. We also show how functional dependencies can be supported using the existing translation into CHRs.

Contents

Chapter 1

Introduction

Take a look at the following sentence: "Shall we take a break?". At first sight this is a very simple sentence, however 'break' already has at least 28 different meanings. Luckily, we are able infer the meaning of break from the context. However, this is not always the case. Consider the following example attributed to the linguist Noam Chomsky: "Flying aircraft may be hazardous." Does this sentence mean that it is hazardous to be a pilot, or that airplanes themselves may be hazardous when flying? We cannot infer the meaning of this sentence from the context and thus we need some additional context information. Inferring the meaning of a word from the context is reminiscent of overloading in programming languages.

In a programming language we typically want to use the same function identifier for different operations; we call such a function *an overloaded function*. An example of an overloaded function is the operator $(\equiv)$: we want to use this operator to compare two integers, but also to compare two floating point values, and perhaps even to compare two strings for equality. Because the implementation of comparing two integers is different from that of comparing two strings, the compiler must infer the meaning of the operator $(\equiv)$ from the context.

There is an important difference between overloading and polymorphism. Overloaded functions execute different code for different types of argument. Comparing two integers requires other code to be executed than when comparing two strings. On the other hand, *polymorphic* functions execute the same code for different types of arguments. For example, the function $length :: \forall\ a \Rightarrow [a] \rightarrow Int$ can be executed for all types of lists. Defining equality as a polymorphic function is even impossible, because this would imply we would be able to define equality between functions. Alternatively, an overloaded function can be implemented by testing the type of the operands and then execute different code for integers and strings. This is not a very elegant and extensible solution for overloading.

Type classes were introduced by Kaes [Kaes, 1988] and Wadler and Blot [Wadler and Blott, 1989] as a solution for overloading identifiers. Their type class design was adopted by Haskell [Peyton Jones, 2003]. The original design of type classes was conservative and for that reason a large number of extensions are proposed [Peyton Jones et al., 1997]. Implementing those extensions in a compiler and experimenting with those extensions is not very easy. In this thesis we propose a framework for resolving overloading in which various type class extensions can be formulated. A requirement for this framework is that it must be relatively easy to experiment with type class extensions and various design decisions.

1.1 Research problem

Much effort had gone into specifications [Hall et al., 1996, Faxén, 2002] and implementations [Augustsson, 1993, Jones, 1999] of Haskell type classes. However, no clear specifications and implementations of the various extensions exist, let alone a framework in which we can easily formulate type classes and extensions to type classes. We list some problems with type classes:

- There is no general framework which can be used by compilers to resolve overloading in Haskell. Instead each compiler uses its own solution. This makes it difficult to experiment with type classes.
- Type classes, like many other language features, come with a price tag in terms of runtime efficiency. In general, we want to optimize away such a cost when the corresponding feature is used.
- Error messages concerning type classes are often difficult to understand [Heeren and Hage, 2005].

1.2 Objectives

The proposed research is to create a constraint-based framework in which we can formulate Haskell type classes and its various extensions. As a proof of concept we use this framework in the compilers EHC [Dijkstra, 2005] and Helium [Heeren et al., 2003b]. We first list the requirements of this framework:

- The framework must support Haskell 98 type classes.
- The framework must support multi-parameter type classes.
- The framework must support local instances and explicit passing of type class dictionaries, such as formulated by Dijkstra and Swierstra [Dijkstra and Swierstra, 2005].
- The framework must be general enough for use in the compilers EHC and Helium to resolve overloading and generate code for type classes.
- It must be easy to formulate type class extensions in the framework.
- The framework must be well documented and structured, so that people can easily use, adapt, and experiment with the framework.

Second, we list additional requirements which are nice to implement, but the absence of which will not cause this project to fail.

- It should be easy to incorporate optimizations [Augustsson, 1993] applied to the generated code into the framework.
- Also the framework should be designed so that existing and future work on type class directives [Heeren and Hage, 2005] can be integrated.
- It would be very nice to implement functional dependencies [Jones, 2000] in the framework or the alternative solution in terms of associated types [Chakravarty et al., 2005b,a].

1.3 Contribution

Are we the first developing such a framework? Yes and no, for instance, Chameleon [Sulzmann and Wazny, 2001] is a Haskell-style language, which implements the ideas described by Stuckey and Sulzmann [Stuckey and Sulzmann, 2002]. In Chameleon it is possible to define overloaded functions. However, Chameleon does not support Haskell 98 type classes nor any type class extensions. Very interesting is the formulation of overloading in rules and the way users can supply rules to experiment with the type system. In Chapter 3 we review the work of Sulzmann et all. The second framework is Top [Heeren, 2005]. This is a constraint-based type inferencer for Haskell designed to produce high quality type error messages. Top supports Haskell 98 type classes, but no type class extensions. The interesting part of Top is the way users can script the type inference process [Heeren et al., 2003a] to give better type error messages and the formulation of overloading resolution using constraints.

1.4 Structure of this thesis

In Chapter 2 we introduce type classes and qualifier entailment. We review in Chapter 3 a selection of the literature concerning type classes. The following two chapters describe the current situation. This framework must become part of the constraint-based type inferencer Top. Currently, Top is used by the Helium compiler. In Chapter 4 we explain how overloading is resolved in Top. The framework we describe in this thesis is also going to be used in EHC. Therefore, we explain in Chapter 5 how overloading is implemented in EHC. In Chapter 6 we describe the first version of the framework. We explain how overloading resolution can be formulated with constraints and how those constraints can be solved with Constraint Handling Rules [Frühwirth, 1998]. In Chapter 7 we extend the framework to translate a language with overloading into a language without. Furthermore, we show how design decisions can be specified in the form of heuristics. We explain in Chapter 8 how local instances can be formulated in our framework. In Chapter 9 we extend the framework with improving substitution and show how the existing translation from functional dependencies into CHRs can be used. Chapter 10 concludes.

Chapter 2

Preliminaries

In this chapter we give some preliminary information concerning type classes and constraints. We first introduce various aspects of type classes: class and instance declarations, context reduction, dictionary translation, and ambiguities. In the last part of this chapter we introduce the entailment relation and qualified types.

2.1 Type classes

Type classes are proposed as a solution for overloading identifiers in Haskell. In this subsection we introduce Haskell 98 type classes and describe how a language with overloading can be translated into a language without overloading.

2.1.1 Class declarations

In Haskell 98, overloaded functions are declared as part of a *class declaration*:

class $Eq\ a$ **where**
$(\equiv) :: a \to a \to Bool$
$(\not\equiv) :: a \to a \to Bool$
$x \not\equiv y = not\ (x \equiv y)$
$x \equiv y = not\ (x \not\equiv y)$

This introduces a new *type class predicate* named Eq. Furthermore, two overloaded operators for equality ($\equiv$) and inequality ($\not\equiv$) are introduced. These operators are overloaded in the type variable a. The type variable a is scoped over the class signature and the operator signatures. Also, *default definitions* are given for both operators. Type classes can also be arranged in a hierarchy; we can declare for instance:

class $Eq\ a \Rightarrow Ord\ a$ **where**
$(\leqslant) :: a \to a \to Bool$

A class declaration consists of two parts; the part before the $\Rightarrow$ is called the *context* and the part after the $\Rightarrow$ is called the *head*. Here the context specifies that Eq is a *superclass* of Ord. The superclass relation must form a directed acyclic graph. In Figure 2.1 we present an overview of the standard Haskell classes.

2.1.2 Instance declarations

We can make a type an instance of a type class with the *instance declaration*. For example, the following instance declarations show how equality and ordering on booleans is defined and how equality on lists is defined.

instance $Eq\ Bool$ **where**
$True \equiv True = True$
$False \equiv False = True$
$\_ \equiv \_ = False$

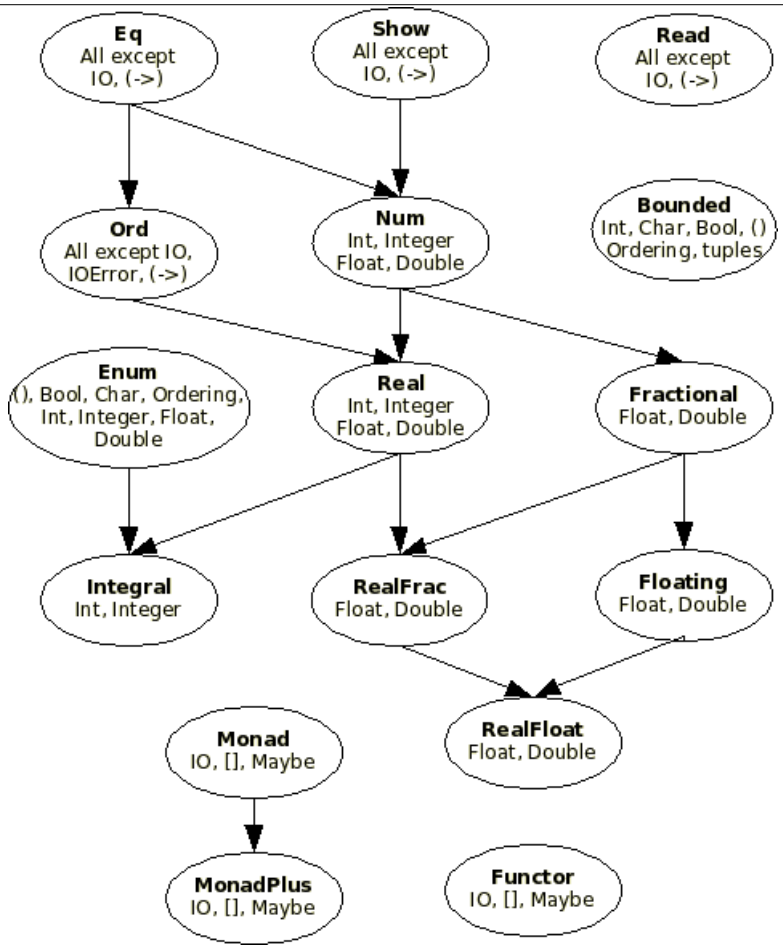

Figure 2.1: Standard Haskell classes

```
instance Ord Bool where
  _     ⩽ True  = True
  False ⩽ _     = True
  _     ⩽ _     = False
instance Eq a ⇒ Eq [a] where
  []       ≡ []       = True
  (x : xs) ≡ (y : ys) = x ≡ y ∧ xs ≡ ys
  _        ≡ _        = False
```

In the first instance declaration we define equality on booleans. We only have to specify a definition for equality, because we already have specified a default definition for inequality in the class declaration. The third instance means that once we have equality on elements ($Eq\ a$), we know how to provide equality on lists ($Eq\ [a]$). This can also be seen in the definition of ($\equiv$), first equality on the first elements of the lists is determined ($x \equiv y$), then the tails of the lists are compared for equality by a recursive call ($xs \equiv ys$). Note that $[[a]]$ and $[[[a]]]$ are now also instances of Eq, so this instance declaration makes an infinite number of types instances of Eq.

2.1.3 Functions and context reduction

The introduced type classes are used in the following functions for testing if a list is still ordered after an insertion:

```
insert :: Ord a ⇒ a → [a] → [a]
sort   :: Ord a ⇒ [a] → [a]
testInsert :: Ord a ⇒ a → [a] → Bool
testInsert x xs = let ys = insert x (sort xs)
                  in  sort ys ≡ ys
```

The signature contexts of the above functions mention the earlier introduce type class *Ord*. *Ord a* in the above signatures is a *type class predicate* or *type class qualifier*. This is an additional requirement on the type variable a, restricting a to types that are instances of *Ord*. When inferring the type of the function *testInsert*, we must prove the predicate *Ord a* because the functions *insert* and *sort* are used; luckily we may assume that there is an instance for *Ord a* because this predicate is also mentioned in the signature of *testInsert*. However, the overloaded operator ($\equiv$) is also used ($sort\ ys \equiv ys$), so we also have to prove the predicate $Eq\ [a]$. The reason why we do not find this predicate in the signature is that $Eq\ [a]$ can be derived from $Eq\ a$ via the instance declaration for lists and $Eq\ a$ can be derived from *Ord a* because *Eq* is a superclass of *Ord*. This process is called context reduction, which is an optimization that minimizes the number of predicates required in a type signature.

2.1.4 Dictionary translation

A Haskell program with overloading can be translated into a Haskell program without, using the dictionary translation. In Figure 2.2 we show the translation of the example code snippets introduced in this chapter. The class declarations are translated into datatype declarations and functions adding the default declarations to a dictionary. Note how the superclass/subclass relation between *Ord* and *Eq* is visible in the field *deq* of datatype *DictOrd*. The instance declarations are translated into functions constructing the dictionaries.

The predicate in the function *testInsert* is translated into an additional parameter. The caller of the function *testInsert* has to provide this parameter and it can be used in the body. Finally, the overloaded functions are selected from the dictionary parameter with exactly the same derivation as we saw when we performed context reduction.

```
data DictEq  a = DictEq  {(≡) :: a → a → Bool
                         , (≢) :: a → a → Bool}
data DictOrd a = DictOrd {(⩽) :: a → a → Bool
                         , deq :: DictEq a      }
defDictEq    :: DictEq a  → DictEq a
defDictEq    d            = DictEq   {(≢) = λx y → not (((≡) d) x y)
                                      , (≡) = λx y → not (((≢) d) x y)}
defDictOrd   :: DictOrd a → DictOrd a
defDictOrd   d            = DictOrd  {}
dictEqBool   :: DictEq Bool
dictEqBool   = let d = (defDictEq d) {(≡) = λx y → case (x, y) of
                                                     (True, True)  → True
                                                     (False, _)    → True
                                                     (_, _     )   → False}
               in d
dictOrdBool  :: DictOrd Bool
dictOrdBool  = let d = (defDictOrd d){(⩽) = λx y → case (x, y) of
                                                     (_, True)     → True
                                                     (False, False)→ True
                                                     (_, _     )   → False
                                     , deq = dictEqBool}
               in d
dictEqList   :: DictEq a → DictEq [a]
dictEqList da = let d = (defDictEq d){(≡) = λx y → case (x, y) of
                                                     ([], [])      → True
                                                     (x : xs, y : ys) → ((≡) da) x y ∧
                                                                        ((≡) d) xs ys
                                                     (_, _)        → False}
               in d
testInsert   :: DictOrd a → a → [a] → Bool
testInsert d x xs = let ys = insert d x (sort d xs)
                    in ((≡) (dictEqList (deq d))) (sort d ys) ys
```

Figure 2.2: Haskell program resulting from the dictionary translation

2.2 Caveats

2.2.1 Ambiguous types and defaulting

Using Haskell overloading can cause ambiguous types. An *ambiguous type* is a type of the form $\forall\ a.ctx \Rightarrow \tau$, where a is occurring in ctx, but not in τ [Peyton Jones, 2003]. For example, the following function has an ambiguous type:

$$
\begin{aligned}
&am :: (Show\ a, Read\ a) \Rightarrow String \\
&am = \textbf{let}\ x = read\ \texttt{"1.5"} \\
&\qquad \textbf{in}\ \ show\ x \\
&read\ \ :: Read\ \ a \Rightarrow String \rightarrow a \\
&show :: Show\ a \Rightarrow a \rightarrow String
\end{aligned}
$$

The ambiguity arises because the use of am does not specify to which type a should be instantiated. The reason for this is that a does not occur outside the context of the type scheme of am. This is a consequence of the implicit nature of Haskell's type inference system. Haskell provides several ways around this, which all make explicit to what type a should instantiate. For instance, a programmer can annotate a value with an explicit type, such as $(x :: Double)$. Because such ambiguities often occur in the context of class Num, Haskell provides a way to resolve them without explicit type annotations, the *default declaration*:

$$\textbf{default}\ (t_1, ..., t_n)$$

where $n \geqslant 0$ and t_n must be an instance of the Num class. An ambiguous type variable v can be defaulted if:

- v appears in a predicate of the form $C\ a$, where C is a class, and
- at least one of these classes is Num, or a subclass of Num,
- all of the these classes are defined in the Haskell 98 Prelude.

The ambiguous type variable v is defaulted to the first type t_i that is an instance of all the involved type classes.

2.2.2 The monomorphism restriction

In addition to the usual Hindley-Milner restriction that a type variable can only be generalized if it does not occur free in the environment, Haskell adds another restriction called the *monomorphism restriction* [Peyton Jones, 2003]. A constrained type variable is not generalized when a binding group is restricted. Type variables are constrained if there is a class predicate concerning that type variable. A binding group is unrestricted if:

- every variable in the group is bound by a function binding or a simple pattern binding, and
- an explicit type signature is given for every variable in the group that is bound by a simple pattern binding.

Any monomorphic type variables that remain when type inference for an entire module is complete, are considered ambiguous, and are resolved to particular types using the defaulting rules.

As a consequence, the inferred type for the function $f\ x\ y = x + y$ is what we expect: $Num\ a \Rightarrow a \rightarrow a \rightarrow a$. However, the type remains monomorphic if we define the same function with a simple pattern binding instead of a function binding: $f = \lambda x\ y \rightarrow x + y$. A type is defaulted using the default declaration if the type remains monomorphic after type inferencing the entire module. With the following default declaration:

$$\textbf{default}\ (Int, Double)$$

the type of the function f is defaulted to $Int \rightarrow Int \rightarrow Int$, because Int is the first type in the default declaration that is an instance of Num.

2.3 Entailment and qualified types

Type class predicates are an example of qualified types [Jones, 1992]. A qualifier places an extra restriction on type variables. For example, the qualifier in the signature of *testInsert*: $Ord\ a \Rightarrow a \rightarrow [a] \rightarrow Bool$, restricts

$$\frac{P \supseteq Q}{P \Vdash_e Q}\,(Mono) \qquad \frac{P \Vdash_e Q \quad Q \Vdash_e R}{P \Vdash_e R}\,(Trans) \qquad \frac{P \Vdash_e Q}{SP \Vdash_e SQ}\,(Closed)$$

Figure 2.3: Basic rules for qualifier entailment

$$\frac{P \Vdash_e \pi_1 \quad \pi_2 \in Q \qquad (Class\ Q \Rightarrow \pi_1) \in \Gamma}{P \Vdash_e \{\pi_2\}}\,(Super) \qquad \frac{P \Vdash_e Q \quad (Inst\ Q \Rightarrow \pi) \in \Gamma}{P \Vdash_e \{\pi\}}\,(Inst)$$

Figure 2.4: Additional rules for entailment relation of type class qualifiers

a to the members of type class *Ord*. Other qualifiers are used to type extensible records [Jones and Peyton Jones, 1999]:

- $(r\ has\ l :: a)$ means that record r has a field l of type a.
- $(r\ lacks\ l)$ means that the record r does not has a field l.

The *entailment* relation ($\Vdash_e$) describes the relation between two finite sets of qualifiers. The meaning of this relation depends on the qualifiers used. However, in Figure 2.3 we list three properties that hold for all qualifiers. In these rules P, Q, and R are sets of qualifiers and S is a substitution mapping type variables to types.

The specific properties for type class qualifiers are listed in Figure 2.4. For each class declaration: **class** $Q \Rightarrow \pi$, we add $(Class\ Q \Rightarrow \pi)$ to the environment (Γ) and for each instance: **instance** $Q \Rightarrow \pi$, we add $(Inst\ Q \Rightarrow \pi)$. Let us first consider the *Inst* rule: A set of predicates P entails π if there is an instance $Inst\ Q \Rightarrow \pi$ and $P \Vdash_e Q$. Consider the following example derivation to illustrate the *Inst* rule. We assume that the instance $(Inst\{Eq\ a\} \Rightarrow Eq\ [a])$ is in the environment.

$$\dfrac{\dfrac{(Inst\{Eq\ a\} \Rightarrow Eq\ [a]) \in \Gamma \qquad \dfrac{}{\{Eq\ a\} \Vdash_e \{Eq\ a\}}\,(Mono)}{\{Eq\ a\} \Vdash_e \{Eq\ [a]\}}\,(Inst)}{\{Eq\ v_1\} \Vdash_e \{Eq\ [v_1]\}}\,(Closed)$$

We search for a predicate π that matches the *head* of an instance when using the *Inst* rule. On the other hand, when using the *Super* rule, we search for a predicate π_2 that is part of the *context*. This is an important difference between the *Inst* and the *Super* rule, because the arrow in a class declaration could easily be misinterpreted as implication.

Chapter 3

Literature on type classes

This chapter summarizes a selection of literature concerning type classes. First, to give some background information for this thesis, and furthermore, to make a clear separation between our work and work of others. We give a literature overview focusing on two aspects of type classes:

- Design; the design of type classes and various extensions to type classes.
- Constraints; the resolving of overloading using a constraint solver.

3.1 Design of type classes

Type classes were introduced by Wadler and Blot [Wadler and Blott, 1989] and Kaes [Kaes, 1988]. Their type class design was adopted by Haskell [Peyton Jones, 2003]. In this chapter we give a chronological overview of the various papers that have appeared about the design of type classes.

3.1.1 A system of constructor classes

The first extension, called constructor classes, was proposed by Jones [Jones, 1993]. This extension also allows type constructors to be in a class. For example, the type class *Monad* is defined for types of kind $* \rightarrow *$, with instances for $[\,]$ and *Maybe*:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
instance Monad Maybe where
  return x = Just x
  n >>= f  = case n of
               Nothing → Nothing
               Just x  → f x
```

Here m in the definition of the class *Monad* is a type of kind $* \rightarrow *$, a type-level function from proper types to proper types. The type expression $m\ a$ means m applied to a. *Maybe* and $[\,]$ are both of kind $* \rightarrow *$ so we can make them instances of *Monad*. This extension is part of Haskell 98.

3.1.2 A second look at overloading

Odersky, Wadler, and Wehr [Odersky et al., 1995] describe a simple restriction on type classes which ensures that no ambiguities can arise. The restriction is that each function that is overloaded over type variable a must have a type of the form $a \rightarrow t$, where t may itself involve a. Their redesign of type classes is called *System O*. Below we present a System O example which we cannot express with Haskell 98 type classes:

```
over first
inst  first :: (a, b) → a
```

$$
\begin{array}{ll}
 & \mathit{first}\ (x, y) = x \\
\textbf{inst} & \mathit{first} :: (a, b, c) \to a \\
 & \mathit{first}\ (x, y, z) = x \\
\mathit{ff} :: (\mathit{first} :: a \to b, \mathit{first} :: b \to c) \Rightarrow a \to c & \\
\mathit{ff}\ r = \mathit{first}\ (\mathit{first}\ r) &
\end{array}
$$

Note that we can express this example using multi-parameter type classes and functional dependencies which we will explain later. System O has a number of advantages: one does not have to decide in advance which operations belong together in a class and the definition of overloaded operators is less verbose. On the other hand, the inferred types become more verbose because each overloaded function is mentioned separately in the context.

3.1.3 Type classes: an exploration of the design space

Peyton Jones, Jones, and Meijer [Peyton Jones et al., 1997] argue that the original design of type classes is fairly conservative. They discuss a number of design choices for each of the following aspects: multi-parameter type classes, context reduction, overlapping instances, instance types, instance contexts, superclasses, improving substitution, and class declarations. In the conclusion the authors take the following design decisions:

- Allow multi-parameter type classes.
- No limitations on superclass contexts.
- Allow the class variables to be constrained in the class-member type signatures.
- Forbid overlapping instance declarations.
- Allow arbitrary instance types in the head of an instance declaration, except that at least one must not be a type variable.
- Allow repeated type variables in the head of an instance declaration.
- Restrict the context of an instance declaration to mention type variables only.
- Allow arbitrary contexts in types and type signatures.
- Use the instance context reduction rule only when forced by a type signature, or when the overloading can be resolved at compile time.

The Haskell compiler GHC [Marlow and Peyton Jones, 2006] implements these design choices when enabling extensions. Even overlapping instances are allowed with an additional flag. Furthermore, the design choice that the context of an instance declaration may only mention type variables is relaxed. However, the predicates in the context must have fewer constructors and variables (taken together and counting repetitions) than the head.

3.1.4 Type classes with functional dependencies

Jones [Jones, 2000] shows that the use of multiple-parameter type classes often causes ambiguities and inaccuracies. To tackle this problem the author introduces *functional dependencies* between parameters of type classes. In the example below: $a \to b$ (a uniquely determines b), means that the relation between a and b is a function. This means, for example, that it is forbidden to define both the instance $D\ Int\ Bool$ and $D\ Int\ String$.

$$
\begin{array}{l}
\textbf{class}\ C\ a\ b \\
\textbf{class}\ D\ a\ b \mid a \to b \\
\textbf{class}\ E\ a\ b \mid a \to b, b \to a
\end{array}
$$

In the last class declaration the relation between a and b is a one-to-one mapping. So if there is an instance $E\ Int\ Bool$ there cannot be another instance where either a is Int, or b is $Bool$.

The example we gave for System O can also be expressed with multi-parameter type classes and functional dependencies:

$$
\begin{array}{l}
\textbf{class}\ \mathit{First}\ a\ b \mid a \to b\ \textbf{where} \\
\quad \mathit{first} :: a \to b \\
\textbf{instance}\ \mathit{First}\ (a, b)\ a\ \textbf{where} \\
\quad \mathit{first}\ (x, y) = x \\
\textbf{instance}\ \mathit{First}\ (a, b, c)\ a\ \textbf{where}
\end{array}
$$

$$
\begin{aligned}
&\quad first\ (x, y, z) = x \\
&ff :: (First\ a\ b, First\ b\ c) \Rightarrow a \rightarrow c \\
&ff\ x = first\ (first\ x)
\end{aligned}
$$

3.1.5 Making implicit parameters explicit

Dijkstra and Swierstra [Dijkstra and Swierstra, 2005] describe how dictionaries can be passed explicitly to overloaded functions. Consider the function below that checks if a value is an element of a list. The predicate $Eq\ a$ in the type of the function *elem* corresponds to an implicit argument, the dictionary for the predicate $Eq\ a$. If we use this function we do not have to supply the dictionary for $Eq\ a$, it is automatically inserted by the compiler.

$$
\begin{aligned}
&elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\
&elem\ x\ [\,] \quad\quad = False \\
&elem\ x\ (y : ys) = x \equiv y \vee elem\ x\ ys \\
&test\ xs = elem\ 4\ xs
\end{aligned}
$$

However, if we want to use this function with another equality, for example modulo 2, we cannot specify this, because it is not possible to influence the machinery that automatically inserts the dictionary. With the extension described by Dijkstra and Swierstra it is possible to insert the dictionary implicitly and explicitly. We can for example define a named instance of *Eq Int* where equality is defined modulo 2.

$$
\begin{aligned}
&\textbf{instance}\ dEqIntMod :: Eq\ Int\ \textbf{where} \\
&\quad x \equiv y = primEqInt\ (x\ `mod`\ 2)\ (y\ `mod`\ 2) \\
&test\ xs = elem\ \{!dEqIntMod <: Eq\ Int!\}\ 4\ xs
\end{aligned}
$$

Now we can explicitly insert the named dictionary between the $\{!\ldots!\}$ signs, where otherwise the normal dictionary for equality would be inserted. The authors present three different instance declaration:

$$
\begin{aligned}
&\textbf{instance}\ \quad\quad\quad\quad Eq\ Int\ \textbf{where}\ \ldots \\
&\textbf{instance}\ dEqIntMod <: Eq\ Int\ \textbf{where}\ \ldots \\
&\textbf{instance}\ dEqIntMod :: \ Eq\ Int\ \textbf{where}\ \ldots
\end{aligned}
$$

The first two instance declarations participate in the machinery for automatic dictionary translation, and the last one does not. Additionally the last two instance declarations can be used to explicitly insert dictionaries by referring to the introduced identifier to which the dictionary is bound.

3.2 Constraints and type classes

3.2.1 Type classes and constraint handling rules

Glynn, Stuckey, and Sulzmann [Glynn et al., 2000, Sulzmann, 2006] describe how class and instance declarations can be translated into Constraint Handling Rules (CHRs). Furthermore, the authors show how instance declarations can be checked with CHRs and how predicates can be simplified using CHRs. CHRs are a high level declarative language extension especially designed for writing constraint solvers [Frühwirth, 1998]. CHRs are often embedded in a host language: constraints are defined in CHRs, but auxiliary computations are executed in the host language. CHRs operate on a set of constraints and rewrite constraints into simpler ones until they are solved.

The following propagation CHR is generated for a class declaration of the form: **class** $Q \Rightarrow \pi$.

$$\pi \Longrightarrow Q$$

This propagation constraint means that the constraints Q can be inserted if π is an element of the constraint set. A propagation CHR is applied only once to each constraint to prevent non-termination. The following simplification CHR is generated for an instance declaration of the form: **instance** $Q \Rightarrow \pi$.

$$\pi \Longleftrightarrow Q$$

This simplification CHR means that π can be *replaced* with Q when π is an element of the constraint set. If the set Q is empty it is abbreviated with *true*. CHRs are generated for all instance and class declarations in

a program. Instance declarations are correct when the generated CHRs form a confluent set. A set of rules is *confluent* if from any given constraint set every possible order of rule applications ends in the same final constraint set. For example, this is not the case with overlapping instances:

data $Foo = Foo$
instance $Eq\ [Foo]$ **where**
  $xs \equiv ys = length\ xs \equiv length\ ys$

The following CHRs are generated for the above instance declaration and the standard instance declaration for lists:

$Eq\ [a] \iff Eq\ a$
$Eq\ [Foo] \iff true$

These rules are non-confluent because there are two possible derivations for solving the constraint $Eq\ [Foo]$. However, if we would allow overlapping instances then we typically choose the most specific instance. The most specific instance is **instance** $Eq\ [Foo]$ in this case.

The CHRs generated for class and instance declarations can be used to perform context reduction. However, an additional step is needed to simplify constraints using the class hierarchy. The following CHRs are generated for a class declaration of the form **class** $(p1, ..., \pi_n) \Rightarrow \pi$ to simplify constraints using the class hierarchy:

$\pi, \pi_i \iff \pi$

The CHRs resulting from the first translation simplify constraints using instance declarations and propagate the class hierarchy to check instance declarations. The CHRs resulting from the second translation simplifies constraints using the class hierarchy. To perform context reduction, constraints are first solved using the CHRs resulting from the first translation and the result is solved with the CHRs resulting from the second translation. The authors do not explain how explicit type signatures are checked with CHRs.

3.2.2 Implementing overloading in Chameleon

Stuckey and Sulzmann present a minimal extension of the Hindley/Milner system to support overloading of identifiers [Stuckey and Sulzmann, 2002]. This approach relies on a combination of the HM(X) type system framework with CHRs. This proposal provided the basis for the Chameleon language described by Sulzmann and Wazny [Sulzmann and Wazny, 2001]. Chameleon supports only single overloaded definitions, so it is not possible to group overloaded identifiers in classes and arrange those classes in a hierarchy. However, the programmer can specify arbitrary additional conditions by providing CHR propagation rules in the program text. With these user provided CHRs it is possible to mimic Haskell class hierarchies. Here we will focus on the translation scheme described to resolve overloading in Haskell.

The authors use the same translation scheme as described in the previous subsection. Consider the following example presented in the Chameleon paper:

class $Eq\ a$ **where**
  $(\equiv) :: a \to a \to Bool$
class $Eq\ b \Rightarrow Foo\ a\ b$ **where**
  $foo :: b \to a$
instance $Eq\ a \Rightarrow Foo\ [a]\ a$

The following CHRs are generated for the above class and instance declarations:

$Foo\ a\ \ b \implies (Eq\ b)_1$ -- (Foo)
$Foo\ [a]\ a \iff (Eq\ a)_2$ -- (FooInst)

A propagation CHR is generated for the declaration of class *Foo* and a simplification CHR is generated for the instance declaration. The overloaded functions $(\equiv)$ and *foo* are used in the following definition:

$f :: (Foo\ a\ b)_6 \Rightarrow b \to (Bool, a)$
$f\ x = (foo_4\ [x]\ \equiv_3\ x, foo_5\ x)$

Analyzing the function f results in the following constraints:

$\{(Eq\ b)_3, (Foo\ [b]\ b)_4, (Foo\ a\ b)_5, (Foo\ a\ b)_6\}$

Each constraint is annotated with a location identifier. Applying the generated CHRs to the constraints results in the following constraint set:

$$\{(Eq\ b)_3, (Eq\ b)_{4,2}, (Foo\ a\ b)_5, (Eq\ b)_{5,1}$$
$$, (Foo\ a\ b)_6, (Eq\ b)_{6,1}\}$$

The constraints are annotated with justifications to keep track of the history of CHR applications. The first four constraints are called inferred constraints whereas the last two constraints result from the type signature. Propagated inferred constraints are discarded. An example of such a discarded constraint is $(Eq\ b)_{5,1}$ which results from applying the propagation rule 1 to the constraint at location 5. The last step is to match the remaining inferred constraints against annotated constraints: $(Foo\ a\ b)_5$ is matched against $(Foo\ a\ b)_6$ and $(Eq\ b)_3, (Eq\ b)_{4,2}$ against $(Eq\ b)_{6,1}$. The justifications are used to generate evidence for the remaining inferred constraints.

The authors admit that there is some non-determinism when matching inferred constraints against annotated constraints and solving constraints with CHRs. However, they do not remove this non-determinism because the different solutions denote the same result.

3.2.3 CHRs and functional dependencies

Duck, Peyton Jones, Stuckey, and Sulzmann give a reformulation of functional dependencies in terms of CHRs [Duck et al., 2004, Sulzmann et al., 2007]. For example, consider the following class and instance declarations:

class $Coll\ c\ e \mid c \to e$ **where**
  $empty :: c$
  $insert :: e \to c \to c$
  $member :: e \to c \to Bool$
instance $Ord\ a \Rightarrow Coll\ [a]\ a$ **where**
  $empty = [\,]$
  $insert = (:)$
  $member = elem$

The improvement rules for the functional dependency $c \to e$ are expressed with the following CHRs:

$$Coll\ c\ e, Coll\ c\ d \Longrightarrow e \equiv d$$
$$Coll\ [a]\ b \Longrightarrow a \equiv b$$

The first rule is generated for the functional dependency $c \to e$ and expresses that if there are two constraints over the same type c, then it must be that e and d are the same type. The second rule is generated for the instance declaration. The authors have verified that the termination and consistency conditions [Jones, 2000] are sufficient to guarantee sound and decidable type inference. They even show how those conditions can be safely relaxed.

3.2.4 Type class directives

Heeren and Hage [Heeren and Hage, 2005] propose a number of directives to improve type error messages concerning Haskell type classes. The authors introduce directives besides class and instance declarations to describe type classes. These directives are used to improve error messages concerning type classes. With these directives one is able to exclude a type from a type class, allow only a finite number of types in a type class, and express that the types in two type classes are disjoint. Below are some example directives:

```
never    Eq (a → b)          :"functions cannot be tested for equality"
never    Num Bool            :"arithmetic on booleans isn't supported"
close    Integral            :"the only Integral instances are Int and Integer"
disjoint Integral Fractional :"something which is fractional can never be integral"
```

Directives are checked during context reduction and the associated error message is presented if a directive does not hold.

Chapter 4

Overloading in Helium

Helium [Heeren et al., 2003b] is a compiler designed with the emphasis on high quality type error messages. Helium is suitable for teaching students functional programming because the type errors are clear and precise. It supports most of Haskell 98: the most notable omission is that the user cannot specify class and instance declarations. However, Helium supports overloading with a finite set of built-in classes and instances. Helium uses the constraint solver Top [Heeren, 2005] for type inferencing and checking. In this chapter we will focus on how overloading is resolved in Helium and Top.

4.1 The type system

The type system of Helium is rather standard and has the following structure:

Type:

τ	::=	a	(type variable)
	|	T	(type constant)
	|	$\tau_1\ \tau_2$	(type application)

Context:

P	::=	$(\pi_1, ..., \pi_n)$	$(n \geqslant 0)$

Type class qualifier:

π	::=	$C\ \tau$

Type scheme:

σ	::=	$\forall\ \bar{a}.P \Rightarrow \tau$

The types in this type system are partitioned in polymorphic types (σ) and monomorphic types (τ). A type variable is an identifier starting with a lowercase letter. In this chapter we use the infinite list of identifiers $(v_1, ..., v_n)$ as type variables. Both a type constant (T) and a class name (C) are identifiers starting with an uppercase letter. Type application allows us to write useless types like ($Int\ Bool$). Actually, we need a kind system to check well-formedness of types, but we assume that we only deal with well-formed types. Type schemes with an empty context: $\forall\ \bar{a}.() \Rightarrow \tau$ are written as: $\forall\ \bar{a}.\tau$.

4.2 Constraint based type inferencing

Helium collects type constraints by analyzing the abstract-syntax tree of a Haskell program. This results in a tree of constraints which has the same shape as the abstract-syntax tree of the program. This tree is flattened into a list using a flattening strategy. The resulting constraints are solved by Top.

In this section we explain how these constraints are solved. First, we consider equality constraints, and then we show how constraints for polymorphism are solved.

4.2.1 The equality constraint

An equality constraint specifies that two monomorphic types must be equal:

Constraints:
$c \quad ::= \quad \tau_1 \equiv \tau_2 \quad$ (equality)

For instance, consider the following function:

$f\ x\ y\ z =$ **if** x
then y
else z

Analyzing the if-then-else part of this function yields the following result:

$\mathcal{A} = [x \ \mapsto \ v_1, y \ \mapsto \ v_2, z \ \mapsto \ v_3]$
$\mathcal{C} = \{v_1 \equiv \mathit{Bool}$
$, v_2 \equiv v_4$
$, v_3 \equiv v_4\}$

$\mathcal{A}$ is an assumption set relating monomorphic types to identifiers and $\mathcal{C}$ is the constraint set. These constraints specify that the guard must have type *Bool* and that the then and else branch must have the same type. The solver maintains a state while solving these constraints. This state consists of a substitution which maps type variables to types:

$\mathit{state} \quad ::= \quad S$
$S \quad ::= \quad [v_1 \ \mapsto \ \tau_1, ..., v_n \ \mapsto \ \tau_n]$

Helium solves equality constraints by generating substitutions under which the two types are syntactically equivalent. The mapping $v_1 \ \mapsto \ \mathit{Bool}$ is added to the substitution to solve the first constraint of the example. The second constraint has two solutions: either v_2 is mapped to v_4, or v_4 is mapped to v_2. The same holds for the last constraint. Which of the two solutions is chosen does not matter. Helium chooses to map the left hand side to the right hand side. This results in the following solution for the list of example constraints:

$S = [v_1 \ \mapsto \ \mathit{Bool}, v_2 \ \mapsto \ v_4, v_3 \ \mapsto \ v_4]$

It is not allowed for a type variable of one component of an equality constraint to occur in the other component. For example consider the following constraint:

$v_5 \equiv \mathit{Maybe}\ v_5$

One could come up with the substitution $[v_5 \ \mapsto \ \mathit{Maybe}\ v_5]$ as solution, but this will result in an infinite type. In other words, the substitution must be idempotent. This restriction is called 'the occurs check'.

4.2.2 Constraints for polymorphism

To support polymorphism we need to add another layer to the type system. Beside monomorphic types (τ) and polymorphic types (σ) we add ρ types which are either type schemes (σ) or type scheme variables (σ_v). Furthermore, we extend the state with an environment which maps type scheme variables to type schemes:

$\mathit{state} ::= (S, \Sigma)$
$\Sigma \quad ::= [s_1 \ \mapsto \ \sigma_1, ..., s_n \ \mapsto \ \sigma_n]$

We use the infinite list of identifiers $(s_1, ..., s_n)$ as type scheme variables. Also, three constraints are added to the constraint language to support polymorphism:

Constraints:

c	$::=$	$\tau_1 \equiv \tau_2$	(equality)
	$\mid$	$\sigma_v := \mathit{Gen}\ (\mathcal{M}, \tau)$	(generalization)
	$\mid$	$\tau := \mathit{Inst}\ (\rho)$	(instantiation)
	$\mid$	$\tau := \mathit{Skol}\ (\mathcal{M}, \rho)$	(skolemization)

In the remainder of this subsection we will explain in more detail how polymorphism constraints are used and solved.

The generalization constraint

A generalization constraint generalizes a type with respect to a set of type variables that should remain monomorphic ($\mathcal{M}$). For instance, Helium generates the following constraints for the function: $id\ x = x$.

$$\begin{aligned}\mathcal{A} &= [\,id \mapsto v_0, x \mapsto v_3\,]\\ \mathcal{C} &= \{\,v_0 \equiv v_2 \to v_1\\ &\quad\ , v_3 \equiv v_2\\ &\quad\ , v_3 \equiv v_1\\ &\quad\ , id := \mathit{Gen}\ ([\,], v_0)\,\}\end{aligned}$$

The first constraint states that the binding id is a function. The second constraint states that the variable x should have the same type as the argument of the function. The third constraint states that x should have the same type as the result of the function. The last constraint generalizes the binding and maps the type-scheme variable id to the resulting type scheme. Consider the state of the solver just before considering the last constraint:

$$\begin{aligned}S &= [\,v_0 \mapsto v_1 \to v_1, v_3 \mapsto v_1, v_2 \mapsto v_1\,]\\ \Sigma &= [\,]\end{aligned}$$

Solving a generalization constraint $\sigma_v := \mathit{Gen}\ (\mathcal{M}, \tau)$ consists of four steps:

- First, the substitution is applied to the set of type variables that should remain monomorphic ($\mathcal{M}$) and the type τ. As a result the constraint becomes: $id := \mathit{Gen}\ ([\,], v_1 \to v_1)$.
- Second, the type variables that are going to be generalized are computed: $\bar{a} = \mathit{ftv}\ (\tau) - \mathit{ftv}\ (\mathcal{M})$, in this case v_1.
- The third step is to generalize the type τ over the set of type variables $\bar{a}$.
- And finally the generalized type is stored in the environment under the type-scheme variable σ_v, in this case id.

For our current example this will result in the following state:

$$\begin{aligned}S &= [\,v_0 \mapsto v_1 \to v_1, v_3 \mapsto v_1, v_2 \mapsto v_1\,]\\ \Sigma &= [\,id \mapsto \forall\ a.a \to a\,]\end{aligned}$$

Under this solution the equality constraints hold and the type for the function id is inferred to be $\forall\ a.a \to a$.

The instantiation constraint

An instantiation constraint states that type τ should be an instance of a type scheme. For example, the following constraints are generated for the expression ($id\ 1, id$ `'c'`):

$$\begin{aligned}\mathcal{A} &= [\,]\\ \mathcal{C} &= \{\,v_4 := \mathit{Inst}\ (\forall\ a.a \to a)\\ &\quad\ , v_4 \equiv \mathit{Int} \to v_3\\ &\quad\ , v_7 := \mathit{Inst}\ (\forall\ a.a \to a)\\ &\quad\ , v_7 \equiv \mathit{Char} \to v_6\\ &\quad\ , (v_3, v_6) \equiv v_2\,\}\end{aligned}$$

The polymorphic function id is instantiated twice: once for an argument of type *Int* and once for an argument of type *Char*.

Solving an instantiation constraint $\tau := \mathit{Inst}\ (\rho)$ consists of three steps:

- First, if ρ is a type scheme variable, the type scheme is looked up in the environment. In this example the type schemes are already known.
- Second, the type scheme is instantiated by replacing the universally ($\forall$) quantified variables with fresh type variables.
- Finally, an equality constraint is generated between τ and the instantiated type.

Consider the first instantiation constraint of the above example. The type scheme of this constraint is instantiated with the fresh type variable v_{10}. This results in the equality constraint $v_4 \equiv v_{10} \to v_{10}$ which means that v_4 is a function with one argument where the type of the argument is equal to the type of the

result. In the same way, the type scheme of the second instantiation constraint is instantiated with the fresh type variable v_{11} resulting in the constraint $v_7 \equiv v_{11} \rightarrow v_{11}$. Solving the equality constraints results in the following solution:

$$S = [v_2 \mapsto (Int, Char), v_3 \mapsto Int, v_4 \mapsto Int \rightarrow Int, v_6 \mapsto Char, v_7 \mapsto Char \rightarrow Char, v_{10} \mapsto Int, v_{11} \mapsto Char]$$
$$\Sigma = [\,]$$

Under this solution the equality constraints holds.

The skolemization constraint

A skolemization constraint states that type τ should be equal to a skolemized type scheme. A type scheme can be skolemized by instantiating it with fresh type constants. We use v for type variables and c for type constants. The difference between the two is that a type constant cannot be unified with another type constant whereas a type variable can be unified with a type constant. Let us define the identity function again, but now with an explicit type signature.

$$id :: \forall\ a.a \rightarrow a$$
$$id\ x = x$$

Helium generates the following constraints for this function:

$$\mathcal{A} = [id \mapsto v_0, x \mapsto v_3]$$
$$\mathcal{C} = \{v_0 \equiv v_2 \rightarrow v_1, v_0 := Skol\ ([\,], \forall\ a.a \rightarrow a), v_3 \equiv v_2, v_3 \equiv v_1\}$$

If we compare these constraint to the constraints generated when not giving a type signature, a skolemization constraint is generated instead of a generalization constraint. Also the order of the constraints differ: the generalization constraint did appear after the constraints for analyzing the body of the function *id*, but the skolemization constraint appears before. In fact the generalization constraint *must* appear after the constraints for analyzing the body, but the position of the skolemization constraint does not matter.

The following three steps are needed to solve a skolemization constraint $\tau := Skol\ (\mathcal{M}, \rho)$:

- First, if ρ is a type-scheme variable, the type scheme is looked up in the environment.
- Second, the universally ($\forall$) quantified variables are replaced with fresh type constants.
- Third, the fresh type constants and $\mathcal{M}$ are stored together so that we can check afterwards that the type constants do not escape via a type variable in $\mathcal{M}$.
- Finally, an equality constraint between τ and the skolemized type is generated.

The type scheme of the constraint $v_0 := Skol\ ([\,], \forall\ a.a \rightarrow a)$ is skolemized with the fresh type constant c_5. The last step in solving the skolemization constraint is that we generate the constraint: $v_0 \equiv c_5 \rightarrow c_5$. Solving the list of equality constraints will result in the following substitution:

$$S = [v_0 \mapsto c_5 \rightarrow c_5, v_1 \mapsto c_5, v_2 \mapsto c_5, v_3 \mapsto c_5]$$

4.3 Constraints for overloading

In this section we explain how overloading is resolved in Top. To support overloading we extend the constraint language with two additional constraints:

Constraints:

c	::=	$\tau_1 \equiv \tau_2$	(equality)
	|	$\sigma_v := Gen\ (\mathcal{M}, \tau)$	(generalization)
	|	$\tau := Inst\ (\rho)$	(instantiation)
	|	$\tau := Skol\ (\mathcal{M}, \rho)$	(skolemization)
	|	$Prove\ (\pi)$	*(prove qualifier)*
	|	$Assume\ (\pi)$	*(assume qualifier)*

The *Prove* constraint means that we have the obligation to proof the qualifier π. This means that evidence should be constructed of type π at the location in the abstract-syntax tree where the *Prove* constraint occurs. An *Assume* constraint means that we can assume the qualifier π. This means that evidence is available of type π.

To solve these constraint we extend the state with the following components:

$$\begin{array}{lll} state & ::= & (S, \Sigma, \Pi_{prove}, \Pi_{assume}, \Pi_{gen}) \\ \Pi & ::= & \{\pi_1, ..., \pi_n\} \end{array}$$

Π_{prove} is a set of qualifiers that must be proven and Π_{assume} is a set of qualifiers that can be assumed. In the remainder of this chapter we explain what the meaning of the list of qualifiers Π_{gen} is.

4.3.1 Generating overloading constraints

Assume and *Prove* constraints are not generated by Helium, but result from solving instantiation and skolemization constraints. Consider the following example to illustrate this:

$$\begin{array}{l} elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\ elem\ x\ [\,] \quad\quad = False \\ elem\ x\ (y : ys) = x \equiv y \vee elem\ x\ ys \end{array}$$

Notice that this function has an explicit type signature and that the overloaded functions ($\equiv$) and *elem* are used in the body of *elem*. In this example we do not consider every generated constraint, but only the following skolemization and instantiation constraints:

$$\begin{array}{l} \mathcal{C} = \{v_0 := Skol\ ([\,], \forall\ a.Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool) \\ \quad\quad , v_1 := Inst\ (\forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool) \\ \quad\quad , v_2 := Inst\ (\forall\ a.Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool)\} \end{array}$$

First, the type signature is skolemized: $v_0 := Skol\ ([\,], \forall\ a.Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool)$. The quantified type variable is replaced with a fresh type constant: $Eq\ c_3 \Rightarrow c_3 \rightarrow [c_3] \rightarrow Bool$. Finally, an *Assume* constraint is generated for every qualifier in the skolemized type. So for the above type signature the constraint $Assume\ (Eq\ c_3)$ is generated.

Second, the function ($\equiv$) is instantiated: $v_1 := Inst\ (\forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool)$. The quantified type is instantiated with a fresh type variable: $Eq\ v_4 \Rightarrow v_4 \rightarrow v_4 \rightarrow Bool$. Finally, a *Prove* constraint is generated for every qualifier in the instantiated type. So for the first instantiation constraint $Prove\ (Eq\ v_4)$ is generated. Likewise, the quantified type of the third constraint is instantiated using the fresh type variable v_5 and the constraint $Prove\ (Eq\ v_5)$ is generated. The type variables v_4 and v_5 will be substituted to c_3 later in the type inference process.

To sum it up, instantiation generates a *Prove* constraint for every qualifier of the instantiated type scheme. Skolemization generates an *Assume* constraint for every qualifier of the skolemized type scheme.

4.3.2 Solving overloading constraints

Solving a *Prove* or an *Assume* constraint is rather straightforward. For a *Prove* constraint the qualifier π is inserted into the set of qualifiers to prove (Π_{prove}). In the same way the qualifier of an *Assume* constraint is inserted into the set of qualifiers that can be assumed (Π_{assume}).

Simplification

The question that arises now is what happens with the set of assumed qualifiers and the set of qualifier to proof . The answer is that the constraints are solved during the simplification step. In principle this step can be applied at any point in the solving process. Top executes this step just before generalizing and just after solving the last constraint. For example, consider the function *elem* again:

$$\begin{array}{l} elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\ elem\ x\ [\,] \quad\quad = False \\ elem\ x\ (y : ys) = x \equiv y \vee elem\ x\ ys \end{array}$$

When the constraints for the above function are solved the state is:

$$\begin{array}{ll} S & = [v_1 \mapsto c_1, v_2 \mapsto c_1, v_0 \mapsto c_1 \to [c_1] \to \mathit{Bool}, \mathit{v18} \mapsto c_1 \to c_1 \to \mathit{Bool}, ...] \\ \Sigma & = [\,] \\ \Pi_{\mathit{prove}} & = \{\, \mathit{Eq}\ v_1, \mathit{Eq}\ v_2 \,\} \\ \Pi_{\mathit{assume}} & = \{\, \mathit{Eq}\ c_1 \,\} \\ \Pi_{\mathit{gen}} & = \{\,\} \end{array}$$

The last step of the constraint solver is to simplify the set of qualifiers to prove (Π_{prove}). The set of predicates to prove must be empty when the solver is finished, otherwise overloading is not resolved. Simplification consists of three steps. First, the substitution is applied to the qualifiers. In our example the type variables are substituted to the type variable constant c_1:

$$\begin{array}{ll} \Pi_{\mathit{prove}} & = \{\, \mathit{Eq}\ c_1 \,\} \\ \Pi_{\mathit{assume}} & = \{\, \mathit{Eq}\ c_1 \,\} \end{array}$$

Second, the prove qualifiers are simplified using instance declarations and the class hierarchy. The last step is that the prove qualifiers entailed by the assume qualifiers are removed. In this example the assume qualifier $\mathit{Eq}\ c_1$ entails the prove qualifier $\mathit{Eq}\ c_1$, because they are equal. This means that the prove qualifier $\mathit{Eq}\ c_1$ can be removed. There are no prove qualifiers left so overloading is resolved in the function *elem* and the simplification step is finished.

Generalization

Simplification is part of the steps needed to solve the generalization constraint $\sigma_v := \mathit{Gen}\ (\mathcal{M}, \tau)$. Consider the following example:

$$\begin{array}{l} f\ x\ y = \textbf{if}\ x > y \\ \qquad\quad \textbf{then}\ \mathit{show}\ x \\ \qquad\quad \textbf{else}\ \ g\ x\ y \\ g\ x\ y = f\ (\mathit{succ}\ x)\ y \end{array}$$

For the above program the following constraints are generated; we only consider the generalize and instantiation constraints.

$$\begin{array}{lll} \mathcal{A} = [x \mapsto v_1, y \mapsto v_1] & & \\ \mathcal{C} = \{ v_1 \to v_1 \to \mathit{Bool} & := \mathit{Inst}\ (\forall\ a.\mathit{Ord}\ a \Rightarrow a \to a \to \mathit{Bool}) & \text{-- (>)} \\ \quad , v_1 \to \mathit{String} & := \mathit{Inst}\ (\forall\ a.\mathit{Show}\ a \Rightarrow a \to \mathit{String}) & \text{-- (show)} \\ \quad , v_1 \to v_1 & := \mathit{Inst}\ (\forall\ a.\mathit{Enum}\ a \Rightarrow a \to a) & \text{-- (succ)} \\ \quad , f & := \mathit{Gen}\ ([\,], v_1 \to v_1 \to \mathit{String}) & \\ \quad , g & := \mathit{Gen}\ ([\,], v_1 \to v_1 \to \mathit{String})\,\} & \end{array}$$

Because the functions f and g are mutually recursive, the constraints of the body's of the functions are analyzed before both functions are generalized. The first constraint is generated for the instantiation of the operator (>), the second for the function *show*, and the third for the function *succ*. The last two constraints generalize f and g, respectively.

There are eight steps needed to solve a generalization constraint $\sigma_v := \mathit{Gen}\ (\mathcal{M}, \tau)$:

1. *simplification*
2. *substitution*
3. $\bar{a} = \mathit{ftv}\ (\tau) - \mathit{ftv}\ (\mathcal{M})$
4. $\Pi_1 = \{\pi \mid \pi \leftarrow \Pi_{\mathit{prove}}, (\mathit{ftv}\ \pi\ \cap\ \bar{a}) \not\equiv [\,]\}$
5. $\Pi_2 = \{\pi \mid \pi \leftarrow \Pi_{\mathit{gen}}\ \ , (\mathit{ftv}\ \pi\ \cap\ \bar{a}) \not\equiv [\,]\}$
6. $\Pi_{\mathit{prove}} = \Pi_{\mathit{prove}} - \Pi_1$
7. $\Pi_{\mathit{gen}} = \Pi_{\mathit{gen}}\ \cup\ \Pi_1$
8. $\sigma_v \mapsto \forall\ \bar{a}.(\Pi_1\ \cup\ \Pi_2) \Rightarrow \tau$

We first solve the first generalization constraint: $f := \mathit{Gen}\ ([\,], v_1 \to v_1 \to \mathit{String})$. The overloading state before solving this constraint is:

$$\begin{array}{ll} \Pi_{\mathit{prove}} & = \{\, \mathit{Ord}\ v_1, \mathit{Show}\ v_1, \mathit{Enum}\ v_1 \,\} \\ \Pi_{\mathit{assume}} & = \{\,\} \\ \Pi_{\mathit{gen}} & = \{\,\} \end{array}$$

First, we simplify the predicates with the earlier described simplification step and then we apply the substitution. The type variable over which we can generalize is v_1. This results in the following sets of qualifiers:

$\Pi_1 = \{\, Ord\ v_1, Show\ v_1, Enum\ v_1 \,\}$
$\Pi_2 = \{\,\}$

The set Π_1 is subtracted from the set of proof predicates and inserted into the set of predicates that have been generalized. Finally, the generalized type is: $\forall\ a.(Ord\ a, Show\ a, Enum\ a) \Rightarrow a \rightarrow a \rightarrow String$. Before we consider the last constraint: $g := Gen\ ([\,], v_1 \rightarrow v_1 \rightarrow String)$, the overloading state is:

$\Pi_{prove} = \{\,\}$
$\Pi_{assume} = \{\,\}$
$\Pi_{gen} = \{\, Ord\ v_1, Show\ v_1, Enum\ v_1 \,\}$

Again the type variable over which we can generalize is v_1. The difference now is that filtering the set of prove qualifiers yields the empty set. However, filtering the set of generalized qualifiers yields the predicates we expect:

$\Pi_1 = \{\,\}$
$\Pi_2 = \{\, Ord\ v_1, Show\ v_1, Enum\ v_1 \,\}$

Now again the result of the generalization is the type scheme: $\forall\ a.(Ord\ a, Show\ a, Enum\ a) \Rightarrow a \rightarrow a \rightarrow String$. As we have seen the generalized qualifiers are stored into the set of generalized qualifiers (Π_{gen}) because they can show up in multiple type schemes.

4.4 Conclusion

If we only look at the implementation, the most beautiful aspect of Helium and Top is the elegant way of using constraints for type inference and overloading. The use of constraints enables a nice separation of concerns: Helium generates the constraints, Top solves the constraints, and Helium uses the result of the solver. It is also possible to use the constraint solver stand-alone to experiment with it. Also, Helium can be asked to print the constraints of a program.

Not only the types should be inferred and checked for resolving overloading, but also code or evidence should be generated. Evidence is not part of the solve result of Top, while Top has all the information to generate evidence. Only the types are in the result and Helium must reconstruct context reduction to generate evidence. This is the first point that could be improved.

A second point for improvement is that inside Top and Helium an environment with instance and class declarations is used to perform context reduction. Helium and Top could be further decoupled by using Constraint Handling Rules (CHRs) to represent this information. CHRs are a much clearer specification of the meaning of class and instance declarations. Furthermore, CHRs can be easily extended to support various type class extensions.

A third remark is about the generalization constraint. A generalization constraint in Top now only covers one type τ to generalize. But a binding group may consist of more than one type that should be generalized. Helium solves this by generating multiple generalization constraints in sequence. However, for solving such a sequence Top must store a set of generalized qualifiers because they could be needed by subsequent generalizations. We could get rid of the set of generalized qualifiers (Π_{gen}) by adapting the generalization constraint to generalize a set of types.

Chapter 5

Overloading in EHC

In this chapter we explain how overloading is implemented in the Essential Haskell Compiler (EHC) [Dijkstra, 2005, Dijkstra and Swierstra, 2005]. First, we introduce the language EH and its type system. Second, we explain how the resolution of overloading is implemented. Finally, we conclude with a discussion of the strong and weak points of EHC.

5.1 The language EH

EHC is designed to experiment with advanced features such as higher ranked types, existential types, type classes, partial type signatures, and records. On the other hand, the compiler is also an educational platform for teaching students how a compiler is implemented. Therefore, the compiler consists of ten ordered versions, each adding new features on top of its preceding version. Also, syntactic sugar is kept to a minimum. In Figure 5.1 we present the concrete syntax of EH associated with version 9, which resembles desugared Haskell 98 [Peyton Jones, 2003]. In the next section we present the type system and the syntax of σ and π types. In this section we explain three major differences with respect to Haskell 98.

5.1.1 Instance declarations

EH offers two additional forms of instance declarations besides the one already known from Haskell 98:

- A *named instance* is not used by the compiler for the automatic resolution of overloading, but the dictionary is bound to the identifier i. This identifier can then be used for the *value introduced instance*, but also as an explicitly passed dictionary to a function, of which we show examples hereafter.
- A *value introduced instance* adds a dictionary to the collection of instances that are used by the compiler for the automatic resolution of overloading.

There is a third form (**instance** $i <: \bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$) which is syntactic sugar for the named instance combined with the value introduced instance:

instance $i :: \bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$
instance $i <: (\bar{\pi} \Rightarrow \pi)$

First the dictionary is bound to i. Then the dictionary is added to the instances that are used for the automatic resolution of overloading. In other words, the named introduced instance means the same as the instance declaration we know from Haskell 98, but the resulting dictionary is also bound to an identifier.

5.1.2 Explicit implicit application and abstraction

A Haskell 98 compiler automatically inserts additional dictionary parameters as a result of overloading resolution and the programmer cannot influence this. However, in EHC the programmer can overrule the automatic resolution by passing a dictionary explicitly. In the following example we introduce a type class for equality and an two instances for *Eq Int*:

Value expressions:

e	$::=$	$int \mid char$	(literals)
	$\mid$	i	(variable)
	$\mid$	**let** $\bar{d}$ **in** e	(local definitions)
	$\mid$	$e\ e$	(application)
	$\mid$	$\lambda i \rightarrow e$	(abstraction)
	$\mid$	$e\ \{!e <: \pi!\}$	*(explicit implicit application)*
	$\mid$	$\lambda\ \{!i <: \pi!\} \rightarrow e$	*(explicit implicit abstraction)*

Declarations of bindings:

d	$::=$	$i :: \sigma$	(value type signature)
	$\mid$	$i = e$	(value binding)
	$\mid$	**data** $\overline{\sigma} = \overline{I\overline{\sigma}}$	(datatype)
	$\mid$	**class** $\bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$	(class)
	$\mid$	**instance** $\bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$	(introduced instance)
	$\mid$	**instance** $i :: \bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$	*(named instance)*
	$\mid$	**instance** $e <: \pi$	*(value introduced instance)*
	$\mid$	**instance** $i <: \bar{\pi} \Rightarrow \pi$ **where** $\bar{d}$	*(named introduced instance)*

Figure 5.1: Syntax of EH version 9

```
let class Eq a where
    eq :: a → a → Bool
  instance Eq Int where
    eq = primEqInt
  instance eqDeg :: Eq Int where
    eq = λx y → eq (mod x 360) (mod y 360)
in  (eq 90 450
    , eq {!eqDeg <: Eq Int!} 90 450)
```

The first instance for Eq Int is standard, the second is not. Checking two angles for equality would not work with the default instance so we declare a named instance for equality modulo 360. The instance for equality on degrees does not participate in the automatic resolution of overloading because it is only a named instance. However, the programmer can explicitly pass the *Eq Int* dictionary for degrees to the function *eq*. Evaluating the above expression yields (*False*, *True*).

5.1.3 Local instances

In EH it is also possible to declare instances locally. Named instances can be used as normal values, and participate in the usual scoping rules. However, for instance declarations participating in the automatic resolution a different strategy must be used. For example, when instances overlap, the innermost instance, as defined by lexical scoping, takes precedence over the outermost. Two instances overlap when they are for the same class and if there is a substitution (S) which unifies the heads of the instances.

$(C\ \overline{\sigma})$ *overlaps* $(D\ \overline{\sigma}')$ **if**
$C \equiv D \wedge S\ \overline{\sigma} \equiv S\ \overline{\sigma}'$

where
$S = \overline{bind}$
$bind ::= v\ \mapsto\ \sigma$

The following EHC program illustrates local instances:

```
let class Eq a where
    eq :: a → a → Bool
  instance eqD <: Eq Int where
    eq = primEqInt
```

Types:

σ	::=	$Int \mid Char$	(literals)
	$\mid$	v	(type variable)
	$\mid$	T	(type constant)
	$\mid$	$\sigma\ \sigma$	(type application)
	$\mid$	$\forall\ \bar{v}.\sigma$	(universally quantified type)
	$\mid$	$\exists\ \bar{v}.\sigma$	(existentially quantified type)
	$\mid$	$\bar{\pi} \Rightarrow \sigma$	(implicit abstraction)

Predicates:

π	::=	$C\ \overline{\sigma}$	(predicate)
	$\mid$	$\pi \Rightarrow \pi$	(predicate transformer/abstraction)
	$\mid$	ϖ	(predicate wildcard variable)

Figure 5.2: Type language of EH version 9

$$
\begin{array}{l}
\quad test1 = eq\ 90\ 450 \\
\quad test2 = \textbf{let instance}\ eqDeg <: Eq\ Int\ \textbf{where} \\
\qquad\qquad eq = \lambda x\ y \rightarrow primEqInt\ (mod\ x\ 360)\ (mod\ y\ 360) \\
\qquad\quad \textbf{in}\ \ eq\ 90\ 450 \\
\textbf{in}\ \ (test1, test2)
\end{array}
$$

The two instances in the above program overlap because they unify under the empty substitution. In the body of the function *test1* the instance named *eqD* will be used because it is the only valid instance in scope. However, the instance named *eqDeg* will be used in the body of the function *test2* because it is the innermost valid instance. Note that class declarations are global in EHC just like in Haskell.

Finally, note that all bindings in a let expression are analyzed together, so a let expression in EHC is equal to a binding group in Haskell.

5.2 The type system

The type system of EH implements features like higher ranked types, existential types, type classes, partial type signatures, and records. The type language is presented in Figure 5.2. Three features of this type system deserve special attention.

5.2.1 Higher ranked types

The type system allows higher ranked types. The rank of a type is the depth at which universal quantifiers appear on argument position (contra-variantly). For instance, a rank-0 type is a monomorphic type without quantifiers. A rank-1 type is only quantified on the outermost level, for instance, $\forall\ a.a \rightarrow a$. A rank-2 type is a function with a polymorphic argument, for instance:

$$
\begin{array}{l}
hr :: (\forall\ a.a \rightarrow a) \rightarrow (Char, Int) \\
hr = \lambda id \rightarrow (id\ \texttt{'c'}, id\ 1)
\end{array}
$$

EHC allows also ranks higher than 2. In other word, EHC supports higher ranked types. Note that EHC does not infer higher ranked types, but checks these types by propagating type annotations. Type inference for ranks higher than 2 is even undecidable.

5.2.2 Quantifiers and predicates everywhere

Quantifiers and predicates may occur at the right-hand side of the function type constructor ($\rightarrow$). EHC places the quantifiers and predicates as close as possible to the place where the quantified type variables occur. This

differs from Haskell 98 which allows quantifiers only at the leftmost position in a type (implicitly). To illustrate this, consider the following function:

$$\begin{aligned} &index = \lambda xs\ n \rightarrow \textbf{if}\ n \equiv 0 \\ &\qquad\qquad\quad \textbf{then}\ head\ xs \\ &\qquad\qquad\quad \textbf{else}\ \ index\ (tail\ xs)\ (n-1) \end{aligned}$$

A Haskell 98 compiler infers the following type for this function (ignoring the monomorphism restriction):

$$\forall\ a\ b.Num\ a \Rightarrow [b] \rightarrow a \rightarrow b$$

On the other hand EHC infers:

$$\forall\ b.[b] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow b$$

5.2.3 Higher order predicates

Packaged as an instance declaration, higher order predicates are already available; for example the instance declaration for lists actually takes an instance for its arguments:

instance $Eq\ a \Rightarrow Eq\ [a]$ **where** ...

This instance declaration is translated into a function that creates a dictionary for $Eq\ [a]$ from a dictionary for $Eq\ a$. Such a function is called a dictionary transformer. EHC allows not only predicates in the context, but also predicate transformers:

$$\begin{aligned} &g :: (\forall\ a.Eq\ a \Rightarrow Eq\ [a]) \Rightarrow Int \rightarrow [Int] \rightarrow Bool \\ &g = \lambda p\ q \rightarrow [p] \equiv q \end{aligned}$$

The predicate transformer argument is used to resolve overloading in the above function.

5.3 Implementation of overloading

The richer type language of EH unfortunately implies a more complex implementation. In particular, allowing predicates to occur anywhere in a type means that EH has to anticipate for dictionaries to pass and be passed at arbitrary argument positions. This is illustrated by the following example:

$$\begin{aligned} &index :: \forall\ b.[b] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow b \\ &index = \lambda xs\ n \rightarrow \textbf{if}\ n \equiv 0 \\ &\qquad\qquad\quad \textbf{then}\ head\ xs \\ &\qquad\qquad\quad \textbf{else}\ \ index\ (tail\ xs)\ (n-1) \\ &main = index\ \varpi_1^{\cdot}\ \texttt{"class"}\ \varpi_2^{\cdot}\ 4\ \varpi_3^{\cdot} \end{aligned}$$

There are three possible positions where dictionaries could be inserted at the place where the function *index* is used. Compare this with Haskell 98 which only allows dictionaries before the first parameter of an identifier (position ϖ_1). In EH predicate positions correspond directly to argument passing of dictionaries, a type therefore describes the parameter passing order. EHC instantiates the type of the function *index* to $[Char] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow Char$. No dictionaries have to be inserted at position 1 because there is no predicate before $[Char]$ in the type. Furthermore, EHC instantiates $\forall\ a.Num\ a \Rightarrow a \rightarrow Char$ when the second parameter is applied to the function. The type variable a is instantiated to *Int* and the predicate *Num Int* has to be proven. For that reason, the dictionary corresponding to the predicate *Num Int* has to be inserted at position ϖ_2 in the abstract-syntax tree. No dictionary has to be inserted at position ϖ_3.

5.3.1 Predicate substitution

The above features of EH prevent us from using the standard solution for resolving overloading, because predicates may occur at any position in a type. Instead, resolving of overloading is implemented using a substitution which on top of mapping type variables to types, also describes where which predicates have to be inserted:

$$
\begin{array}{rcl}
bind ::= v & \mapsto & \sigma \\
\mid \varpi & \mapsto & (\pi, \varpi) \\
\mid \varpi & \mapsto & \emptyset
\end{array}
$$

This substitution not only maps type variables (v) to types (σ), but also maps predicate variables (ϖ) to predicates (π). For example, consider the use of *index* again:

$$main = index\ \varpi_1^{\cdot}\ \texttt{"class"}\ \varpi_2^{\cdot}\ 4\ \varpi_3^{\cdot}$$

Predicate variables are distributed over the abstract-syntax tree during the type inference process. The information that becomes available during type inferencing is represented in the substitution. For instance, the solution for the above program is the following substitution:

$$S = \{\varpi_1 \mapsto \emptyset, \varpi_2 \mapsto (Num\ Int, \varpi_4), \varpi_3 \mapsto \emptyset, \varpi_4 \mapsto \emptyset\}$$

Positions in the abstract-syntax tree where no predicates have to be proven are mapped to the empty set.

5.3.2 Type matching

The type of an EH expression is inferred and checked by performing a subsumption check at each node of the abstract-syntax tree. The subsumption check matches the expected type (σ^k) against the actual type (σ) of an expression:

$$\sigma \leqslant \sigma^k$$

The substitution (S) is one of the results of this function.

Consider again the following fragment:

$$main = index\ \texttt{"class"}\ 4$$

We present the steps that are performed by the subsumption check to type the use of the function *index* below. At the left hand side of the subsumption check ($\leqslant$) we see the type of the function *index*. The right hand side shows the type constructed from analyzing the abstract-syntax tree and the fact that a predicate may occur anywhere. In the rightmost column we show which constraints are added to the substitution S.

$$
\begin{array}{rrll}
\forall\ b.[b] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow b & \leqslant \varpi_1 \Rightarrow [Char] \rightarrow \varpi_2 \Rightarrow Int \rightarrow \varpi_3 \Rightarrow v_6 & & \\
\quad [v_7] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow v_7 & \leqslant \varpi_1 \Rightarrow [Char] \rightarrow \varpi_2 \Rightarrow Int \rightarrow \varpi_3 \Rightarrow v_6 & \varpi_1 & \mapsto \emptyset \\
\quad [v_7] \rightarrow \forall\ a.Num\ a \Rightarrow a \rightarrow v_7 & \leqslant [Char] \rightarrow \varpi_2 \Rightarrow Int \rightarrow \varpi_3 \Rightarrow v_6 & v_7 & \mapsto Char \\
\quad\quad \forall\ a.Num\ a \Rightarrow a \rightarrow Char & \leqslant \varpi_2 \Rightarrow Int \rightarrow \varpi_3 \Rightarrow v_6 & & \\
\quad\quad\quad Num\ v_8 \Rightarrow v_8 \rightarrow Char & \leqslant \varpi_2 \Rightarrow Int \rightarrow \varpi_3 \Rightarrow v_6 & \varpi_2 & \mapsto Num\ v_8 \\
\quad\quad\quad\quad v_8 \rightarrow Char & \leqslant Int \rightarrow \varpi_3 \Rightarrow v_6 & v_8 & \mapsto Int \\
\quad\quad\quad\quad\quad Char & \leqslant \varpi_3 \Rightarrow v_6 & \varpi_3 & \mapsto \emptyset \\
\quad\quad\quad\quad\quad Char & \leqslant v_6 & v_6 & \mapsto Char
\end{array}
$$

The following steps are taken to check the use of the function *index*:

- The function *index* is instantiated with a fresh type variable (v_7).
- No predicate is present in the left hand side which could bind to ϖ_1, so ϖ_1 binds to the empty set of predicates.
- The type variable v_7 is bound to Char.
- The type of function *index* is again instantiated with a fresh type variable (v_8).
- The predicate variable ϖ_2 is bound to the predicate $Num\ v_8$.

The subsumption check proceeds until the type is checked or an error is found. The subsumption check also result in a list of predicates that must be proven in order to resolve overloading. The context must provide a proof for the predicate *Num Int* because this predicate occurs at the left hand side of the above example. The proof can not yet be given, so it is delayed until later, in particular when dealing with the enclosing let.

5.3.3 Context-reduction rules

EHC generates a rule for each class and instance declaration. These rules are used to perform context reduction and to construct the code that has to be inserted. A rule consists of the following components:

$$Rule\ (\sigma, \vartheta, name, i, c)$$

A rule has a type (σ), code for constructing evidence (ϑ), a name, a unique identifier, and a number indicating the cost of applying the rule. The unique identifier (i) is a tuple consisting of a number identifying the scope in which the class or instance declaration occurs and a number identifying the rule itself. The cost (c) consist of the level of the rule and the cost of constructing a dictionary with this rule. For a class declaration of the form: **class** $(\pi_1, ..., \pi_n) \Rightarrow \pi$ the following rules are generated given a context identifier ($contextId$):

$$
\begin{array}{l}
i \in (1 \ldots n): \\
\quad u \ \mathit{fresh} \\
\quad \mathit{Rule}\ (\pi \rightarrow \pi_i, \vartheta, \_, (\mathit{contextId}, u), 1)
\end{array}
$$

The first component of the rule $\pi \rightarrow \pi_i$ is the type of the coercion function that selects a superclass from the subclass dictionary. The second component is the coercion function itself. The scope where this class declaration is introduced is identified with $contextId$. The rule is further uniquely identified with the identifier u. The last component indicates that the cost of applying this rule is 1.

For an instance declaration of the form: **instance** $(\pi_1, ..., \pi_n) \Rightarrow \pi$ the following rule is generated given a dictionary name (nm) and a context identifier ($contextId$):

$$
\begin{array}{l}
u \ \mathit{fresh} \\
\mathit{Rule}\ (\pi_1 \rightarrow ... \rightarrow \pi_n \rightarrow \pi, \vartheta, nm, (\mathit{contextId}, u), 2 * n)
\end{array}
$$

For an instance declaration only one rule is generated. The first component of the rule is the type of the function for constructing the dictionary. The second component is the code for applying the construction function itself. The third component is the name of the construction function. The instance rule is identified with $contextId$ and a unique identifier. Finally, the cost of applying the rules is the number of predicates in the context of the instance declaration times 2.

5.4 Conclusion

A nice aspect of EHC is that it implements an advanced type system in combination with very experimental type-class extensions. Also the encoding of class and instance declarations in rules is an elegant abstraction.

The resolution of overloading with such an advanced type system is complex and not yet sufficiently documented in the literature [Dijkstra, 2005, Dijkstra and Swierstra, 2005]. It is a challenge to precisely describe how overloading can be resolved with the type system of EHC.

EHC allows qualifiers and predicates at the right hand side of the function type ($\rightarrow$). Consequently, dictionaries have to be inserted not only after an instantiated variable, but at almost every position in the abstract-syntax tree. The question is if this feature justifies the complexities it introduces. There are two important reasons to support this feature. First, this feature allows us to write shorter programs:

$$
\begin{array}{l}
f :: (\forall\ a.Eq\ a \Rightarrow a \rightarrow a) \rightarrow \ ... \\
g :: \forall\ b.b \rightarrow \forall\ a.Eq\ a \Rightarrow a \rightarrow a \\
x = f\ (g\ 2)
\end{array}
$$

The following program expresses the same fragment in a language that does not allow qualifiers and predicates at arbitrary positions in a type:

$$
\begin{array}{l}
g :: \forall\ a\ b.Eq\ a \Rightarrow b \rightarrow a \rightarrow a \\
x = \mathbf{let}\ g' = g\ 2\ \mathbf{in}\ f\ g'
\end{array}
$$

In this fragment we must first bind the application ($g\ 2$) in a let to trigger generalization. Second, the combination of this feature with existential types has not yet been fully explored, but it is expected that it offers a flexible mechanism for constructing records with higher ranked functions taking predicates, an essential ingredient for module systems.

The encoding of class and instance declarations in rules is elegant. However, the code where predicates are simplified and resolved using these rules is very complex and proved to be difficult to understand. Deep knowledge about the resolving of overloading is encoded in this algorithm and the challenge is to use a solver without specific knowledge of type classes.

Finally, the syntax for explicit implicit application and explicit explicit abstraction is a bit verbose [Dijkstra and Swierstra, 2005]. Besides specifying the value of the predicate also the type has to be given:$\{!dictEqInt <: Eq\ Int!\}$. It would be nice if we can do the same without this additional predicate annotation.

Chapter 6

A framework for overloading resolution

In this chapter we present the initial version of a framework for the resolution of overloading. This version resolves overloading and reports unresolved predicates as an error. In the next chapters we extend this version to support code generation, local instances, and other type class extensions.

6.1 Introduction

Multiple compilers should be able to use our framework to resolve overloading. This means that we cannot depend on how a particular compiler uses this framework. We accomplish this by introducing two abstractions:

First, we formulate the resolution of overloading as a constraint problem. Constraint programming yields several advantages [Aiken, 1999]: The most important advantage is separation of concerns; that is, the specification is separated from the implementation. Generation of constraints is the specification of the analysis; solving of constraints the implementation. The constraints are generated by a compiler and this framework computes a solution for those constraints. Furthermore, class and instance declarations are translated into Constraint Handling Rules (CHRs) and the framework uses these CHRs to solve the constraints.

Second, we do not make any assumption about the structure of the predicate language. Instead, we require that a number of functions are implemented on the predicates. This is accomplished by using type classes as the abstraction mechanism in our implementation.

In this chapter we formulate the overloading problem in terms of a constraint language. Then we show how CHRs can be used to solve overloading constraints and present a translation from class and instance declarations into CHRs. Finally, we describe the implementation of this framework and show the interaction between CHR solving and generalization.

6.2 Constraints for overloading

In this section we explain how the problem of resolving overloading can be formulated into a constraint language. Our approach is based on the formulation in Top [Heeren, 2005]. We use two different types of constraints: First, the $(Assume\ \pi)$ constraint which means that we have an assumption for the qualifier π. Second, the $(Prove\ \pi)$ constraint which means that we have a proof obligation for the qualifier π. For the moment we do not give a precise specification of a qualifier, but leave it abstract. However, in the examples we use type class qualifiers to explain the meaning of the constraints.

$$\begin{aligned} \mathcal{C} ::= &\ Prove \quad \pi \\ \mid &\ Assume\ \pi \end{aligned}$$

Overloading is resolved if all *Prove* constraints are entailed by *Assume* constraints. Consider for example the equality function $(\equiv)$ of type $\forall\ a.Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$. If we use this function, then its type is instantiated with a fresh type variable, say v_1. We have to proof that equality is defined on the type v_1 because the type

class qualifier $Eq\ a$ is part of the type scheme. For that reason, we generate the constraint $Prove\ (Eq\ v_1)$ after instantiating the type. Later in the type inference process we may infer that v_1 is actually of type Int. If we assume that the standard instance for equality on integers is defined, then we are able to solve $Prove\ (Eq\ Int)$ because $\{\,\} \Vdash_e \{\,Eq\ Int\,\}$. In this chapter we use the entailment relation ($\Vdash_e$) introduced in Chapter 2 (figures 2.3 and 2.4).

It is not always possible to directly fulfill proof obligations. Consider for example the following function:

$$\begin{array}{l} max\ x\ y \\ \quad \mid x \geqslant y \quad\ = x \\ \quad \mid otherwise = y \end{array}$$

The overloaded operator $\geqslant$ is used in the function max. This operator has the type $\forall\ a.Ord\ a \Rightarrow a \to a \to Bool$ and is instantiated with the fresh type variable v_2. We do not acquire more type information for the type variable v_2, so we end up with the constraint $Prove\ (Ord\ v_2)$. The inferred type for the function max is $v_2 \to v_2 \to v_2$. After generalizing this type we get the type scheme: $\forall\ a.Ord\ a \Rightarrow a \to a \to a$. The type class qualifier $Ord\ a$ is part of the generalized type because there was a remaining $Prove$ constraint concerning the type variable v_2. Because the type class qualifier $Ord\ v_2$ is generalized, we add the constraint $Assume\ (Ord\ v_2)$ and then $\{\,Ord\ v_2\,\} \Vdash_e \{\,Ord\ v_2\,\}$ trivially holds.

We also generate $Assume$ constraints for each qualifier in the context of an explicitly typed function. Consider for example the following function:

$$\begin{array}{l} elem :: Ord\ a \Rightarrow a \to [a] \to Bool \\ elem\ x\ [\,] \quad\ = False \\ elem\ x\ (y : ys) = x \equiv y \vee elem\ x\ ys \end{array}$$

The type signature of the function $elem$ is skolemized using a fresh type constant, for example c_1. Skolemization of the type signature results in the constraint $Assume\ (Ord\ c_1)$. The use of the two overloaded functions ($\equiv$) and $elem$ result in the constraints $Prove\ (Eq\ v_1)$ and $Prove\ (Ord\ v_2)$, respectively. Later in the type inference process we infer that the type variables v_1 and v_2 are equal to c_1. We then have to solve the following set of constraints: $\{\,Assume\ (Ord\ c_1), Prove\ (Eq\ c_1), Prove\ (Ord\ c_1)\,\}$. Overloading in the function $elem$ is resolved because the qualifier in the $Assume$ constraint entails the qualifiers in the $Prove$ constraints: $\{\,Ord\ c_1\,\} \Vdash_e \{\,Eq\ c_1, Ord\ c_1\,\}$.

Definition 6.1 (Constraint Satisfaction). Satisfaction of $Prove$ and $Assume$ constraints is defined as follows. Consider a solution Θ which consists of a set of assumed qualifiers Π_Θ. A $Prove$ constraints is satisfied if the corresponding qualifier is entailed by the set of assumed qualifiers Π_Θ.

$$\begin{array}{l} \Theta \vdash_s Prove\ \ \pi =_{def} \Pi_\Theta \Vdash_e \Theta(\pi) \\ \Theta \vdash_s Assume\ \pi =_{def} \Theta(\pi) \in \Pi_\Theta \end{array}$$

Furthermore, an $Assume$ constraint is solved if the assumed qualifier is an element of Π_Θ.

6.3 Resolving overloading with CHRs

Until now we have only given a declarative specification of entailment. In this section we investigate whether it is possible to formulate the entailment check with Constraint Handling Rules (CHRs).

6.3.1 Constraint handling rules

CHRs [Frühwirth, 1998] are a high-level declarative language extension especially designed for writing constraint solvers. CHRs are often embedded in a host language: constraints are defined in CHRs, but auxiliary computations are executed in the host language. CHRs operate on a set of constraints and rewrite constraints into simpler ones until they are solved. We use two types of CHRs with the following syntax:

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

The constraints $(H_1, ..., H_i)$ are the *head* of a CHR, the conditions $(G_1, ..., G_j)$ are the *guard* of a CHR, and the constraints $(B_1, ..., B_k)$ are the *body* of a CHR. The empty constraint set is abbreviated with *true*.

Operationally, a simplification rule replaces the set of constraints in the head by the constraints in the body when the conditions in the guard are satisfied. A propagation rule adds the constraints in the body if the constraints in the head are present and the conditions in the guards are satisfied. A propagation rule can be applied infinitely many times, but non-termination is avoided by applying a propagation rule only once to every constraint in the constraint set. CHRs can be given a declarative semantics [Frühwirth, 1998]. A simplification is a logical equivalence if the guards are satisfied:

$$\forall\, \bar{x}\, \forall\, \bar{y}\, ((G_1 \wedge ... \wedge G_j) \rightarrow (H_1 \wedge ... \wedge H_i \leftrightarrow \exists\, \bar{z}\, (B_1 \wedge ... \wedge B_k))$$

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

$$\forall\, \bar{x}\, \forall\, \bar{y}\, ((G_1 \wedge ... \wedge G_j) \rightarrow (H_1 \wedge ... \wedge H_i \rightarrow \exists\, \bar{z}\, (B_1 \wedge ... \wedge B_k)))$$

The sequences of variables $\bar{x}$ and $\bar{y}$ are scoped over the whole CHR. The variables $\bar{x}$ occur in the head of a CHR and the variable $\bar{y}$ in the guard. On the other hand, the variables $\bar{z}$ are only scoped over the body of a CHR.

6.3.2 Formulating entailment with CHRs

The entailment relation ($P \Vdash_e Q$) allows us to check whether it is possible to deduce the predicates in Q from those in P. We have to perform two steps to formulate this relation into CHRs: The first step is called simplification where we try to find the minimum set of predicates P given Q where $P \Vdash_e Q$. This step is also called context reduction. The second step is only performed if there are *Assume* constraints, and checks whether the set of predicates P from the first step are entailed by the predicates in the *Assume* constraints. Simplification is applied to *Prove* constraints and the entailment check matches *Prove* constraints against *Assume* constraints.

Simplification

Simplification itself consists of three steps: removal of duplicate constraints, simplification using instance declarations, and simplification using the class hierarchy. We get removal of duplicate constraints for free, because the CHR solver operates on a set of constraints.

To simplify *Prove* constraints we generate CHRs for each class and instance declaration. In this section we present a number of example translations. In the next section we present the systematic translation. Consider the following class declarations:

```
class Eq a                    -- (C1)
class Eq a  ⇒ Ord a           -- (C2)
class Ord a ⇒ Real a          -- (C3)
instance Eq a ⇒ Eq [a]        -- (I1)
```

The following CHRs are generated for these declarations (note that a is an implicitly universally quantified variable in each CHR):

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

If we apply the rule above on the constraint set $\{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\}$ we could get the following derivation:

$$\begin{array}{ll} & \{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\ \longrightarrow_{I1} & \{Prove\ (Eq\ v_1)\ , Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\ \longrightarrow_{C2} & \{Prove\ (Ord\ v_1)\ , Prove\ (Real\ v_1)\} \\ \longrightarrow_{C3} & \{Prove\ (Real\ v_1)\} \end{array}$$

First, the rule *I1* is applied, then *C2*, and finally *C3*, but the following derivation is also possible:

$$\begin{array}{ll} & \{Prove\ (Eq\ [v_1]), Prove\ (Ord\ v_1), Prove\ (Real\ v_1)\} \\ \longrightarrow_{C3} & \{Prove\ (Eq\ [v_1]), Prove\ (Real\ v_1)\} \\ \longrightarrow_{I1} & \{Prove\ (Eq\ v_1)\ , Prove\ (Real\ v_1)\} \end{array}$$

This shows that the rules are not confluent. A set of rules is *confluent* if from any given state every possible order of rule applications ends in the same final state. By adding the following rule we make our rules confluent:

$$Prove\ (Eq\ a), Prove\ (Real\ a) \Longleftrightarrow Prove\ (Real\ a)$$

This means that we have to add a rule for every superclass of a class, and not only for the direct superclasses. In other words, we have to explicitly add the transitive closure of the superclass relation to the CHRs used for simplification.

Entailment

Until now we have only presented the rules to simplify *Prove* constraints. The second step is to check whether the predicates in the *Prove* constraints match with the predicates in the *Assume* constraints. Therefore we introduce the following CHR:

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

This rule means that the proof obligation p can be removed if we have an assumption for p. Note that this rule abstracts over the predicate language used. Recall the function *elem*:

$$\begin{aligned} &elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\ &elem\ x\ [\,] \quad\ = False \\ &elem\ x\ (y : ys) = x \equiv y \vee elem\ x\ ys \end{aligned}$$

If the explicit type signature of *elem* is skolemized with the fresh type constant c_1 we eventually have to solve the set of constraints: $\{Prove\ (Eq\ c_1), Assume\ (Eq\ c_1)\}$. This set can easily be solved by applying rule E. But giving the following signature for *elem* is also correct:

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

The current rules are not sufficient to solve this problem. We not only have to simplify proof obligations, but also propagate assumptions. If there is an assumption for *Ord a* then also all superclasses of *Ord a* can be assumed. Therefore we introduce a propagation CHR for each class declaration:

$$\begin{aligned} &\textbf{class}\ Eq\ a \ \Rightarrow Ord\ a && \text{-- (C2)} \\ &\textbf{class}\ Ord\ a \Rightarrow Real\ a && \text{-- (C3)} \\ &Assume\ (Ord\ a) \Longrightarrow Assume\ (Eq\ a) && \text{-- (CA2)} \\ &Assume\ (Real\ a) \Longrightarrow Assume\ (Ord\ a) && \text{-- (CA3)} \end{aligned}$$

Now we are able to solve the following set of constraints:

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

The predicates in the *Assume* constraints entail those in the *Prove* constraints if there are no *Prove* constraints left after solving.

We have shown how we can formulate the entailment check with CHRs. In the next section we present the first implementation of the framework together with functions for generating CHRs from class and instance declarations.

6.3.3 A Haskell CHR solver

In order to solve constraints with CHRs we have implemented a basic CHR solver in Haskell. This solver is used by the framework, but could also be replaced by another CHR solver. We do not explain the implementation of this solver, but only present the interface. We represent a CHR with the following datatype:

$$\begin{aligned} &\textbf{data}\ CHR\ c\ s = CHR\{\ head\ \ :: [c] \\ &\qquad\qquad\qquad\quad ,\ guard :: (s \rightarrow Maybe\ s) \\ &\qquad\qquad\qquad\quad ,\ body\ \ :: [c] \\ &\qquad\qquad\qquad\quad \} \end{aligned}$$

The datatype CHR is parametrized with two type variables: c represents the type of the constraint, s represents the type of the substitution. A successful match of the head of a CHR with actual constraints results in a substitution. Variables in the head of the CHR are bound to values in this substitution. The guard is a function which gets this substitution as a parameter. The guard returns a new substitution if the conditions in the guards are satisfied, otherwise the guard returns *Nothing*. This substitution is applied to the body of the CHR after the substitution resulting from the head is applied.

We introduce two operators to construct simplification and propagation CHRs, respectively:

```
infix 1 ⟺, ⟹
(⟺), (⟹) :: Monoid s ⇒ [p] → [p] → CHR p s
hs ⟺ bs = CHR hs emptyGuard bs
hs ⟹ bs = hs ⟺ (hs ++ bs)

emptyGuard :: Monoid s ⇒ t → Maybe s
emptyGuard = const (Just mempty)

infixr 0 ▷
(▷) :: CHR p s → (s → Maybe s) → CHR p s
(▷) (CHR hs _ bs) g = CHR hs g bs
```

We represent both propagation and simplification rules with the CHR datatype. A propagation rule is immediately translated into a simplification rule when constructing it with the operator $(\Longrightarrow)$. This is achieved by appending the head of the propagation rule to the body of the simplification rule. These operators construct CHRs with a guard that always returns the empty substitution. Note that the identifier *mempty* is from the *Monoid* class. A guard can be attached to a CHR with the operator $(\triangleright)$.

The following type class is used to support matching of constraints:

```
class (Ord c, Monoid s) ⇒ Matchable c s | c → s where
  match :: c → c → Maybe s
  subst  :: s → c → c

  match x y | x ≡ y      = Just mempty
            | otherwise  = Nothing
  subst = flip const
```

The default implementation of this class is matching on syntactic equality. An instance of this class must be given for the type of the constraints used. Also more advanced forms of matching with patterns are possible. The function *match* is not symmetric, in other words, there are two values: x, y where $match\ x\ y \not\equiv match\ y\ x$. The function *match* expects a constraint as left hand side and a constraint of the head of a CHR as right hand side. The head constraint is matched against the constraint and when the match is successful a substitution is returned. The following QuickCheck [Claessen and Hughes, 2000] property should hold:

```
propMatch c h =
  case match c h of
    Just s   → property (c ≡ subst s h)
    Nothing → property ()
```

A constraint c should be equal to the resulting substitution applied to the head if the constraint matches with the head. The asymmetry is also the reason why there is no need to instantiate a CHR first with fresh variables as with instantiating type schemes. The substitution is only applied to the CHR and not on the constraints, therefore the variables of a CHR will never leak to the constraints.

Finally, we introduce the function that solves a set of constraints given a list of CHRs:

```
chrSolve :: Matchable c s ⇒ [CHR c s] → Set c → Set c
```

For convenience, we also introduce a function that solves a list of constraints using CHRs. Internally, the list is converted into a set and visa versa:

```
chrSolveList :: Matchable c s ⇒ [CHR c s] → [c] → [c]
chrSolveList chrs cs = Set.toList (chrSolve chrs (Set.fromList cs))
```

6.4 Implementation

We have described a constraint language for specifying the overloading problem. Furthermore, we have shown how CHRs can be used to check entailment. This section describes the implementation of those ideas in our first version of the framework. We explain how we represent the constraint language in Haskell. Then we present a systematic translation from class and instance declarations into CHRs. Finally, we show how constraints are solved using CHRs.

This section is also a literate Haskell module:

```
import CHRSolver
import Control.Monad.State
import qualified Data.Set as Set
import Data.List (intersect)
```

Besides some standard Haskell libraries we import the CHR solver presented in the previous section.

6.4.1 Constraint language

A constraint is either a proof obligation or an assumption:

```
data Constraint p = Prove p | Assume p
                  deriving (Eq, Ord)
instance Functor Constraint where
  fmap f (Prove  p) = Prove  (f p)
  fmap f (Assume p) = Assume (f p)
```

The type variable p represents the type of the predicate. We also make the type *Constraint* an instance of *Functor* so we can easily map functions over a *Constraint*. Furthermore, we make *Constraint* an instance of *Matchable*:

```
instance Matchable p s ⇒ Matchable (Constraint p) s where
  match (Prove p)   (Prove q)   = match p q
  match (Assume p) (Assume q) = match p q
  match _           _           = Nothing
  subst s (Prove p)             = Prove  (subst s p)
  subst s (Assume p)            = Assume (subst s p)
```

Prove and *Assume* constraints only match when the predicates in these constraints match. So we require that matching is defined on the type of the predicate language.

6.4.2 Translation to CHRs

For the resolution of overloading we have to translate class and instance declarations into CHRs. Class and instance declaration are translated into:

1. CHRs for propagating the class hierarchy and removing entailed constraints.
2. CHRs for simplifying constraints using the class hierarchy.
3. CHRs for simplifying constraints using instance declarations.

We introduce some type synonyms before presenting the systematic translation:

```
type Rule p s = CHR (Constraint p) s
```

The type synonym *Rule* is a CHR on the earlier defined constraint language. We also introduce type synonyms for instance and class declarations:

```
type ClassDecl    p = ([p], p)
type InstanceDecl p = ([p], p)
```

Both are tuples where the first component is the context and the second component is the head of the declaration.

Step 1: CHRs for propagating the class hierarchy and removing entailed constraints.

We start with the translation from a class declaration into CHRs:

$$
\begin{array}{l}
genAssumeChrs :: Matchable\ a\ s \Rightarrow ClassDecl\ a \rightarrow [Rule\ a\ s] \\
genAssumeChrs\ (context, head) = \\
\quad \textbf{let}\ solve \quad\ = [Assume\ head, Prove\ head] \Longleftrightarrow [Assume\ head] \\
\qquad\ propSuper = [Assume\ head] \Longrightarrow map\ Assume\ context \\
\quad \textbf{in}\ \ [solve, propSuper]
\end{array}
$$

For each class declaration we generate a simplification CHR. This simplification removes a *Prove* constraint if there is an assumption in the set for the same predicate. We do not remove the assumption, because it can be used to solve another *Prove* constraint. Furthermore, a propagation CHR is generated that propagates assumptions for the context of a class declaration if there is an assumption for the head. The simplification rule is generated for each class declaration, but in fact we could use the general rule: $Prove\ p, Assume\ p \Longleftrightarrow Assume\ p$. Adding such a rule would introduce an additional substitution from variables to predicates and because we do not need this rule in the remainder of this thesis we omit it for simplicity. The function application $genAssumeChrs\ ([Eq\ a, Show\ a], Num\ a)$ results from translating the class declaration **class** $(Eq\ a, Show\ a) \Rightarrow Num\ a$. This function application evaluates to the following list of CHRs:

$$
\begin{array}{l}
[\,Assume\ (Num\ a), Prove\ (Num\ a) \Longleftrightarrow Assume\ (Num\ a) \\
\,, Assume\ (Num\ a) \Longrightarrow Assume\ (Eq\ a), Assume\ (Show\ a)]
\end{array}
$$

Step 2: CHRs for simplifying constraints using the class hierarchy.

The propagation rules generated for the class declarations encode the class hierarchy. In this step we use the rules of step 1 to generate CHRs for the simplification of *Prove* constraints using the class hierarchy.

$$
\begin{array}{l}
genClassChrs :: Matchable\ a\ s \Rightarrow [\,ClassDecl\ a] \rightarrow [Rule\ a\ s] \\
genClassChrs\ clsDecls = \\
\quad \textbf{let}\ assumeChrs = concatMap\ genAssumeChrs\ clsDecls \\
\qquad\ simplChrs \quad = concatMap\ (genClassSimplChrs\ assumeChrs)\ clsDecls \\
\quad \textbf{in}\ \ assumeChrs +\!\!+ simplChrs \\
genClassSimplChrs :: Matchable\ a\ s \Rightarrow [Rule\ a\ s] \rightarrow ClassDecl\ a \rightarrow [Rule\ a\ s] \\
genClassSimplChrs\ rules\ (context, head) = \\
\quad \textbf{let}\ superClasses = chrSolveList\ rules\ (map\ Assume\ context) \\
\qquad\ simpl\ (Assume\ sClass) = [Prove\ head, Prove\ sClass] \Longleftrightarrow [Prove\ head] \\
\quad \textbf{in}\ \ \textbf{if}\ elem\ (Assume\ head)\ superClasses \\
\qquad\ \textbf{then}\ error\ (\texttt{"Cyclic class hierarchy"}) \\
\qquad\ \textbf{else}\ \ map\ simpl\ superClasses
\end{array}
$$

We perform the following steps for each class declaration: First we translate the predicates in the context into *Assume* constraints. Then the assume constraints are solved using the rules for the propagation of the class hierarchy. Solving those constraints result in a constraint set consisting of all superclasses of the predicate *head*. The class hierarchy is cyclic if the head of the class declaration is an element of the superclass set. If this is not the case, a simplification CHR is generated for each class in this set. In this way we obtain a confluent rule set while checking if the class hierarchy is acyclic. For example, consider the following application of the function *genClassSimplChrs* for the class: **class** $(Ord\ a, Num\ a) \Rightarrow Real\ a$.

$$
\begin{array}{l}
genClassSimplChrs \\
\quad [\,Assume\ (Real\ \ a) \Longrightarrow Assume\ (Num\ a), Assume\ (Ord\ \ a) \\
\quad \,, Assume\ (Num\ a) \Longrightarrow Assume\ (Eq\ \ a), Assume\ (Show\ a) \\
\quad \,, Assume\ (Ord\ \ a) \Longrightarrow Assume\ (Eq\ \ a)] \\
\quad ([\,Ord\ a, Num\ a], Real\ a)
\end{array}
$$

Evaluating this expression results in:

$$
\begin{array}{l}
[\,Prove\ (Real\ a), Prove\ (Ord\ \ a) \Longleftrightarrow Prove\ (Real\ a) \\
\,, Prove\ (Real\ a), Prove\ (Num\ \ a) \Longleftrightarrow Prove\ (Real\ a) \\
\,, Prove\ (Real\ a), Prove\ (Eq\ \ a) \Longleftrightarrow Prove\ (Real\ a) \\
\,, Prove\ (Real\ a), Prove\ (Show\ a) \Longleftrightarrow Prove\ (Real\ a)]
\end{array}
$$

Step 3: CHRs for simplifying constraints using instance declarations.

To simplify *Prove* constraints using instances we generate the following CHR for each instance declaration:

```
genInstanceChrs :: Matchable a s ⇒ [InstanceDecl a] → [Rule a s]
genInstanceChrs =
  let genSimpl (context, head) = [Prove head] ⟺ map Prove context
  in  map genSimpl
```

For each instance we generate a rule where the head of the instance is simplified into the context of the instance. Finally, we concatenate the rules for class and instance declarations:

```
genChrs :: Matchable a s ⇒ [ClassDecl a] → [InstanceDecl a] → [Rule a s]
genChrs classes insts =
  let classChrs = genClassChrs classes
      instChrs  = genInstanceChrs insts
  in  classChrs ++ instChrs
```

6.4.3 Constraint solving

The solver maintains a state when solving constraints. This state consists of two components:

```
data SolveState p s =
  SolveSt{ constraints :: ConstrSet p
         , rules       :: [Rule p s]
         }
type ConstrSet p = Set.Set (Constraint p)
```

The first component of the state is a set of both *Prove* and *Assume* constraints. The second component is a list of CHRs for the constraint language. The initial state of the solver consists of an empty constraint set and a list of CHRs generated for the class and instance declarations in the program.

Solving a constraint is straightforward: it consists of inserting the constraint into the constraint set:

```
solveConstraint :: (MonadState (SolveState p s) m, Matchable p s) ⇒ Constraint p → m ()
solveConstraint c =
  modifyConstrSet (Set.insert c)
modifyConstrSet f =
  modify (λs → s{ constraints = f (constraints s)})
```

Both *Assume* and *Prove* constraints are inserted into the constraint set. Recall the definition of constraint satisfaction:

$$\Theta \vdash_s \mathit{Prove}\ \ \pi \ =_{def}\ \Pi_\Theta \Vdash_e \Theta(\pi)$$
$$\Theta \vdash_s \mathit{Assume}\ \pi \ =_{def}\ \Theta(\pi) \in \Pi_\Theta$$

As we have seen in this chapter *Prove* constraints result from instantiating type schemes and *Assume* constraints result from skolemizing type schemes of explicitly typed functions. We immediately solve an *Assume* constraint by inserting it into the constraint set. But a *Prove* constraint is only solved if it is entailed by the *Assume* constraints.

The following function solves *Prove* constraints by removing them from the constraint set if they are entailed by the *Assume* constraints:

```
simplify :: (MonadState (SolveState p s) m, Matchable p s) ⇒ m ()
simplify =
  do rls ← gets rules
     modifyConstrSet (chrSolve rls)
```

This function can be applied at any stage in the type-inference process. Overloading is resolved if there are only *Assume* constraints left in the constraint set after the last simplification. *Prove* constraints that eventually remain are reported to the user as an error.

The description of the framework would be complete if every function in a Haskell program would be explicitly typed because then all *Assume* constraints also are explicitly given. However, for functions without a type

signature we have to infer a minimal set of *Assume* constraints that entail the *Prove* constraints of the function body. Predicates should be generalized if the predicate and the type that must be generalized have free type variables in common. Therefore, we introduce a type class for substitutables:

```
class Eq v ⇒ Substitutable a v s | a → v, a → s where
  ftv        :: a → [v]
  substitute :: s → a → a
instance Substitutable p v s ⇒ Substitutable (Constraint p) v s where
  ftv (Prove p)    = ftv p
  ftv (Assume p) = ftv p
  substitute s (Prove p)    = Prove    (substitute s p)
  substitute s (Assume p) = Assume (substitute s p)
```

The overloaded function *ftv* computes a list of type variables of type v given a value of type a and the function *substitute* applies a substitution s. The following function returns the predicates that should be generalized given a list of type variables that are going to be generalized.

```
getProveObligations :: (MonadState (SolveState p s) m, Substitutable p v s) ⇒ [v] → m [p]
getProveObligations tps =
  do prvs ← gets constraints
     let g = not.null.intersect tps.ftv
         isProve (Prove _) = True
         isProve _         = False
         unProve (Prove p) = p
     return [unProve p | p ← (Set.toList prvs), isProve p, g p]
```

The predicates are selected with the function g. This function returns true if the intersection of the free type variables and *tps* is not empty.

We present the steps needed to generalize a type to illustrate the interaction between a compiler and the framework during this step. We assume the compiler using this framework has a substitution S and an environment Σ. Generalizing a type consists of the following steps given type τ and a set of type variables that should remain monomorphic $\mathcal{M}$.

- The substitution S is applied to τ, $\mathcal{M}$, and the constraint set. This ensures that the type information acquired during the analysis of the binding group is applied.
- The constraints are simplified by invoking the function *simplify*. Simplification ensures that the constraints are reduced to head normal form so that the minimal set of *Prove* constraints needed to resolve overloading is constructed.
- The type variables that are going to be generalized are computed by removing the variables in $\mathcal{M}$ from the free type variables in τ.
- The function *getProveObligations* is used to select the predicates that should be part of the generalized type. This function gets the set of type variables computed in the previous step as argument.
- The type τ is generalized together with the selected predicates and stored into the environment Σ.
- Finally, the generalized predicates are added as assumptions to the constraint set. This makes the inferred context available as assumptions, so the prove constraints can be solved.

6.5 Conclusion

In this chapter we have shown that is possible to formulate the resolution of overloading as a constraint problem using *Prove* and *Assume* constraints. Furthermore, we have shown how those constraints can be simplified using Constraint Handling Rules (CHRs). We not only presented CHRs to simplify *Prove* constraints, but also presented CHRs that remove *Prove* constraints when entailed by *Assume* constraints. We have presented a systematic translation from class and instance declarations into CHRs and during this translation we also check if the class hierarchy is acyclic. On top of that, we have also implemented a domain specific language for CHRs together with a basic CHR solver in Haskell. The most important observation is that we have presented a framework for resolving overloading without having concrete knowledge about the structure of a predicate. This ensures that multiple compilers can use this framework.

Chapter 7

Evidence translation

In this chapter we extend the framework with evidence translation. This means that we also construct evidence associated with the solution of proof obligations.

7.1 Introduction

The translation of a language with overloading into a language without is called *evidence translation*. There are two well-known translation schemes described in the literature. The first one is the so-called dictionary translation scheme [Wadler and Blott, 1989] where overloaded functions are translated into functions with additional dictionary parameters. The second translation scheme is based on partial evaluation [Jones, 1995a] where overloaded functions are translated into several specialized versions. This framework aims at supporting the first translation scheme.

The CHRs presented in the previous chapter only check whether *Prove* constraints are entailed by *Assume* constraints. However, to generate evidence we need to keep track of *how* constraints are solved. Furthermore, we explain how non-confluency is avoided by generating all correct reduction alternatives for solving a constraint. In the next section we give motivation for these changes and modify the translation from class and instance declarations into CHRs. Solving constraints with these new CHRs results in a set of all possible reductions. In the section 7.4 we explain how these reductions and other information needed to generate evidence are represented in a graph. Finally, we show how heuristics can be used to choose between different reduction alternatives in the graph. As an illustration, we show how overlapping instances can be supported using the framework.

7.2 Translation to CHRs

In this section we adapt the translation from class and instance declarations into CHRs. We first show how the derivation steps performed by the CHR solver can be traced. Then we explain why every correct alternative for simplifying a constraint must be generated and how this is achieved. We do not give a new systematic translation because the main idea of the translation to CHRs stays the same.

7.2.1 Tracing CHR derivations

In order to generate evidence, we have to consider the derivation steps needed to arrive at a solution. For example, consider the following function for testing whether a list is still ordered after an insertion:

$$
\begin{aligned}
&\mathit{testInsert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool}\\
&\mathit{testInsert}\ x\ xs = \mathbf{let}\ ys = \mathit{insert}\ x\ (\mathit{sort}\ xs)\\
&\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{sort}\ ys \equiv ys
\end{aligned}
$$

When type checking this function, the explicit type signature is skolemized with some type constant (c_1). After analyzing the body of the function we discover that we have to solve the following constraint set:

$$\{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Prove}\ (\mathit{Ord}\ c_1), \mathit{Prove}\ (\mathit{Eq}\ [c_1])\}$$

The following derivation is possible if we assume that the standard class and instance declarations for *Eq* and *Ord* are available:

$$\begin{array}{ll} & \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Prove}\ (\mathit{Ord}\ c_1),\ \ \mathit{Prove}\ (\mathit{Eq}\ [c_1])\} \\ \longrightarrow & \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Assume}\ (\mathit{Eq}\ c_1), \mathit{Prove}\ (\mathit{Ord}\ c_1), \mathit{Prove}\ (\mathit{Eq}\ [c_1])\} \\ \longrightarrow & \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Assume}\ (\mathit{Eq}\ c_1), \mathit{Prove}\ (\mathit{Eq}\ [c_1])\} \\ \longrightarrow & \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Assume}\ (\mathit{Eq}\ c_1), \mathit{Prove}\ (\mathit{Eq}\ c_1)\} \\ \longrightarrow & \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Assume}\ (\mathit{Eq}\ c_1)\} \end{array}$$

Overloading is resolved, because there are no *Prove* constraints left in the set of constraints. However, to generate code we need to know how we arrived at the final set of constraints. For example, consider the translated version of the function *testInsert*:

$$\begin{array}{l} \mathit{testInsert} :: \mathit{DictOrd}\ a \to a \to [a] \to \mathit{Bool} \\ \mathit{testInsert}\ d\ x\ xs = \mathbf{let}\ ys = \mathit{insert}\ d\ x\ (\mathit{sort}\ d\ xs) \\ \qquad\qquad\qquad\qquad\ \mathbf{in}\ \ ((\equiv)\ (\mathit{eqList}\ (\mathit{eqOrd}\ d)))\ (\mathit{sort}\ d\ ys)\ ys \end{array}$$

We need to know which intermediate steps are needed to deduce $\mathit{Eq}\ [c_1]$ from $\mathit{Ord}\ c_1$ to generate the code in the above function for using the equality $(\equiv)$ operator.

To achieve this, we let each CHR generate an additional *Reduction* constraint. This constraint is used to record a derivation step. Let us first extend the constraint language with a *Reduction* constraint:

$$\begin{array}{lll} \mathcal{C} ::= & \mathit{Prove} & \pi \\ \mid & \mathit{Assume} & \pi \\ \mid & \mathit{Reduction}\ \pi\ \mathit{info}\ [\pi_1, ..., \pi_n] & \end{array}$$

A *Reduction* constraint consists of three components: a predicate that is reduced, an annotation, and the predicates that are the result of the reduction. A *Reduction* constraint means that it is possible to reduce the first component to the predicates that form the third component. At the same time it means that the first component can be constructed using the evidence associated with the predicates that form the third component. The third component of a reduction constraint is a list, because a predicate can be reduced to zero or more predicates and we want to remember the predicate order. For example, the predicate *Eq Bool* is reduced to the empty list, but the predicate $\mathit{Eq}\ (v_1, v_2)$ reduced to $[\mathit{Eq}\ v_1, \mathit{Eq}\ v_2]$. The reduction constraints can be annotated with information. This information consists, for example of identifiers for generating code, information for generating type-error messages, or information which is used by heuristics to make a choice.

Consider the class and instance declarations below:

$$\begin{array}{l} \mathbf{class}\ \mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a \\ \mathbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] \end{array}$$

For these declarations we generate the following CHRs:

$$\begin{array}{llll} \mathit{Assume}\ (\mathit{Ord}\ a) & \Longrightarrow \mathit{Assume}\ (\mathit{Eq}\ a),\ \ \mathit{Reduction}\ (\mathit{Eq}\ a) & \texttt{"eqOrd"} & [\mathit{Ord}\ a] \\ \mathit{Prove}\ \ (\mathit{Ord}\ a), \mathit{Prove}\ (\mathit{Eq}\ a) & \Longleftrightarrow \mathit{Prove}\ \ (\mathit{Ord}\ a), \mathit{Reduction}\ (\mathit{Eq}\ a) & \texttt{"eqOrd"} & [\mathit{Ord}\ a] \\ \mathit{Prove}\ \ (\mathit{Eq}\ [a]) & \Longleftrightarrow \mathit{Prove}\ \ (\mathit{Eq}\ a),\ \ \mathit{Reduction}\ (\mathit{Eq}\ [a]) & \texttt{"eqList"} & [\mathit{Eq}\ \ a] \end{array}$$

The reduction constraints in the rules are annotated with the identifier of the dictionary transformer that performs the reverse step required for evidence construction. Applying these rules on the set of constraint we considered earlier will result in the following constraint set:

$$\begin{array}{l} \{\mathit{Assume}\ (\mathit{Ord}\ c_1), \mathit{Assume}\ (\mathit{Eq}\ c_1) \\ , \mathit{Reduction}\ (\mathit{Eq}\ c_1) \quad \texttt{"eqOrd"} \quad [\mathit{Ord}\ c_1] \\ , \mathit{Reduction}\ (\mathit{Eq}\ [c_1])\ \texttt{"eqList"}\ [\mathit{Eq}\ \ c_1]\ \} \end{array}$$

This constraint set contains two reduction constraints which give information how $\mathit{Eq}\ [c_1]$ can be deduced from $\mathit{Ord}\ c_1$. This information can be used to generate evidence. We do not have enough information yet to perform the complete translation: however, we can already generate code for constructing dictionaries.

7.2.2 Derivation alternatives and confluence

A problem occurs when we simplify constraints with CHRs: constraint sets exist where different derivations lead to the same answer. For example, consider the constraint set $\{\mathit{Prove}\ (\mathit{Eq}\ [v_1]), \mathit{Prove}\ (\mathit{Ord}\ [v_1])\}$. Two

derivations are possible: the first derivation simplifies the constraints by first using instances and then using the class hierarchy:

$$
\begin{aligned}
& \{\mathit{Prove}\ (\mathit{Eq}\ [v_1]), \mathit{Prove}\ (\mathit{Ord}\ [v_1])\} \\
\longrightarrow\ & \{\mathit{Prove}\ (\mathit{Eq}\ v_1),\quad \mathit{Prove}\ (\mathit{Ord}\ [v_1]), \mathit{Reduction}\ (\mathit{Eq}\ [v_1])\ \texttt{"eqList"}\ [\mathit{Eq}\ v_1]\} \\
\longrightarrow\ & \{\mathit{Prove}\ (\mathit{Eq}\ v_1),\quad \mathit{Prove}\ (\mathit{Ord}\ v_1),\quad \mathit{Reduction}\ (\mathit{Eq}\ [v_1])\ \texttt{"eqList"}\ [\mathit{Eq}\ v_1] \\
& \quad , \mathit{Reduction}\ (\mathit{Ord}\ [v_1])\ \texttt{"ordList"}\ [\mathit{Ord}\ v_1]\} \\
\longrightarrow\ & \{\mathit{Prove}\ (\mathit{Ord}\ v_1), \mathit{Reduction}\ (\mathit{Eq}\ [v_1])\ \texttt{"eqList"}\ [\mathit{Eq}\ v_1] \\
& \quad , \mathit{Reduction}\ (\mathit{Ord}\ [v_1])\ \texttt{"ordList"}\ [\mathit{Ord}\ v_1], \mathit{Reduction}\ (\mathit{Eq}\ v_1)\ \texttt{"eqOrd"}\ [\mathit{Ord}\ v_1]\}
\end{aligned}
$$

The second derivation simplifies the constraints by reversing the preference for instances and classes; first the class hierarchy is used, then the available instances:

$$
\begin{aligned}
& \{\mathit{Prove}\ (\mathit{Eq}\ [v_1]),\quad \mathit{Prove}\ (\mathit{Ord}\ [v_1])\} \\
\longrightarrow\ & \{\mathit{Prove}\ (\mathit{Ord}\ [v_1]), \mathit{Reduction}\ (\mathit{Eq}\ [v_1])\ \texttt{"eqOrd"}\ [\mathit{Ord}\ [v_1]]\} \\
\longrightarrow\ & \{\mathit{Prove}\ (\mathit{Ord}\ v_1),\quad \mathit{Reduction}\ (\mathit{Eq}\ [v_1])\ \texttt{"eqOrd"}\ [\mathit{Ord}\ [v_1]] \\
& \quad , \mathit{Reduction}\ (\mathit{Ord}\ [v_1])\ \texttt{"ordList"}\ [\mathit{Ord}\ v_1]\}
\end{aligned}
$$

Both derivations are correct and lead to the same answer. However, it is not desirable that different derivations are chosen for the same constraint set in a non-deterministic way. The problem becomes even more evident with extensions such as overlapping instances. Consider the following overlapping instances example:

$$
\begin{aligned}
& \textbf{data}\ \mathit{Unit} = \mathit{Unit} \\
& \textbf{instance}\ \mathit{Eq}\ [\mathit{Unit}]\ \textbf{where} \\
& \quad \mathit{xs} \equiv \mathit{ys} = \mathit{length}\ \mathit{xs} \equiv \mathit{length}\ \mathit{ys} \\
& \textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]\ \textbf{where}\ \ldots
\end{aligned}
$$

The CHRs generated for these instance declarations are:

$$
\begin{aligned}
& \mathit{Prove}\ (\mathit{Eq}\ [a]) \quad \Longleftrightarrow \mathit{Prove}\ (\mathit{Eq}\ a) \\
& \mathit{Prove}\ (\mathit{Eq}\ [\mathit{Unit}]) \Longleftrightarrow \mathit{true}
\end{aligned}
$$

Two non-confluent derivations are possible when we try to solve the constraint $\mathit{Prove}\ (\mathit{Eq}\ [\mathit{Unit}])$. One derivation results in the empty constraint set and the other derivation results in $\{\mathit{Prove}\ (\mathit{Eq}\ \mathit{Unit})\}$. The most specific instance is preferred if there are multiple possibilities when using overlapping instances, but a CHR solver chooses one of the two derivations in a non-deterministic way. A solution proposed for this problem is to add guards to the simplification rules [Stuckey and Sulzmann, 2002]:

$$
\begin{aligned}
& \mathit{Prove}\ (\mathit{Eq}\ [a]) \quad \Longleftrightarrow a \not\equiv \mathit{Unit} \mid \mathit{Prove}\ (\mathit{Eq}\ a) \\
& \mathit{Prove}\ (\mathit{Eq}\ [\mathit{Unit}]) \Longleftrightarrow \mathit{true}
\end{aligned}
$$

This results in a confluent rule set. However, the design decision that the most specific instance is preferred is not really clear from the rules because the guard is added to the CHR generated for the 'normal' instance and not for the overlapping instance. Furthermore, guards can become very complex especially when using overlapping instances in combination with other type-class extensions, such as multi-parameter type classes or local instances.

To solve this problem we propose another solution: We use CHRs to generate every possible type-correct derivation. This means that we do not make a choice, but just generate all the possibilities. With the reductions generated from this process we construct a graph and a heuristic chooses a solution from this graph. With this approach we separate the process of finding possible solutions from the process of selecting the preferred solution.

The systematic translation from class and instance declarations into CHRs has to be adapted to generate every correct alternative. We never let CHRs remove constraints from the constraint set to achieve confluent derivations and to avoid encoding extensions in guards of CHRs. Therefore, every applicable CHR will be applied to a constraint. This means that we cannot use simplification CHRs anymore because they replace constraints by other constraints and thereby remove constraints from the constraint set. We have to modify the systematic translation with respect to the following points:

- The rule $\mathit{Assume}\ p, \mathit{Prove}\ p \Longleftrightarrow \mathit{Assume}\ p$ is not used anymore, because it removes constraints from the constraint set.
- Other simplification CHRs are replaced by propagation CHRs.
- We let each CHR generate reduction constraints to trace the reduction step performed.

Because these changes are relatively small we do not present the systematic translation again. Instead, we give some examples of the adapted translation. Consider the class declaration for numbers:

class $(Eq\ a, Show\ a) \Rightarrow Num\ a$

The following CHRs are generated for this class declaration:

$$\begin{aligned}
&Assume\ (Num\ a) \Longrightarrow Assume\ (Eq\ \ \ a), Reduction\ (Eq\ \ \ a)\ \texttt{"eqNum"}\ \ [Num\ a]\\
&\qquad\qquad\qquad\quad , \quad Assume\ (Show\ a), Reduction\ (Show\ a)\ \texttt{"showNum"}\ [Num\ a]\\
&Prove\ (Eq\ \ \ a), Prove\ (Num\ a) \Longrightarrow Reduction\ (Eq\ \ \ a)\ \texttt{"eqNum"}\ \ [Num\ a]\\
&Prove\ (Show\ a), Prove\ (Num\ a) \Longrightarrow Reduction\ (Show\ a)\ \texttt{"showNum"}\ [Num\ a]
\end{aligned}$$

The first rule adds the superclasses of *Num* to the constraint set and adds the corresponding reduction constraints. The second and the third rule add a reduction from superclass to subclass if there are proof obligations for both classes. Again consider the overlapping instance declarations:

data $Unit = Unit$

instance $Eq\ [Unit]$ **where**
  $xs \equiv ys = length\ xs \equiv length\ ys$

instance $Eq\ a \Rightarrow Eq\ [a]$ **where** ...

For these instance declarations we generate the following rules:

$$\begin{aligned}
&Prove\ (Eq\ [a]) \quad\ \Longrightarrow Prove\ (Eq\ a), Reduction\ (Eq\ [a])\ \texttt{"eqList"}\ [Eq\ a]\\
&Prove\ (Eq\ [Unit]) \Longrightarrow Reduction\ (Eq\ [Unit])\ \texttt{"eqListUnit"}\ []
\end{aligned}$$

The new rules for the overlapping instances are confluent, because the rules do not remove constraints from the set anymore. We obtain the following result if we apply these rules on the constraint set $\{Prove\ (Eq\ [Unit])\}$.

$$\begin{aligned}
&\{Prove\ (Eq\ [Unit])\\
&, Reduction\ (Eq\ [Unit])\ \texttt{"eqList"}\ [Eq\ Unit]\\
&, Prove\ (Eq\ Unit)\\
&, Reduction\ (Eq\ [Unit])\ \texttt{"eqListUnit"}\ []\}
\end{aligned}$$

The result of solving the constraint does not immediately tell us if overloading is resolved. Moreover, there are two possible and correct solutions present in the constraint set. That is the reason why we represent the reductions in a graph and select one of the solutions using a heuristics in the next section.

7.3 Simplification graphs

In this section we explain how we represent the information needed to generated evidence in a graph. The reductions and predicates represented in this graph are shared as much as possible. Also different context-reduction alternatives can be expressed in the graph. In this section we first explain the structure of a graph. Then we show how graphs are used in the framework. This section and the following section present a literate Haskell module for the second version of the framework:

import *AGraph*
import *CHRSolver*
import *Control.Monad.State*
import *Data.List* $(nub, maximumBy)$
import *qualified Data.Map as Map* $(Map, empty, insertWith, foldWithKey, keys, assocs)$

Two imported modules deserve special attention. The module *AGraph* is a wrapper module for the functional graph library [Erwig, 2001] and the module *CHRSolver* is the Haskell CHR solver introduced in Section 6.3.

7.3.1 Graph representation

We represent reductions and information needed to generate evidence in a directed graph. A directed graph G is an ordered pair $G = (V, A)$ with

- A set of nodes V, and
- a set of pairs of nodes A, representing directed edges.

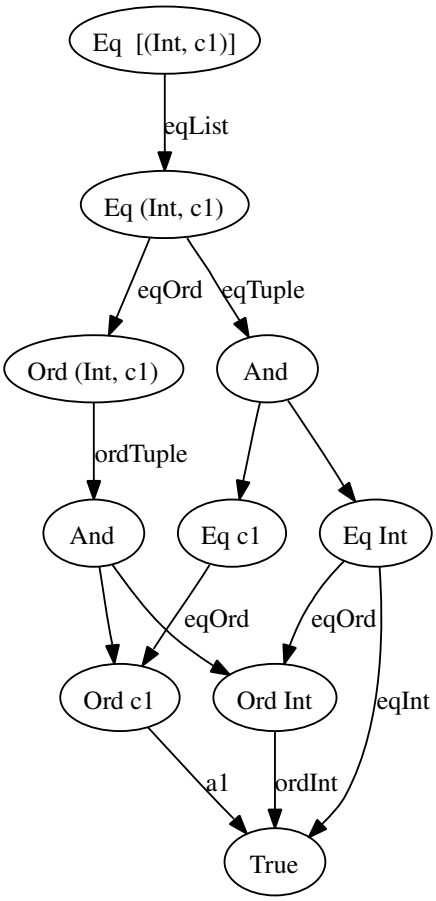

Figure 7.1: Simplification graph for the function *testInsert* with a restricting signature

The idea behind simplification graphs is that nodes represent predicates and edges between nodes represent reductions. Edges are annotated with meta-information, such as unique identifiers and error messages. Again, consider the function *testInsert*, but now with a restricting type signature:

$$
\begin{aligned}
&\mathit{testInsert} :: (\mathit{Ord}\ a)_{a1} \Rightarrow (\mathit{Int}, a) \to [(\mathit{Int}, a)] \to \mathit{Bool} \\
&\mathit{testInsert}\ x\ xs = \mathbf{let}\ ys = \mathit{insert}_{p1}\ x\ (\mathit{sort}_{p2}\ xs) \\
&\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{sort}_{p3}\ ys\ \equiv_{p4}\ ys
\end{aligned}
$$

The type signature of this function is skolemized with a fresh type constant, say c_1. After analyzing the function body we have to solve the constraints *Assume* $(\mathit{Ord}\ c_1)$ occurring at position *a1*, *Prove* $(\mathit{Ord}\ (\mathit{Int}, c_1))$ at positions *p1*, *p2*, *p3*, and *Prove* $(\mathit{Eq}\ [(\mathit{Int}, c_1)])$ at position *p4*. Solving these constraints results in the simplification graph presented in Figure 7.1. There is one special type of node besides the node for predicates:

```
data Node p = Pred p
            | And [p]
          deriving (Eq, Ord)
instance Show p ⇒ Show (Node p) where
  show (Pred p) = show p
  show (And []) = "True"
  show (And _) = "And"
```

Multiple outgoing edges from the *Pred* node represent different reduction alternatives. For example, in Figure 7.1 we see that there are two alternatives for reducing the predicate $\mathit{Eq}\ (\mathit{Int}, c_1)$. As opposed to the *Pred* node, multiple outgoing edges from an *And* node means that each edge is needed to resolve the predicate. For example, the predicate $\mathit{Ord}\ (\mathit{Int}, c_1)$ is reduced to *Ord Int* and $\mathit{Ord}\ c_1$ in Figure 7.1. The list of predicates needed by the *And* constructor seems to be unnecessary, because this information is already available in the graph. However, this list of predicates is needed to remember the order of the predicates we reduce to. For example, we must remember which predicate is needed for which component of a tuple. An *And* node without

outgoing edges means that overloading is resolved at this point. We abbreviate this special case of *And* with *true*.

```
true :: Node p
true = And []
```

We use the inductive graph library [Erwig, 2001] to represent graphs in Haskell. However, in this context it is not very interesting to explain the interface of this library. Therefore, we present the wrapper module (*AGraph*) exporting only the following functions:

```
emptyAGraph :: Ord a ⇒ AGraph a b
insertEdge  :: Ord a ⇒ (a, a, b)  → AGraph a b → AGraph a b
insertEdges :: Ord a ⇒ [(a, a, b)] → AGraph a b → AGraph a b
deleteEdge  :: Ord a ⇒ (a, a)     → AGraph a b → AGraph a b
successors, predecessors :: Ord a ⇒ AGraph a b → a → [(b, a)]
```

AGraph stands for A(nnotated) Graph. Such a graph consists of two types of annotations: annotations on nodes (a) and annotations on edges (b). The function *insertEdge* inserts an edge between nodes and inserts the nodes if they are not already present. The function *deleteEdge* deletes all directed edges from the first node to the second node. With the function *successors* and *predecessors* it is possible to inspect the graph. In this section we use the following type synonym for *AGraph*'s of *Node*'s:

```
type Graph p info = AGraph (Node p) info
```

7.3.2 Representation of constraints

In the previous section we have presented a third constraint beside the *Prove* and *Assume* constraint:

```
data Constraint p info = Prove      p
                       | Assume     p
                       | Reduction p info [p]
                       deriving (Eq, Ord)
```

The *Reduction* constraint is a special constraint because it is only used internally; a user of the framework is not allowed to use it. *Reduction* constraints result from solving *Prove* and *Assume* constraints with CHRs and are used to construct simplification graphs. Furthermore, the *Reduction* constraint is the only constraint with an *info* component. This *info* component can be used to store meta-information, such as unique identifiers and error messages. We also make the constraint language an instance of the type class *Matchable*.

```
instance (Matchable p s, Ord info) ⇒ Matchable (Constraint p info) s where
  match (Prove p) (Prove q)      = match p q
  match (Assume p) (Assume q)    = match p q
  match _ _                      = Nothing
  subst s (Prove p)              = Prove    (subst s p)
  subst s (Assume p)             = Assume (subst s p)
  subst s (Reduction p i ps)     = Reduction (subst s p) i (map (subst s) ps)
```

To generate evidence and to represent information needed to generate evidence we have to extend the state of the solver:

```
data SolveState p s info =
  SolveSt{ rules       :: [CHR (Constraint p info) s]
         , heuristic   :: Heuristic      p info
         , constraints :: Constraints    p info
         , evidence    :: EvidenceMap p info
         }
type Constraints    p info = Map.Map (Constraint p info) [info]
type EvidenceMap p info = Map.Map info (Evidence p info)
modifyConstraints f = modify (λs → s{ constraints = f (constraints s)})
modifyEvidence    f = modify (λs → s{ evidence    = f (evidence    s)})
```

The state is extended with two components: a heuristic and an evidence map. In the next section we explain how the heuristic and the evidence map are used. Furthermore, the constraint set in the state is replaced by a map from constraints to occurrences of constraints. Evidence has to be generated for each occurrence of a constraint. Each occurrence of a constraint is tupled with information uniquely identifying the constraint occurrence. For example, a unique identifier may serve this purpose. Additionally, location information for type error messages could be included as well. Solving a list of constraints just consists of mapping the function *solveConstraint* in the monad:

$$
\begin{aligned}
&\mathit{solveConstraints} :: (\mathit{MonadState}\ (\mathit{SolveState}\ p\ s\ \mathit{info})\ m, \mathit{Matchable}\ p\ s, \mathit{Ord}\ \mathit{info}) \\
&\qquad \Rightarrow [(\mathit{Constraint}\ p\ \mathit{info}, \mathit{info})] \to m\ () \\
&\mathit{solveConstraints} \quad = \mathit{mapM\_}\ \mathit{solveConstraint} \\
&\mathit{solveConstraint}\ (c, i) = \mathit{modifyConstraints}\ (\mathit{Map.insertWith}\ (+\!\!+)\ c\ [i])
\end{aligned}
$$

The function *solveConstraint* expects a constraint tupled with information. A constraint is inserted into the constraint map together with the information packed into a singleton list. Information lists are concatenated when a constraint is already in the map. This means that there can be multiple occurrences of the same constraint. There are a number of advantages of representing constraints annotated with information in a map. For instance, constraints that occur multiple times have to be solved only once. Furthermore, using the constraint map and the simplification graph we may share evidence.

7.3.3 Simplification of constraints

Everything comes together when simplifying constraints. Simplification can be applied at any stage during the type inference process, just as with the previous version of the framework. The difference is that we now only use CHRs to generate *Reduction* constraints. These *Reduction* constraints are represented in a simplification graph and a heuristic solves constraints by choosing a solution. To illustrate simplification, consider the function *testInsert* again:

$$
\begin{aligned}
&\mathit{testInsert} :: (\mathit{Ord}\ a)_{a1} \Rightarrow (\mathit{Int}, a) \to [(\mathit{Int}, a)] \to \mathit{Bool} \\
&\mathit{testInsert}\ x\ xs = \mathbf{let}\ ys = \mathit{insert}_{p1}\ x\ (\mathit{sort}_{p2}\ xs) \\
&\qquad\qquad\qquad\ \mathbf{in}\ \ \mathit{sort}_{p3}\ ys\ \equiv_{p4}\ ys
\end{aligned}
$$

The following list of constraints is generated for the function *testInsert*:

$$
\begin{aligned}
&[(\mathit{Assume}\ (\mathit{Ord}\ c_1) \qquad , a1), (\mathit{Prove}\ (\mathit{Ord}\ (\mathit{Int}, c_1)), p1), (\mathit{Prove}\ (\mathit{Ord}\ (\mathit{Int}, c_1)), p2) \\
&, (\mathit{Prove} \quad (\mathit{Ord}\ (\mathit{Int}, c_1)), p3), (\mathit{Prove}\ (\mathit{Eq}\ [(\mathit{Int}, c_1)]), p4)]
\end{aligned}
$$

Solving the above list of constraints result in the following constraint map:

$$
\{\mathit{Assume}\ (\mathit{Ord}\ c_1)\ \mapsto\ [a1], \mathit{Prove}\ (\mathit{Ord}\ (\mathit{Int}, c_1))\ \mapsto\ [p1, p2, p3], \mathit{Prove}\ (\mathit{Eq}\ [(\mathit{Int}, c_1)])\ \mapsto\ [p4]\}
$$

Simplification of these constraints is performed with the function *simplify*:

$$
\begin{aligned}
&\mathit{simplify} :: (\mathit{MonadState}\ (\mathit{SolveState}\ p\ s\ \mathit{info})\ m, \mathit{Matchable}\ p\ s, \mathit{Ord}\ \mathit{info}) \Rightarrow m\ () \\
&\mathit{simplify} = \\
&\quad \mathbf{do}\ \mathit{chrs} \quad \leftarrow \mathit{gets}\ \mathit{rules} \\
&\qquad \mathit{cnstrs} \leftarrow \mathit{gets}\ \mathit{constraints} \\
&\qquad \mathbf{let}\ \mathit{initGraph} = \mathit{Map.foldWithKey}\ \mathit{addAssumption}\ \mathit{emptyAGraph}\ \mathit{cnstrs} \\
&\qquad\quad\ \ \mathit{reductions} = \mathit{chrSolveList}\ \mathit{chrs}\ (\mathit{Map.keys}\ \mathit{cnstrs}) \\
&\qquad\quad\ \ \mathit{graph} \quad\ = \mathit{foldr}\ \mathit{addReduction}\ \mathit{initGraph}\ \mathit{reductions} \\
&\qquad \mathit{modifyConstraints}\ (\mathit{const}\ \mathit{Map.empty}) \\
&\qquad \mathit{mapM\_}\ (\mathit{constructEvidence}\ \mathit{graph})\ (\mathit{Map.assocs}\ \mathit{cnstrs})
\end{aligned}
$$

First, *Assume* constraints are added to the empty graph by folding the following function over the constraint map:

$$
\begin{aligned}
&\mathit{addAssumption} :: \mathit{Ord}\ p \Rightarrow \mathit{Constraint}\ p\ \mathit{info} \to [\mathit{info}] \to \mathit{Graph}\ p\ \mathit{info} \to \mathit{Graph}\ p\ \mathit{info} \\
&\mathit{addAssumption}\ (\mathit{Assume}\ p)\ \mathit{is} = \mathit{insertEdges}\ (\mathit{zip3}\ (\mathit{repeat}\ (\mathit{Pred}\ p))\ (\mathit{repeat}\ \mathit{true})\ \mathit{is}) \\
&\mathit{addAssumption}\ \_ \qquad\qquad\ \_ = \mathit{id}
\end{aligned}
$$

An *Assume* constraint is represented in the graph as an edge from the assumed predicate to the *true* node. An edge is inserted for each occurrence of an assumption. For example, the assumption for $\mathit{Ord}\ c_1$ annotated with *a1* is present in Figure 7.1. The next step is to generate *Reduction* constraints by solving the constraints

using CHRs. The resulting reductions are inserted into the graph by folding the following function over the set of constraints:

$$
\begin{aligned}
&\mathit{addReduction} :: \mathit{Ord}\ p \Rightarrow \mathit{Constraint}\ p\ \mathit{info} \rightarrow \mathit{Graph}\ p\ \mathit{info} \rightarrow \mathit{Graph}\ p\ \mathit{info}\\
&\mathit{addReduction}\ (\mathit{Reduction}\ p\ i\ [q]) = \mathit{insertEdge}\ (\mathit{Pred}\ p, \mathit{Pred}\ q, i)\\
&\mathit{addReduction}\ (\mathit{Reduction}\ p\ i\ ps) = \mathbf{let}\ \mathit{andNd} = \mathit{And}\ ps\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{edges} = \mathit{map}\ (\lambda q \rightarrow (\mathit{andNd}, \mathit{Pred}\ q, i))\ ps\\
&\qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \mathit{insertEdges}\ ((\mathit{Pred}\ p, \mathit{andNd}, i) : \mathit{edges})\\
&\mathit{addReduction}\ \_ = \mathit{id}
\end{aligned}
$$

The function *addReduction* consist of three cases: The first case is not required, but improves the readability of the graphs by not inserting intermediate *And* nodes. In the second case we add a list of edges: one edge from the predicate to an *And* node and n edges from the *And* node to the predicates we reduce to. Nothing is done for other constraints in the third case. *Prove* and *Assume* constraints propagated by CHRs are discarded. The following reductions are generated when solving the constraints resulting from the function *testInsert*:

$$
\begin{aligned}
&\{\mathit{Reduction}\ (\mathit{Ord}\ (\mathit{Int}, c_1))\ \ \texttt{"ordTuple"}\ [\mathit{Ord}\ \mathit{Int}, \mathit{Ord}\ c_1], \mathit{Reduction}\ (\mathit{Eq}\ \ \mathit{Int})\ \texttt{"eqInt"}\ \ [\,]\\
&, \mathit{Reduction}\ (\mathit{Eq}\ \ [(\mathit{Int}, c_1)])\ \texttt{"eqList"}\ \ [\mathit{Eq}\ (\mathit{Int}, c_1)],\ \ \mathit{Reduction}\ (\mathit{Ord}\ \mathit{Int})\ \texttt{"ordInt"}\ [\,]\\
&, \mathit{Reduction}\ (\mathit{Eq}\ \ (\mathit{Int}, c_1))\ \ \texttt{"eqTuple"}\ \ [\mathit{Eq}\ \mathit{Int}, \mathit{Eq}\ c_1],\ \ \mathit{Reduction}\ (\mathit{Eq}\ \ c_1)\ \ \texttt{"eqOrd"}\ \ [\mathit{Ord}\ c_1]\\
&, \mathit{Reduction}\ (\mathit{Eq}\ \ (\mathit{Int}, c_1))\ \ \texttt{"eqOrd"}\ \ \ [\mathit{Ord}\ (\mathit{Int}, c_1)],\ \ \mathit{Reduction}\ (\mathit{Eq}\ \ \mathit{Int})\ \texttt{"eqOrd"}\ \ [\mathit{Ord}\ \mathit{Int}]\\
&\}
\end{aligned}
$$

Inserting these reductions result in the graph presented in Figure 7.1. The last step of the simplification function is to generated evidence for each entry in the constraint map using the function *constructEvidence*. The function *constructEvidence* determines which constraints are solved and which constraints remain unresolved using the generated simplification graph and heuristics. Therefore we initialize the constraint map to the empty map so that the function *constructEvidence* can add the remaining constraints. In the following section we explain how heuristics are formulated and how the function *constructEvidence* is implemented.

7.4 Heuristics

The simplification graphs presented in the previous chapter contain the information needed to construct evidence. However, in order to generate evidence we have to make choices between different context reduction alternatives. We have chosen to isolate such choices in heuristics. In this section we first explain how we have formulated heuristics. Then we show the implementation of heuristics in the solver. Finally, we present some example heuristics to show how different context-reduction strategies can be emulated and how overlapping instances are resolved.

7.4.1 Formulation of heuristics

The task of a heuristic is to choose between context reduction alternatives. For example, a choice between overlapping instances, scoped instances, or superclass versus instance. However, in some cases a heuristic must also be able to stop context reduction. In such a situation no alternative is chosen, even if there exist alternatives. For example, consider the function *testInsert*, now without a type signature:

$$
\begin{aligned}
&\mathit{testInsert}\ x\ xs = \mathbf{let}\ ys = \mathit{insert}_{p1}\ x\ (\mathit{sort}_{p2}\ xs)\\
&\qquad\qquad\qquad\quad\ \mathbf{in}\ \ \mathit{sort}_{p3}\ ys\ \equiv_{p4}\ \ ys
\end{aligned}
$$

The constraints $\mathit{Prove}\ (\mathit{Ord}\ v_1)$ at positions $p1, p2, p3$ and $\mathit{Prove}\ (\mathit{Eq}\ [v_1])$ at position $p4$ are generated for this function. Simplifying those constraints result in the leftmost graph of Figure 7.2. GHC [Marlow and Peyton Jones, 2006] infers the type $(\mathit{Ord}\ a, \mathit{Eq}\ [a]) \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool}$ for the above function whereas Haskell 98 compilers would infer the type $\mathit{Ord}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Bool}$. GHC stops context reduction at a certain point to maximize the possibilities for using overlapping instances. Furthermore, it is not always the case that the same choices are made for all the occurrences of a predicate. For example, the programmer could annotate an occurrence of a predicate to choose a different context reduction strategy.

Complicated heuristics could be applied to the simplification graph to maximize sharing. However, we choose to generate evidence for each *Prove* constraint separately and thereby not sharing intermediate computations. A heuristic has the following type and is part of the state of the solver:

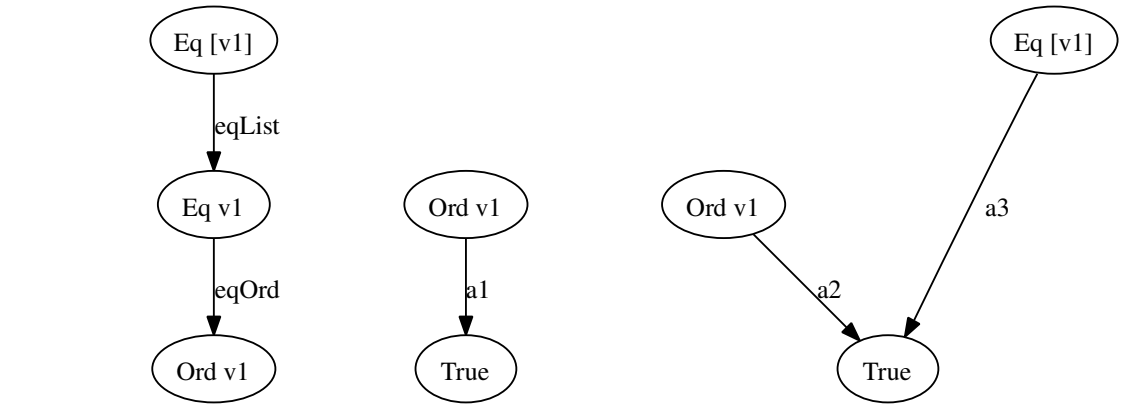

Figure 7.2: Simplification graphs for the function *testInsert*

type *Heuristic p info* = [*info*] → *Alts p info* → [(*info*, *Evidence p info*)]

A heuristic is a function that gets a list of the occurrences of the predicate as first parameter. The second argument of this function is a datatype representing the reduction alternatives for the predicate we must prove:

data *Alts p info* = *Alts*{ *predicate* :: *p*, *alts* :: [*Red p info*] }
data *Red p info* = *Red*{ *info* :: *info*, *context* :: [*Alts p info*] }

The datatype *Alts* consists of a predicate that must be proven and a list of reduction alternatives (*alts*). A heuristic can decide to stop context reduction or to choose one of these reduction alternatives. The datatype *Red* consists of the information annotated on the edges of the graph and the list of predicates we reduced to. A value of this datatype is extracted from a simplification graph in the following way:

```
alternatives :: Ord p ⇒ Graph p info → p → Alts p 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
```

The result of a heuristic is an association list mapping occurrences of predicates to evidence. Evidence has the shape of a tree:

```
data Evidence p info =  Proof p info [Evidence p info]
                     |  Unresolved p
unresolved :: Eq p ⇒ Evidence p info → [p]
unresolved (Unresolved p) = [p]
unresolved (Proof _ _ ps) = nub (concatMap unresolved ps)
```

A *Proof* means that a predicate *p* can be constructed from a list of evidence. Furthermore, a *Proof* is annotated with information. However, it is not always possible to construct evidence at once. Sometimes there are still proof obligations left. The *Unresolved* constructor is used when we are not (yet) able to prove a predicate *p*. The function *unresolved* returns the predicates that are unresolved in an evidence tree. The following function updates unresolved predicates if later in the type-inference process more information about these predicates is acquired:

```
updateUnresolved :: Eq p ⇒ Evidence p info → Evidence p info → Evidence p info
updateUnresolved e                (Unresolved _) = e
updateUnresolved (Proof p i qs)   e              = Proof p i [updateUnresolved q e | q ← qs]
updateUnresolved u@(Unresolved q) e@(Proof p _ _)
  | q ≡ p      = e
  | otherwise  = u
```

The first parameter of this function is evidence that is updated and the second parameter is evidence that is inserted.

7.4.2 Application of heuristics

In the section 7.3 we have presented the simplification function. The last step of this function is to map the function *constructEvidence* over the constraint map. The function *constructEvidence* generates evidence for the different occurrences of the constraint by using a heuristic to choose a solution from the simplification graph:

```
constructEvidence :: (MonadState (SolveState p s info) m, Matchable p s, Ord info)
                     ⇒ Graph p info → (Constraint p info, [info]) → m ()
constructEvidence graph (Prove p, infos) =
  do hrstc ← gets heuristic
     let trees = hrstc infos (alternatives graph p)
     modifyEvidence  (λem → foldr insertEvidence em trees)
     solveConstraints (concatMap remaining trees)
constructEvidence _       (c, infos)        =
     solveConstraints (zip (repeat c) infos)
```

The above function generates evidence for *Proof* constraints. Other constraints are immediately inserted into the constraint map with the function *solveConstraints*. To construct evidence, the heuristic is fetched from the state and bound to an identifier. In the next step, the heuristic is applied to the occurrences and the alternatives for reducing predicate p. The result of the heuristic is an association list from occurrences to evidence. The evidence is inserted in the map by folding the following function over the result of the heuristic:

```
insertEvidence :: (Eq p, Ord info)
                  ⇒ (info, Evidence p info) → EvidenceMap p info → EvidenceMap p info
insertEvidence = uncurry (Map.insertWith updateUnresolved)
remaining :: Eq p ⇒ (info, Evidence p info) → [(Constraint p info, info)]
remaining (i, tree) = zip (map Prove (unresolved tree)) (repeat i)
```

Evidence is updated with the function *updateUnresolved* if there already is an entry for *info* in the map. Finally, *Prove* constraints are generated for the unresolved predicates in the evidence trees with the function *remaining*. Those constraints are solved with the function *solveConstraints* we have discussed earlier in this chapter.

7.4.3 Haskell 98 heuristic

In Haskell 98 [Peyton Jones, 2003] the context of a type must be in head normal form (HNF). This means that the type mentioned in a predicate may only consist of a type variable or the application of a type variable to one or more types. For instance, the predicates in the context of the following type are in HNF:

```
test :: (Monad m, Eq (m Char)) ⇒ m Char → Bool
```

However, the predicate in the following type is not in HNF:

```
test :: Eq (Maybe a) ⇒ Maybe a → Bool
```

Furthermore, different occurrences of the same predicate always have the same solution in Haskell 98. Therefore, we introduce a special type synonym for such heuristics together with a conversion function:

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

A simple heuristic does not depend on information attached to the occurrences of a predicate. Furthermore, a simple heuristic can be converted to a heuristic by just repeating the solution for each occurrence of the predicate. A heuristic must choose at each node between i different choices for each of the j following nodes. In the worst case the heuristic has to choose between i^j choices. We call j the *branching factor*. In general want to keep the branching factor as small as possible. Simple heuristics with the smallest branching factor (1) can be defined with the following function:

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

Besides the fact that the heuristic for Haskell 98 can be expressed as a local heuristic, it also has the property that the different reduction alternatives form a total order. This means that each pair of alternatives can be compared for ordering. Therefore we introduce a function that creates a simple heuristic from an ordering function:

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

We annotate the different constraints with values of the following datatype:

```
data Annotation = ByInstance    String
                | BySuperClass String
                | ProveObl     Int
                | Assumption   Int
                deriving (Eq, Ord)
```

The constructors *ByInstance* and *BySuperclass* are used to annotate the CHRs generated for instance and class declarations respectively. The constructor arguments of type String identify the corresponding dictionaries or dictionary transformers. *Prove* constraints are annotated with *ProveObl* and *Assume* constraints are annotated with *Assumption*. The integer arguments uniquely identify occurrences of *Prove* and *Assume* constraints.

Using these annotations we can define the following heuristic for Haskell 98:

```
haskell98 :: Annotation → Annotation → Ordering
haskell98 (ByInstance    _) _                = GT
haskell98 _                 (ByInstance    _) = LT
haskell98 (BySuperClass _) _                = GT
haskell98 _                 (BySuperClass _) = LT
haskell98 (Assumption    _) _                = GT
haskell98 _                 (Assumption    _) = LT
haskell98 (ProveObl      _) _                = GT
haskell98 _                 (ProveObl      _) = LT

h98Heuristic :: Heuristic p Annotation
h98Heuristic = toHeuristic (binChoice haskell98)
```

Predicates in Haskell 98 compilers are first simplified using instances. This will ensure that the predicates are solved or that they are simplified to predicates in HNF. If possible, the predicates are simplified using the class hierarchy and then using an assumption. Note that Haskell 98 dictates that assumptions must always be in HNF.

Recall the graph generated for the function *testInsert* presented in Figure 7.2. The predicate $Ord\ v_1$ in this graph cannot be simplified further and remains. However, $Eq\ [v_1]$ can be simplified to $Ord\ v_1$ and the Haskell 98 heuristic chooses for this simplification. This will result in the following state of the solver:

```
constraints = {Prove (Ord v1) ↦ [p1, p2, p3, p4]}
evidence    = {p1, p2, p3 ↦ Unresolved (Ord v1)
              , p4 ↦ Proof (Eq [v1]) "eqList" [Proof (Eq v1) "eqOrd" [Unresolved (Ord v1)]]}
```

In a compiler this is usually the point where generalization is performed. The result is that the remaining proof obligations concerning the generalized type are assumed to hold. In this example, the constraint $(Assume\ (Ord\ v_1), a1)$ is added. Simplifying the new set of constraints result in the graph in the middle of Figure 7.2. After constructing more evidence the state of the solver will be:

```
constraints = {Assume (Ord v1) ↦ [a1]}
evidence    = {p1, p2, p3 ↦ Proof (Ord v1) "a1" []
              , p4 ↦ Proof (Eq [v1]) "eqList" [Proof (Eq v1) "eqOrd" [Proof (Ord v1) "a1" []]]}
```

It is easy to see that this result can be used to generate the following translated version of *testInsert*:

```
testInsert a1 x xs = let ys = insert a1 x (sort a1 xs)
                     in ((≡) (eqList (eqOrd a1))) (sort a1 ys) ys
```

7.4.4 GHC heuristic

As we have mentioned earlier, GHC [Marlow and Peyton Jones, 2006] utilizes another context reduction strategy than the one Haskell 98 dictates. This deviation from Haskell 98 is needed to support various extensions in GHC such as overlapping instances and arbitrary contexts in types and type signatures [Peyton Jones et al., 1997]. Haskell 98 compilers reduce the context as far as possible before generalization. In contrast, GHC delays context reduction using instance declarations 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 (BySuperClass _) _                 = GT
ghcBinSolve _                 (BySuperClass _) = LT
ghcBinSolve (ByInstance    _) _                 = GT
ghcBinSolve _                 (ByInstance    _) = LT
ghcBinSolve (ProveObl      _) _                 = GT
ghcBinSolve _                 (ProveObl      _) = LT
ghcSolve :: Eq p ⇒ SimpleHeuristic p Annotation
ghcSolve = binChoice ghcBinSolve
```

The most notable difference between the heuristics of Haskell 98 and GHC is the way they reduce predicates using instance declarations. Haskell 98 reduces predicates first with instance declarations, but GHC reduces predicates only with instance declarations if there are no other alternatives left. The reason that this is possible is that GHC allows arbitrary contexts in type signatures. If GHC would first reduce using instances, it could possibly not find assumptions introduced by type signatures. 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 (BySuperClass _) = 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 is stopped if context reduction using the class hierarchy is not possible 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
```

The *try* combinator applies the first heuristic and only when there are unresolved nodes in the result, the second heuristic is applied. Now we are able to construct a heuristic that emulates the context-reduction behavior of GHC:

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

The *ghcHeuristic* still has a branching factor of one because the two sub heuristics both have a branching factor of one. Two passes are needed for this heuristic to find a solution, but the pass of the second heuristic will never evaluate more reduction alternatives than the first pass. Using this heuristic for resolving overloading in the function *testInsert* results in the rightmost graph of Figure 7.2 after generalization. Using this heuristic results in the following translated version of *testInsert*:

```
testInsert (a2, a3) x xs = let ys = insert a2 x (sort a2 xs)
                           in  ((≡) a3) (sort a2 ys) ys
```

7.4.5 Overlapping instances heuristic

Until now we did not explain how to resolve overlapping instances using this framework. We have chosen to encode overlapping instances in heuristics instead of using guards in CHRs. Here we explain how to extend a local heuristic to support overlapping instances. First, we need to encode more information into the annotation for instances:

```
data Annotation p = ByInstance String p
                  | ...
```

We add the head of an instance declaration to the annotation. The most specific instance is chosen when multiple instances are applicable for simplifying a predicate. Note that the context of an instance declaration is not used in this process. The following function determines what the most specific predicate is:

```
specificness :: Matchable c s ⇒ c → c → Ordering
specificness p q =
  case match p q of
       Nothing → LT
       Just _  → case match q p of
                      Nothing → GT
                      Just _  → error "no most specific instance"
```

This function is defined in terms of the one way unification function *match*. More about this function can be found in Section 6.3. The predicate q is more specific than p if p does not match with q. For example, $Eq\ [[a]]$ is more specific than $Eq\ [a]$ because there is no substitution to get $Eq\ [a]$ from $Eq\ [[a]]$. The most specific instance cannot be determined if the predicate p matches q and the other way around. This is for example the case with the predicates $Eq\ (a, Int), Eq\ (Int, a)$.

Finally, a heuristic can easily be extended to support overlapping instances. For example, by adding the following case to the Haskell 98 heuristic:

```
haskell98 :: Annotation p → Annotation p → Ordering
haskell98 (ByInstance _ p) (ByInstance _ q) = specificness p q
haskell98 (ByInstance _ _) _                = GT
haskell98 _                (ByInstance _ _) = LT
haskell98 ...                               = ...
```

7.5 Conclusion

In this chapter we have shown how our framework supports the translation of a program with overloading into a program without overloading. Furthermore, we have presented different heuristics to show how easy it is to experiment with different design decisions. To achieve this we trace CHR derivations using reduction constraints. However, CHR derivations can sometimes be non-deterministic or even non-confluent. We avoid these problems by generating every type-correct reduction alternative for solving a constraint. From these reduction constraints we construct simplification graphs. These graphs enable us to share predicates as much as possible and to represent different reduction alternatives. Another advantage of these graphs is that they nicely visualize the problem. We use heuristics to choose a solution from a simplification graph for each *Prove* constraint. With these heuristic we are able to emulate Haskell 98 and GHC context reduction. We also show that heuristics can easily be extended to support type class extensions such as overlapping instances.

Chapter 8

Local instances

This chapter describes how the framework can be used to resolve overloading in the context of local instances [Dijkstra, 2005]. Local instances can be used, for example, to encode dynamically scoped variables [Lewis et al., 2000]. This extension also allows the programmer to shadow instances that are imported from other modules.

8.1 Introduction

Consider the following example program to illustrate the problem:

$$
\begin{aligned}
&\textbf{class } \mathit{Eq}\ a \textbf{ where}\\
&\quad (\equiv) :: a \to a \to \mathit{Bool}\\
&\textbf{instance } \mathit{Eq}\ \mathit{Int} \textbf{ where}\\
&\quad (\equiv) = \mathit{primEqInt}\\
&\mathit{test1} = 90 \equiv 450\\
&\mathit{test2} = \textbf{let instance } \mathit{Eq}\ \mathit{Int} \textbf{ where}\\
&\qquad\qquad x \equiv y = \mathit{primEqInt}\ (\mathit{mod}\ x\ 360)\ (\mathit{mod}\ y\ 360)\\
&\qquad \textbf{in}\ \ 90 \equiv 450
\end{aligned}
$$

With local instances, two instances for *Eq Int* are available: one in the global scope and one in the local scope. The global instance must be used to resolve overloading in the function *test1* because that is the only instance in scope. A local instance declaration shadows an instance declaration introduced at an outer level, and thus the local instance for *Eq Int* is used to resolve overloading in the function *test2*.

We could choose to not solve local instances using our framework and leave this to the compiler that uses this framework. This can be achieved by simplifying predicates at each scope using our framework together with an environment containing the instances in scope. However, we choose to formulate local instances using constraints.

- Therefore, we annotate each predicate with a scope identifier and encode the lexical scoping rules into CHRs. The number of design decisions increases when allowing local instances. Again, we do not encode design decisions into the constraints and CHRs.
- Instead, we generate every correct context reduction alternative and let a heuristic choose the preferred solution. *Correct* in this sense means that the rules must respect lexical scoping, but not shadowing. For instance, it is correct to use the global instance for *Eq Int* in the function *test2*, however, using the local instance in the function *test1* is not correct.

8.2 Lexical scoping

Lexical or static scoping is a static property of a program. This makes it easier to reason about code, because it allows the programmer to reason as if variable bindings are carried out by substitution.

$$\frac{P \Vdash_e \{\pi_t\} \quad t \text{ `visibleIn` } s}{P \Vdash_e \{\pi_s\}}\,(Scope) \qquad \frac{P \Vdash_e Q \qquad (Inst\ Q \Rightarrow_t \pi) \in \Gamma \qquad t \text{ `visibleIn` } s}{P \Vdash_e \{\pi_s\}}\,(Inst)$$

Figure 8.1: Entailment rule for scoping and the adapted rule for instances

The structure of lexical scopes in a program can be represented with a tree corresponding to the nesting structure of the source text. A scope is then defined as the path of the root to a node. We represent a tree path with a list of integers. The length of this list is equal to the depth of the scope and each child is identified by an integer which is unique within the directly enclosing parent scope. The empty list identifies the global scope.

```
type TreePath = [Int]
isPrefixOf :: Eq a ⇒ [a] → [a] → Bool
(x : xs) `isPrefixOf` (y : ys) = x ≡ y ∧ xs `isPrefixOf` ys
[]       `isPrefixOf` _        = True
_        `isPrefixOf` []       = False
visibleIn :: TreePath → TreePath → Bool
visibleIn = isPrefixOf
```

We can use the standard function *isPrefixOf* to check if something from scope s is visible in scope t. For convenience, we abbreviate *isPrefixOf* with *visibleIn*. For example, [] is visible in every scope and $[1, 2]$ is visible in $[1, 2, 2]$. In the following sections, we use tree paths to annotate predicates with their scope.

8.3 Entailment

We first adapt the entailment relation before giving a translation into CHRs. Each predicate π is annotated with a scope identifier s (π_s). For a *Prove* constraint this means that π must be proven using the CHRs visible in scope s. An assumption of π in scope s means that π is available in scope s. In Figure 8.1 we present new entailment rules to replace the rules presented in Chapter 2 (figures 2.3 and 2.4).

The *Scope* rule states that P entails π in scope s if P entails π in a parent scope t. For example, consider the constraint set $\{Assume\ (Ord\ a_{[1]}), Prove\ (Ord\ a_{[1,1]})\}$. There is an assumption available for *Ord a* and in an inner scope *Ord a* must be proven. We have to use the *Scope* rule to proof that $\{Ord\ a_{[1]}\} \Vdash_e \{Ord\ a_{[1,1]}\}$. Another situation where we need this rule is with the constraint set $\{Prove\ (Ord\ a_{[1,2]}), Prove\ (Ord\ a_{[1,1]})\}$. Two proof obligations concerning the same predicate occur in two sibling scopes. The duplicate predicates in this set can be removed, because $\{Ord\ a_{[1]}\} \Vdash_e \{Ord\ a_{[1,1]}\ Ord\ a_{[1,2]}\}$.

The *Inst* rule in Figure 8.1 is the adapted version of Figure 2.4. This rule states that only instances in scope can be used to resolve overloading. Note that the predicates Q in the *Inst* rule are instantiated to scope s. For example, the entailment relation $\{Eq\ a_{[1,1]}\} \Vdash_e \{Eq\ [a]_{[1,1]}\}$ holds when $(Inst\{Eq\ a\} \Rightarrow_{[]} Eq\ [a]) \in \Gamma$.

8.4 Translation to CHRs

We now present a translation from class and instance declaration into CHRs. First, we explain which CHRs must be generated for instance declarations. Then we present two alternative encodings of the *Scope* rule.

8.4.1 Instance declarations

To implement the *Inst* rule of Figure 8.1, we have to adapt the translation from instance declarations into CHRs. We generate the following CHR for each instance declaration: **instance** $(\pi_1, .., \pi_n) \Rightarrow \pi$, in scope ς.

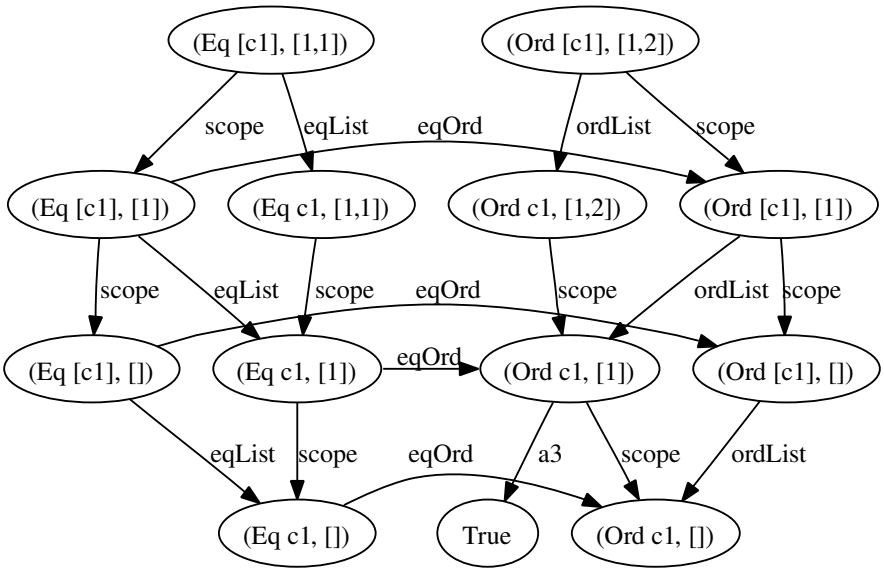

Figure 8.2: Simplification graph generated using the first encoding of the *Scope* rule

$$\begin{aligned} \mathit{Prove}\ (\pi, s) \Longrightarrow\ & \varsigma\ `\mathit{visibleIn}`\ s \\ \mid\ & \mathit{Reduction}\ (\pi, s)\ (\mathit{ByInstance}\ \pi\ \varsigma)\ [(\pi_1, s), ..., (\pi_n, s)] \\ ,\ & \mathit{Prove}\ (\pi_1, s), ..., \mathit{Prove}\ (\pi_n, s) \end{aligned}$$

The reduction constraint generated by this rule is annotated with the head of the instance declaration and a scope identifier. This annotation is used in the next section to define a heuristic for local instances. The difference with the translation presented in the previous chapter is that the predicates are annotated with a scope and that a guard checks whether the rule is in scope. For example, the following rule is generated for the global instance **instance** $\mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a]$:

$$\begin{aligned} \mathit{Prove}\ (\mathit{Eq}\ [a], s) \Longrightarrow\ & []\ `\mathit{visibleIn}`\ s \\ \mid\ & \mathit{Reduction}\ (\mathit{Eq}\ [a], s)\ (\mathit{ByInstance}\ \texttt{"eqList"}\ [])\ [(\mathit{Eq}\ a, s)] \\ ,\ & \mathit{Prove}\ (\mathit{Eq}\ a, s) \end{aligned}$$

8.4.2 First encoding of scoping

We only need the following CHR to implement the *Scope* rule of Figure 8.1:

$$\mathit{Prove}\ (p, s) \Longrightarrow \mathit{not}\ (\mathit{null}\ s) \mid \mathit{Prove}\ (p, \mathit{init}\ s), \mathit{Reduction}\ (p, s)\ (\mathit{ByScope}\ (\mathit{init}\ s))\ [(p, \mathit{init}\ s)]$$

This CHR means that a predicate can be reduced to a predicate in the parent scope. The guard prevents the rule to be applied in the global scope, because the global scope does not have a parent scope. With this rule we encode the *Scope* rule in a very concise way corresponding directly to the *Scope* entailment rule. Note that the generated reduction constraint is annotated with the scope we reduce to. This annotation is used in the next section to define a heuristic for local instances. In Figure 8.2 we present the graph generated when using this rule for solving the constraints $\{\mathit{Assume}\ (\mathit{Ord}\ c_1, [1]), \mathit{Prove}\ (\mathit{Ord}\ [c_1], [1, 2]), \mathit{Prove}\ (\mathit{Eq}\ [c_1], [1, 1])\}$. The drawback of this CHR is that it is applicable to every proof obligation that is not in the global scope. This means that many reduction alternatives are generated which are not always very useful. For that reason, we propose a second encoding of scoping.

8.4.3 Second encoding of scoping

We propose an alternative encoding of the *Scope* rule to reduce the number of alternatives generated. The idea behind the new rules is that a reduction to a parent scope is only inserted when this results in further constraint simplification. The following situations are reasons to insert a scope reduction:

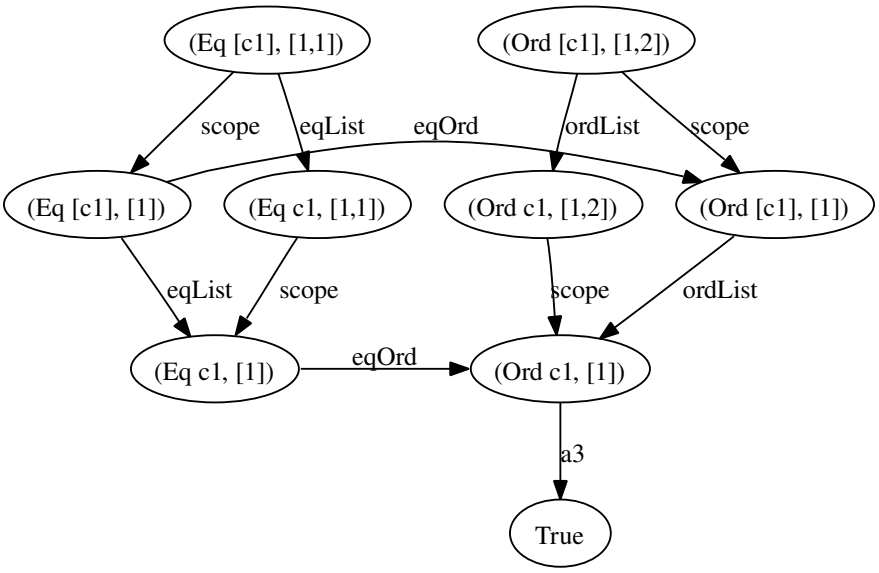

Figure 8.3: Simplification graph generated using the second encoding of the *Scope* rule

- When two *Prove* constraints concerning the same predicate appear in different scopes.
- When two *Prove* constraints concerning the same type appear in different scopes, where one predicate is a subclass of the other.

In both cases constraints can be simplified by removing duplicates or by simplification using the class hierarchy. Later in this section we will discover a third reason to insert scope reductions. We introduce the following CHR to detect and handle the first situation:

$$
\begin{aligned}
&\mathit{Prove}\ (p, s), \mathit{Prove}\ (p, t) \Longrightarrow \mathit{not}\ (s\ `\mathit{visibleIn}`\ t)\\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{Prove}\ (p, \mathit{commonPrefix}\ s\ t)\\
&\qquad\qquad\qquad\qquad\quad ,\ \mathit{Reduction}\ (p, s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{ByScope}\ (\mathit{commonPrefix}\ s\ t))\\
&\qquad\qquad\qquad\qquad\qquad\qquad [(p, \mathit{commonPrefix}\ s\ t)]
\end{aligned}
$$

The head of this CHR matches if there are two prove constraints in different scopes concerning the same predicate p. A reduction for the first constraint is only generated when the scope of the first constraint is not visible in the scope of the second constraint. This guard is needed to prevent cycles in the simplification graphs resulting from reductions from the first constraint to itself. Note that a single constraint never matches both the first and the second pattern of the head at the same time. However, when a constraint matches multiple patterns of a head, the constraint will be matched against each pattern in sequence when possible. The greatest common prefix of two scopes is computed with the following function:

$$
\begin{aligned}
&\mathit{commonPrefix} :: \mathit{Eq}\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a]\\
&\mathit{commonPrefix}\ (x : xs)\ (y : ys) \mid x \equiv y \quad\ = x : \mathit{commonPrefix}\ xs\ ys\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = [\,]\\
&\mathit{commonPrefix}\ \_\qquad \_\qquad\qquad\quad\ = [\,]
\end{aligned}
$$

The second situation where scope reductions are inserted is when two *Prove* constraints exist in different scopes which can be reduced using the class hierarchy. To detect this situation, the following rules are generated for each predicate pair (π_1, π) where π_1 is a superclass of π.

$$
\begin{aligned}
&\mathit{Prove}\ (\pi, s), \mathit{Prove}\ (\pi_1, t) \Longrightarrow \mathit{not}\ (s\ `\mathit{visibleIn}`\ t)\\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{Prove}\ (\pi, \mathit{commonPrefix}\ s\ t)\\
&\qquad\qquad\qquad\qquad\quad ,\ \mathit{Reduction}\ (\pi, s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{ByScope}\ (\mathit{commonPrefix}\ s\ t))\\
&\qquad\qquad\qquad\qquad\qquad\qquad [(\pi, \mathit{commonPrefix}\ s\ t)]\\
&\mathit{Prove}\ (\pi, s), \mathit{Prove}\ (\pi_1, t) \Longrightarrow \mathit{not}\ (t\ `\mathit{visibleIn}`\ s)\\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{Prove}\ (\pi_1, \mathit{commonPrefix}\ s\ t)\\
&\qquad\qquad\qquad\qquad\quad ,\ \mathit{Reduction}\ (\pi_1, t)
\end{aligned}
$$

$$\begin{array}{l} (\mathit{ByScope}\ (\mathit{commonPrefix}\ s\ t)) \\ [(\pi_1, \mathit{commonPrefix}\ s\ t)] \end{array}$$

These rules are very similar to the one for *Prove* constraints concerning the same predicates. Again, we annotate the generated reduction constraint with the scope we reduce to. The difference is that we have to generate the above rules for each superclass of a class. For example, the following rules are generated for the declaration **class** $\mathit{Eq}\ a \Rightarrow \mathit{Ord}\ a$:

$$\begin{array}{lll} \mathit{Prove}\ (\mathit{Ord}\ a, s), \mathit{Prove}\ (\mathit{Eq}\ a, t) \Longrightarrow & \mathit{not}\ (s\ `\mathit{visibleIn}`\ t) & \\ \quad | & \mathit{Prove}\ (\mathit{Ord}\ a, \mathit{commonPrefix}\ s\ t) & \\ \quad , & \mathit{Reduction}\ (\mathit{Ord}\ a, s) & \\ & \qquad (\mathit{ByScope}\ (\mathit{commonPrefix}\ s\ t)) & \\ & \qquad [(\mathit{Ord}\ a, \mathit{commonPrefix}\ s\ t)] & \\ \mathit{Prove}\ (\mathit{Ord}\ a, s), \mathit{Prove}\ (\mathit{Eq}\ a, t) \Longrightarrow & \mathit{not}\ (t\ `\mathit{visibleIn}`\ s) & \\ \quad | & \mathit{Prove}\ (\mathit{Eq}\ a, \mathit{commonPrefix}\ s\ t) & \\ \quad , & \mathit{Reduction}\ (\mathit{Eq}\ a, t) & \\ & \qquad (\mathit{ByScope}\ (\mathit{commonPrefix}\ s\ t)) & \\ & \qquad [(\mathit{Eq}\ a, \mathit{commonPrefix}\ s\ t)] & \end{array}$$

One rule is generated to add a scope reduction for the class and one to add a scope reduction for the superclass.

There is a third situation in which we insert a scope reduction: when there is an assumptions and a prove obligation for the same predicate. The following rule is added to detect this situation.

$$\begin{array}{lll} \mathit{Prove}\ (p, s), \mathit{Assume}\ (p, t) \Longrightarrow & s \not\equiv t & \\ \quad , & t\ `\mathit{visibleIn}`\ s & \\ \quad | & \mathit{Reduction}\ (p, s)\ (\mathit{ByScope}\ t)\ [(p, t)] & \end{array}$$

To sum up, we have to modify the systematic translation with respect to the following points:

- A guard must be added in the translation of instance declaration to check if the instance is in scope.
- A CHR must be added to simplify *Prove* constraints concerning the same predicate in different scopes.
- A CHR must be added to simplify a *Prove* constraint and an *Assume* constraint concerning the same predicate in different scopes.
- CHRs must be generated for the transitive closure of the superclass relation to simplify *Prove* constraints in different scopes where one predicate is a subclass of the other.

In Figure 8.3 we present the graph generated when using the new translation for solving the constraints $\{\mathit{Assume}\ (\mathit{Ord}\ c_1, [1]), \mathit{Prove}\ (\mathit{Ord}\ [c_1], [1,2]), \mathit{Prove}\ (\mathit{Eq}\ [c_1], [1,1])\}$. Compared to Figure 8.2, the number of edges is decreased considerably because no superfluous reductions are generated anymore.

8.5 Heuristic

We also have to define a new heuristic to choose the preferred solution. For example, we must encode that a local instance shadows a global instance. The different constraints are annotated with the values from the following datatype:

$$\begin{array}{lll} \textbf{data}\ \mathit{Annotation} = & \mathit{ByInstance} & \mathit{String}\ \mathit{TreePath} \\ \quad | & \mathit{BySuperClass} & \mathit{String} \\ \quad | & \mathit{Assumption} & \mathit{Int} \\ \quad | & \mathit{ByScope} & \mathit{TreePath} \\ \quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) & & \end{array}$$

The difference with the annotations presented in Subsection 7.4.3 is that we have added a constructor for scoping and that the constructor *ByInstance* has an additional field of type *TreePath*. The field *TreePath* is added to identify the scope of the used instance declaration. Reductions generated by the CHRs for scoping are annotated with *ByScope*. The field *TreePath* of the constructor *ByScope* is the scope where the predicate is reduced to. Using these annotations we define the following heuristic:

$$\begin{array}{l} \mathit{scoped} :: \mathit{Annotation} \to \mathit{Annotation} \to \mathit{Ordering} \\ \mathit{scoped}\ (\mathit{ByInstance}\ \_\ s)\ (\mathit{ByInstance}\ \_\ t) = (\mathit{length}\ s)\ `\mathit{compare}`\ (\mathit{length}\ t) \\ \mathit{scoped}\ (\mathit{BySuperClass}\ \_)\ \_ = \mathit{GT} \\ \mathit{scoped}\ \_\ (\mathit{BySuperClass}\ \_) = \mathit{LT} \end{array}$$

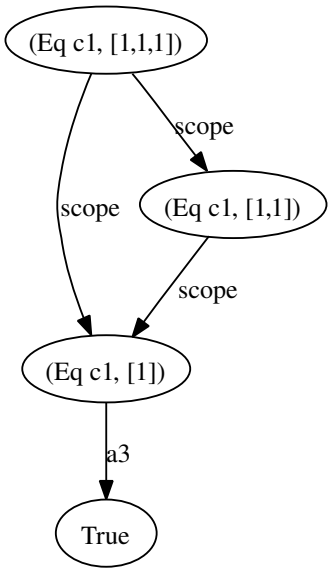

Figure 8.4: Multiple alternatives for reducing to a parent scope.

$$
\begin{aligned}
&\mathit{scoped}\ (\mathit{Assumption}\ \_)\ \_ &&= \mathit{GT}\\
&\mathit{scoped}\ \_ \quad (\mathit{Assumption}\ \_) &&= \mathit{LT}\\
&\mathit{scoped}\ (\mathit{ByScope}\ s)\ (\mathit{ByScope}\ t) &&= (\mathit{length}\ s)\ `\mathit{compare}`\ (\mathit{length}\ t)\\
&\mathit{scopedHeuristic} = \mathit{toHeuristic}\ \$\ \mathit{binChoice}\ \mathit{scoped}
\end{aligned}
$$

The rule that a local instance shadows a global instance is encoded in the first case of the function *scoped*. Instances are ordered by comparing the depth of the scope. Predicates are first reduced using instances, the class hierarchy, and assumptions. A predicate is only reduced to a parent scope when there are no other options left.

In the last case of the heuristic we also compare the depth of the scopes. The reason for this is that there are sometimes multiple alternatives for reducing a predicate to a parent scope. For example, the graph in Figure 8.4 results from the following constraint set:

$$\{\mathit{Prove}\ (\mathit{Eq}\ c_1, [1,1,1]), \mathit{Prove}\ (\mathit{Eq}\ c_1, [1,1]), \mathit{Assume}\ (\mathit{Eq}\ c_1, [1])\}$$

Two alternatives for reducing the predicate $(\mathit{Eq}\ c_1, [1,1,1])$ are generated: one resulting from the *Prove* constraint in the parent scope and the other resulting from the *Assume* constraint. We annotate both alternatives with different values to be able to make a deterministic choice.

8.6 Conclusion

In this chapter we have given entailment rules for scoped instances. We use the entailment rules as the starting point for a first translation. Although the first translation is correct, many redundant reductions are generated. As an alternative, we have presented a second translation to minimize the number of such reductions.

We emphasize that every possible reduction is generated and represented in a graph. This maximizes the opportunities to experiment with scoped instances by testing different heuristics. For example, a programmer could indicate that a constraint must be resolved with instances from the global scope. The heuristic presented in this chapter nicely illustrates how the rule for preferring local instances can be encoded.

This chapter also shows the flexibility of the framework. We have used predicates annotated with scope identifiers and also showed how additional features can be implemented with small modifications in the translation.

Chapter 9

Improving substitution

Until now we have only considered resolution of overloading by means of simplification. In this chapter we extend the framework with improvement of constraint sets.

9.1 Introduction

We have shown how overloading can be resolved using CHRs and heuristics. For example, proof obligations for type class qualifiers are simplified using instance declarations and the class hierarchy. Besides simplification, constraint sets can also be solved by means of improvement [Jones, 1995c]. We give some examples to illustrate what improvement is and why we need it.

The first description of type classes [Wadler and Blott, 1989] already mentions multi-parameter type classes. However, multi-parameter type classes often lead to ambiguities and delayed detection of type errors. As a solution, functional dependencies were introduced [Jones, 2000]. Functional dependencies allow the programmer to explicitly define relations between type class parameters. The following standard example illustrates multi-parameter type classes and functional dependencies:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Coll}\ c\ e \mid c \rightarrow e\ \textbf{where} \\
\quad \mathit{empty} \quad :: c \\
\quad \mathit{insert} \quad :: e \rightarrow c \rightarrow c \\
\quad \mathit{member} :: e \rightarrow c \rightarrow \mathit{Bool}
\end{array}
$$

Coll is a type class for homogeneous collections, where c represents the type of the collection and e the type of the elements. Assume that we use this type class without the functional dependency $c \rightarrow e$. The first ambiguity occurs when using the function *empty* with type $(\mathit{Coll}\ c\ e) \Rightarrow c$ because the type variable e occurs in the context but not in the type. Second, the types assigned to *insert* and *member* are more liberal than we expect. For example, consider the following function:

$$\mathit{test}\ \mathit{xs} = \mathit{insert}\ \texttt{'c'}\ (\mathit{insert}\ \mathit{False}\ \mathit{xs})$$

The type $(\mathit{Coll}\ v_1\ \mathit{Bool}, \mathit{Coll}\ v_1\ \mathit{Char}) \Rightarrow v_1 \rightarrow v_1$ is inferred for this function instead of the type error we expect for inserting two values of different types into the same collection. With functional dependencies these problems can be solved because now we are able to specify that the type of the collection uniquely determines the type of the elements ($c \rightarrow e$). In other words, if we have two type class predicates $\mathit{Coll}\ c\ a$ and $\mathit{Coll}\ c\ b$, then a must be equal to b because both constraints have type c as the first parameter. The substitution $\{a \mapsto b\}$ is then an improvement for the type class predicate $\mathit{Coll}\ c\ a$. Type checking the function *test* with the functional dependency result in the type error we expected because *Int* is not equal to *Bool*.

Functional dependencies are tricky because they can lead to inconsistencies and non-termination. Therefore, Sulzmann et al. [Duck et al., 2004, Sulzmann et al., 2007] formulate functional dependencies in terms of CHRs. By making use of CHRs the authors prove that under some sufficient conditions, functional dependencies allow for sound, complete, and decidable type inference. In this chapter we extend the framework with improvement and show how the proposed translation of Sulzmann et al. can be used.

$$\frac{S\ \tau_1 = S\ \tau_2}{S\ \Vdash_e \{\tau_1 \equiv \tau_2\}}\ (\mathit{Elimination}) \qquad \frac{(P \Longrightarrow \tau_1 \equiv \tau_2)\ \in\ \Gamma \quad S\ \Vdash_e \{\tau_1 \equiv \tau_2\}}{SQ\ \Vdash_e\ SP}\ (\mathit{Introduction})$$

Figure 9.1: Introduction and elimination of equality predicates

9.2 Definition of improvement

Improving a set of qualifiers P results in an improving substitution S. Improvement can be applied at any stage during the type inference process. Intuitively, applying an improving substitution on a predicate set P replaces unsatisfiable predicates by satisfiable predicates. Formally, this is defined in terms of satisfiable instances [Jones, 1995c]:

$$\lfloor P \rfloor_Q = \{\, SP \mid S\ \in\ \mathit{Subst}, Q\ \Vdash_e\ SP \,\}$$

The satisfiable instances of P with respect to Q are $\lfloor P \rfloor_Q$. S *improves* P when $\lfloor P \rfloor_Q \equiv \lfloor SP \rfloor_Q$. Compared to simplification, the evidence for improvement is the resulting substitution. The difference is that often a choice can be made to apply a simplification step; however, the substitution generated by improvement is a fact that must be respected. This substitution must be applied to the predicates and types in the framework and in the compiler that uses this framework. For example, consider the following function for inserting two elements into a collection:

$$\mathit{insTwo}\ x\ y\ c = \mathit{insert}\ x\ (\mathit{insert}\ [y]\ c)$$

The constraints $\{\mathit{Prove}\ (\mathit{Coll}\ v_1\ v_2), \mathit{Prove}\ (\mathit{Coll}\ v_1\ [v_3])\}$ result from the two usages of *insert* in this function. The inferred type for this function without the functional dependency would be $(\mathit{Coll}\ v_1\ v_2, \mathit{Coll}\ v_1\ [v_3]) \Rightarrow v_2 \to v_3 \to v_1 \to v_1$. One of the two constraints is not satisfiable, because we cannot give evidence in the form of two functions inserting elements of different types into a collection of the same type. Luckily, the functional dependency of the type class *Coll* results in the improving substitution $\{v_2 \mapsto [v_3]\}$. Applying the improving substitution on the earlier inferred type results in the type $\mathit{Coll}\ v_1\ [v_3] \Rightarrow [v_3] \to v_3 \to v_1 \to v_1$ for the function *insTwo*.

9.3 Approach

We use the following approach to add improvement to our framework. *Prove* constraints can be solved by simplification using the class hierarchy and using instance declarations. We encode simplification in terms of CHRs generating every correct context reduction alternative. Evidence in the form of a tree is generated from reduction alternatives for each *Prove* constraint. Simplification is a method for solving *Prove* constraints. We add improvement as an additional method for solving *Prove* constraints. The evidence for solving a *Prove* constraint using improvement is a substitution instead of an evidence tree.

No single *Prove* constraint can be solved using both simplification and improvement. For example, the constraint $\mathit{Prove}\ (v_1 \equiv v_2)$ cannot be solved using simplification in the case of type class predicates. However, the improving substitution $\{v_1 \mapsto v_2\}$ is a solution for this constraint. We do not make any assumption about the predicate language of the compiler that uses our framework. However, in practice, the predicate language requires some form of equality predicates:

$$\begin{aligned} \pi &:= \tau_1 \equiv \tau_2 \\ &\mid \ \ldots \end{aligned}$$

The leftmost rule in Figure 9.1 specifies when an equality predicate is solved. An equality predicate is solved when there is a substitution S where $S\ \tau_1$ and $S\ \tau_2$ are syntactically equivalent. The compiler that uses our framework has to specify how a *Prove* constraint can be solved using improvement. Usually, this will consist of a function unifying both components of the equality predicate (τ_1 and τ_2).

The rightmost rule in Figure 9.1 specifies how equality predicates are introduced. An equality predicate is introduced when it is implied by a matching CHR. In this way, the described translation for functional

dependencies into CHRs [Duck et al., 2004, Sulzmann et al., 2007] can easily be used in this framework. We explain the translation using a well known example. Consider the following class declaration for encoding a family of zip functions:

$$\begin{array}{l}\textbf{class } \mathit{Zip}\ a\ b\ c \mid c \rightarrow a, c \rightarrow b \textbf{ where}\\ \quad \mathit{zip} :: [a] \rightarrow [b] \rightarrow c\end{array}$$

A CHR is generated for each functional dependency:

$$\begin{array}{l}\mathit{Prove}\ (\mathit{Zip}\ a\ b\ c), \mathit{Prove}\ (\mathit{Zip}\ d\ e\ c) \Longrightarrow \mathit{Prove}\ (a \equiv d)\\ \mathit{Prove}\ (\mathit{Zip}\ a\ b\ c), \mathit{Prove}\ (\mathit{Zip}\ d\ e\ c) \Longrightarrow \mathit{Prove}\ (b \equiv e)\end{array}$$

The parameters a and b of the type class *Zip* are uniquely determined by c. Therefore, the fresh variables d and e are introduced in the CHRs for a and b respectively. Proof obligation for equality predicates are generated when there are two *Prove* constraints in the constraint set mentioning the same type c. CHRs are also generated for each instance of a class with functional dependencies:

$$\begin{array}{l}\textbf{instance } \mathit{Zip}\ a\ b\ [(a, b)] \textbf{ where}\\ \quad \mathit{zip}\ (x : xs)\ (y : ys) = (x, y) : \mathit{zip}\ xs\ ys\\ \quad \mathit{zip}\ \_\qquad\ \_\qquad\ = [\,]\end{array}$$

The following CHRs are generated for the instance declaration in combination with the functional dependencies:

$$\begin{array}{l}\mathit{Prove}\ (\mathit{Zip}\ d\ e\ [(a, b)]) \Longrightarrow \mathit{Prove}\ (d \equiv a)\\ \mathit{Prove}\ (\mathit{Zip}\ d\ e\ [(a, b)]) \Longrightarrow \mathit{Prove}\ (e \equiv b)\end{array}$$

The type $[(a, b)]$ in the instance declaration uniquely determines the type variables a and b. Therefore, proof obligations for equality predicates are generated when a $\mathit{Prove}\ (\mathit{Zip}\ d\ e\ [(a, b)])$ is matched in the constraint set. The big advantage of translating functional dependencies into CHRs is that all improvements are made explicit.

On two points we deviate from the translation described by Sulzmann et al. The first deviation is that we use explicit *Prove* constraints. Second, we do not propagate functional dependencies from superclasses. For example, consider the following class declaration:

$$\textbf{class } \mathit{Zip}\ a\ b\ c \Rightarrow C\ a\ b\ c$$

The following CHR is generated for the class declaration when using the translation described by Sulzmann et al:

$$C\ a\ b\ c \Longrightarrow \mathit{Zip}\ a\ b\ c$$

This CHR propagates the class hierarchy and thereby also the improvement CHRs of the superclasses. For example, the constraint $(C\ v_1\ v_2\ [(\mathit{Int}, \mathit{Bool})])$ will also trigger the improvement CHRs generated for functional dependencies of type class *Zip*. The result of applying the CHRs on the constraint $(C\ v_1\ v_2\ [(\mathit{Int}, \mathit{Bool})])$ is then:

$$\begin{array}{l}\{\, C\ \ v_1\ v_2\ [(\mathit{Int}, \mathit{Bool})]\\ , \mathit{Zip}\ v_1\ v_2\ [(\mathit{Int}, \mathit{Bool})]\\ , v_1 \equiv \mathit{Int}\\ , v_2 \equiv \mathit{Bool}\,\}\end{array}$$

However, the class hierarchy is not propagated in the translation scheme we proposed. This means that the CHRs generated for functional dependencies of class π must also incorporate the functional dependencies of the superclasses of class π.

9.4 Implementation

In this section we extend the framework to support improvement. This extension does not influence the earlier described work on evidence translation and can be described independently. However, there exist some interaction between simplification and improvement which we will describe at the end of this section. First, we have to extend the state of the solver to store the list of CHRs describing the improvement relation.

$$\begin{array}{l}\textbf{data } \mathit{SolveState}\ p\ s\ \mathit{info} =\\ \quad \mathit{SolveSt}\{\, \mathit{rules}\qquad :: [\mathit{CHR}\ (\mathit{Constraint}\ p\ \mathit{info})\ s]\end{array}$$

```
, imprRules :: [CHR (Constraint p info) s]
, ...
}
```

We could combine the lists of CHRs for improvement and simplification into one list. However, we separate those lists because improvement and simplification are two different steps and it is more clear to describe both relations separately. Improvement can be applied at any stage during the type inference process with the following function:

```
improve :: (MonadState (SolveState p s info) m, Matchable p s, Ord info, Substitutable p v a)
        ⇒ (p → Maybe a) → m a
improve impr =
  do rls      ← gets imprRules
     cnstrs ← gets constraints
     let equalities   = chrSolveList rls (Map.keys cnstrs)
         (s, cnstrs') = foldr (imprSubst impr) (mempty, cnstrs) equalities
     modifyConstraints (const cnstrs')
     return s
```

The above function is parametrized with the function *impr*. This function parameter must be supplied by the compiler using this framework and is required to find an improvement for predicates. The function *improve* first generates equality constraints by applying the CHRs for improvement. Then, the function *imprSubst* is folded over the generated constraints. This function constructs the improving substitution from the generated constraints using the *impr* function.

```
imprSubst :: (Ord p, Ord info, Substitutable p v a) ⇒ (p → Maybe a) → Constraint p info
          → (a, Constraints p info) → (a, Constraints p info)
imprSubst impr c@(Prove p) (s, cs) =
                  case impr (substitute s p) of
                    Nothing → (s,                 Map.insertWith (++) c [] cs)
                    Just s' → (s' `mappend` s, Map.delete c cs)
imprSubst _     c (s, cs) = (s,                 Map.insertWith (++) c [] cs)
```

Assume constraints are solved by just inserting them into the constraint map. Solving a *Prove* constraint consists of the following steps. First, the improving substitution thus far is applied to predicate p. Then, the function *impr* is applied to the result of applying the substitution. The function *impr* tries to find an improving substitution for solving the constraint and gives the following result:

- The function evaluates to *Nothing*. This means that this predicate cannot be solved using an improving substitution.
- Otherwise, the function returns an improving substitution and thereby solving the predicate.

For example, consider the following function from the introduction again:

```
test xs = insert 'c' (insert False xs)
```

The constraints $\{Prove\ (Coll\ v_1\ Char), Prove\ (Coll\ v_1\ Bool)\}$ are generated for this function. Solving these constraints with the CHR $Prove\ (Coll\ a\ b), Prove\ (Coll\ a\ c) \Longrightarrow Prove\ (b \equiv c)$ for improvement results in the constraint $Prove\ (Char \equiv Bool)$. There is no improving substitution for solving the constraint $Prove\ (Char \equiv Bool)$ so this constraint will remain unresolved and can be reported as an error.

It is possible that improvement can lead to new simplifications and the other way around. For example, consider the following example program:

```
class Coll c e | c → e where
  empty  :: c
  insert  :: e → c → c
  member :: e → c → Bool

instance Ord a ⇒ Coll [a] a where
  empty  = []
  insert  = (:)
  member = elem

replaceHead (_ : xs) y = insert y xs
```

The following CHRs are generated for this program:

$$
\begin{array}{lll}
\mathit{Prove}\ (\mathit{Coll}\ a\ b), \mathit{Prove}\ (\mathit{Coll}\ a\ c) & \Longrightarrow \mathit{Prove}\ (b \equiv c) & \text{-- (C1)} \\
\mathit{Prove}\ (\mathit{Coll}\ [a]\ b) & \Longrightarrow \mathit{Prove}\ (a \equiv b) & \text{-- (I1)} \\
\mathit{Prove}\ (\mathit{Coll}\ [a]\ a) & \Longrightarrow \mathit{Prove}\ (\mathit{Ord}\ a) & \\
 & \quad , \ \mathit{Reduction}\ (\mathit{Coll}\ [a]\ a)\ \texttt{"collList"}\ [\mathit{Ord}\ a] & \text{-- (I1)}
\end{array}
$$

The first two CHRs are improvement rules generated for the class and instance declaration respectively and the last CHR is a simplification CHR generated for the instance declaration for list collections. The constraint $\mathit{Prove}\ (\mathit{Coll}\ [v_1]\ v_2)$ is generated for the use of the overloaded function *insert* in the function *replaceHead*. The following function applies simplification and improvement in a fix-point computation:

$$
\begin{array}{l}
\mathit{fixImprove} :: (\mathit{MonadState}\ (\mathit{SolveState}\ p\ s\ \mathit{info})\ m, \mathit{Matchable}\ p\ s, \mathit{Ord}\ \mathit{info}, \mathit{Substitutable}\ p\ v\ a) \\
\qquad \Rightarrow (p \to \mathit{Maybe}\ a) \to a \to m\ a \\
\mathit{fixImprove}\ \mathit{impr}\ s = \\
\quad \textbf{do}\ \mathit{simplify} \\
\qquad s' \leftarrow \mathit{improve}\ \mathit{impr} \\
\qquad \textbf{if}\ s' \equiv \mathit{mempty} \\
\qquad\quad \textbf{then}\ \mathit{return}\ s \\
\qquad\quad \textbf{else}\ \ \textbf{do}\ \mathit{applySubstitution}\ s' \\
\qquad\qquad\qquad \mathit{fixImprove}\ \mathit{impr}\ (s\ `\mathit{mappend}`\ s') \\
\mathit{applySubstitution} :: (\mathit{MonadState}\ (\mathit{SolveState}\ p\ s\ \mathit{info})\ m, \mathit{Ord}\ p, \mathit{Ord}\ \mathit{info}, \mathit{Substitutable}\ p\ v > a) \\
\qquad \Rightarrow a \to m\ () \\
\mathit{applySubstitution}\ s = \\
\quad \textbf{do}\ \mathit{modifyConstraints}\ (\mathit{Map.mapKeysWith}\ (+\!\!+)\ (\mathit{substitute}\ s)) \\
\qquad \mathit{modifyEvidence}\quad (\mathit{Map.map}\qquad\qquad\quad (\mathit{substitute}\ s))
\end{array}
$$

First, the constraints are simplified. In the current example no simplification CHR can be matched against the constraint $\mathit{Prove}\ (\mathit{Coll}\ [v_1]\ v_2)$. Second, the constraints are improved which results for the current example in the substitution $\{v_2 \mapsto v_1\}$. The substitution is applied to the constraints and a recursive call to *fixImprove* takes place. Now, a simplification can be applied to the substituted constraint $\mathit{Prove}\ (\mathit{Coll}\ [v_1]\ v_1)$ resulting in the constraint $\mathit{Prove}\ (\mathit{Ord}\ v_1)$. No improvement rule can be matched against $\mathit{Prove}\ (\mathit{Ord}\ v_1)$ so the resulting substitution is empty and the fix-point is reached.

9.5 Conclusion

In this chapter we have given a short introduction to improvement and explained why improvement is useful. For example, multi-parameter type classes are mostly useful in the context of functional dependencies. Functional dependencies can be translated into CHRs [Duck et al., 2004, Sulzmann et al., 2007]. The advantage of this translation is that CHRs make the improvements generated by functional dependencies explicit. We have explained how this translation can be used in our framework with some minor modifications.

The framework now supports improvement and simplification of qualified types. Thereby, we have given an implementation for the theoretical framework of Jones [Jones, 1992, 1995c,b]. However, improvement is in some sense an ad-hoc extension of our framework. We did not mention how improvement scales in combination with other extensions such as overlapping instances or local instances. It would be nice to express improvements in the simplification graphs. Improvements are then generated depending on the solution chosen by heuristics. This would be an interesting research topic and we think our framework provides a solid basis for this research.

Chapter 10

Conclusion and future work

We have presented a constraint-based framework for the resolution of overloading. In this chapter we first list the contributions of our work and then give an overview of future work.

10.1 Conclusion

The framework presented in this thesis was preceded by a prototype. We tested and implemented this prototype in the type checker 'Typing Haskell in Haskell' [Jones, 1999] and in the constraint-solver Top [Heeren, 2005]. Class and instance declarations were translated into rules constructing a simplification graph. Evidence could be generated from this graph. However, we used a tailor made rule language instead of the more general CHRs, and the heuristic was hard-coded in the evidence generation algorithm.

In Chapter 6 we presented a first version of the framework. We showed how entailment of Haskell 98 type classes can be expressed in terms of CHRs. Furthermore, a systematic translation was presented from class and instance declarations into CHRs. We also check if the class hierarchy is acyclic during this translation. On top of that, we have also implemented a domain specific language for CHRs together with a basic CHR solver in Haskell. In Chapter 7 we presented the final version of the framework. We let each CHR generate reduction constraints to trace the derivation steps needed to arrive at a solution. Furthermore, we use CHRs to generate every possible and correct context reduction alternative. This is achieved by only using propagation CHRs. The context reduction alternatives generated by CHRs are represented in a graph and a heuristic chooses a solution to resolve overloading. These heuristics allows one to easily experiment with different design decisions. To illustrate this, we presented heuristics emulating the context reduction behavior of Haskell 98 and GHC. Furthermore, we presented a heuristic for the resolution of overlapping instances.

In Chapter 8 we explained how overloading can be resolved in the context of local instances. First, we presented new entailment rules for scoped instances, and then we introduced a new predicate language by annotating predicates with a scope identifier. To generate correct alternatives we adapted the translation from class and instance declarations into CHRs. We only introduced one new CHR for scoping and add a guard to the CHRs generated for instance declarations. However, the CHR for scoping generates many redundant reductions. For this reason, we also presented an alternative encoding of scoping in terms of CHRs.

In Chapter 9 we extended the framework with improvement. Thereby, we have given an implementation for the theoretical framework of Jones [Jones, 1992, 1995c,b]. This extension allows one to use the existing translation of functional dependencies into CHRs [Duck et al., 2004, Sulzmann et al., 2007] in this framework. Furthermore, it is also possible to resolve other qualified types using this framework such as extensible records [Jones and Peyton Jones, 1999].

Finally, we have shown that almost every type-class extension can be formulated in this framework. For example, we have explained how overlapping instances, local instances, multi-parameter type classes, and functional dependencies can be encoded.

10.2 Future work

Type class directives

Type class directives [Heeren and Hage, 2005] are proposed to improve type error messages concerning type classes. Consider the following example directives:

```
never    Eq (a → b)          : "functions cannot be tested for equality"
never    Num Bool            : "arithmetic on booleans is not supported"
close    Integral            : "the only Integral instances are Int and Integer"
disjoint Integral Fractional : "something which is fractional can never be integral"
```

The above directives can easily be translated into the following CHRs:

$$
\begin{array}{l}
Prove\ (Eq\ (a \rightarrow b)) \Longrightarrow error\ \texttt{"functions cannot be tested for equality"} \\
Prove\ (Num\ Bool) \Longrightarrow error\ \texttt{"arithmetic on booleans is not supported"} \\
Prove\ (Integral\ a) \Longrightarrow a \notin \{Int, Integer\} \\
\qquad\qquad\qquad\qquad \mid\ error\ \texttt{"the only Integral instances are Int and Integer"} \\
Prove\ (Integral\ a), Prove\ (Fractional\ a) \\
\qquad\qquad\qquad\qquad \Longrightarrow error\ \texttt{"something which is fractional can never be integral"}
\end{array}
$$

The directives can be applied in the framework by adding the above rules to the CHRs for simplification. However, we did not describe how to handle the errors resulting from resolving overloading. We expect that the type error infrastructure of Top can be inherited by integrating this framework with Top.

Type class directives can also lead to improving substitutions, because more information about the set of types in a class is available. It is interesting to examine how directives in combination with improvement can be encoded using this framework.

Existential types and type classes

Combining existential types and type classes results in significant expressive power [Läufer, 1996]. GHC [Marlow and Peyton Jones, 2006] supports existentially quantified data constructors with arbitrary contexts:

$$
\begin{array}{l}
\textbf{data}\ Showable = forall\ a.Show\ a \Rightarrow Showable\ a \\
showable\ (Showable_{a1}\ x) = show_{p1}\ x \\
test = map\ showable\ [Showable_{p2}\ \texttt{'c'}, Showable_{p3}\ \texttt{"Hello World"}, Showable_{p4}\ False]
\end{array}
$$

Surprisingly, the keyword forall is used to existentially quantify a variable. The reason for this decision is that the type $\forall\ a.Show\ a \Rightarrow a \rightarrow Showable$ is assigned to the constructor *Showable*. This construction allows for packaging heterogeneous values in a list and using the functions of type class *Show* on those values. The following constraints are generated to resolve overloading for the above fragment:

$$
\begin{array}{l}
\{\, Prove\ (Show\ Char)_{p2}, Prove\ (Show\ String)_{p3}, Prove\ (Show\ Bool)_{p4} \\
, Assume\ (Show\ c_1)_{a1}, Prove\ (Show\ c_1)_{p1}\, \}
\end{array}
$$

Predicates must be proven when the constructor *Showable* is used and can be assumed when pattern matching on *Showable*. Operationally, this means that the dictionary for *Show* must be stored in the constructor *Showable*.

EH [Dijkstra, 2005] allows a more direct encoding of existential types without the need for packing and unpacking using a datatype. Instead, a type is closed (packed) by annotating a value with an existential type signature and an existential type is opened (unpacked) when binding an existential type to an identifier. Consider the following EH fragment expressing the above example:

$$
\begin{array}{l}
showable :: (\exists\ a.Show\ a \Rightarrow a) \rightarrow String \\
showable = \lambda x_{a1} \rightarrow show_{p1}\ x \\
test = map\ showable\ [\texttt{'c'}_{p2}, \texttt{"Hello World"}_{p3}, False_{p4}] :: [\exists\ a.Show\ a \Rightarrow a]
\end{array}
$$

The types in the list are hidden by the explicit type signature and opened by binding each element to the variable x in the function *showable*. The constraint set generated for the above example is exactly the same as the one presented for the GHC example. The constraints resulting from these examples can be solved using our framework. Also the type system of EH already checks and propagates existential types. However,

the evidence translation scheme for existentially quantified predicates is not yet described and implemented. Furthermore, the meaning of multi-parameter type classes with both existentially and universally quantified types variables is not yet clear.

Recursive bindings

In this thesis we did not address a minor issue occurring when generating evidence for (mutually) recursive bindings. Consider the following recursive binding:

$$\begin{aligned} &elem\ x\ [\,] &&= False \\ &elem\ x\ (y : ys) &&= x \ \equiv_{p1}\ y \vee elem\ x\ ys \end{aligned}$$

In this function the overloaded operator $(\equiv)$ is used. After analyzing the body of the function *elem* we have to solve the constraint $Prove\ (Eq\ v_1)$ at location *p1* in the abstract-syntax tree. We infer the type $Eq\ a \Rightarrow a \to [a] \to Bool$ for the function *elem* because we cannot simplify the predicate $Eq\ v_1$ further. The result is that the function *elem* expects a dictionary as evidence for using the operator $(\equiv)$. There are two alternatives for generating evidence for an overloaded recursive function: *elem1* and *elem2*.

$$\begin{aligned} &elem1\ dictEq = elem' \\ &\quad \textbf{where}\ elem'\ x\ [\,] &&= False \\ &\qquad\quad elem'\ x\ (y : ys) &&= ((\equiv)\ dictEq)\ x\ y \vee elem'\ x\ ys \\ &elem2\ dictEq\ x\ [\,] &&= False \\ &elem2\ dictEq\ x\ (y : ys) &&= ((\equiv)\ dictEq)\ x\ y \vee elem2\ dictEq\ x\ ys \end{aligned}$$

The first translation yields efficient code because the recursive call does not have to pass the dictionary for *Eq*. A drawback is that the translation scheme is complex, especially for mutually recursive bindings. The second translation is more straightforward, but a problem occurs with this translation. Evidence has to be inserted for the recursive call to *elem2*. However, no *Prove* constraint was generated for this recursive call because the type of *elem* is not yet known when analyzing the body of the function. Therefore, no evidence is generated for the recursive call.

A solution is to extend the predicate language with predicate variables. Those predicate variables relate places where predicates are possibly needed to places where predicate possibly must be inserted. For example, the name of the function *elem* could be used as predicate variable. The additionally generated constraint for the recursive call to *elem* would then be $Prove\ (elem)$. A substitution from *elem* to $\{\,Eq\ v_1\,\}$ is added when the type of the function *elem* is inferred. After applying the substitution on the constraints we still are able to prove the predicate using the framework. Of course using a function identifier as a predicate variable is not the best choice because identifiers possibly shadow each other. We deliberately did not implement a mechanism for predicate substitution in the framework because this problem depends on the evidence translation scheme of the compiler. Furthermore, the framework is not able to solve predicate variables until they are substituted.

Resolution of other qualified types

In this thesis we presented a number of entailment rules together with algorithmic versions of these rules in terms of CHRs. There are many entailment relations described in the literature, such as for extensible records and subtyping [Jones, 1995b]. It would be interesting to examine if other forms of qualified types could be encoded into CHRs and could be resolved with this framework. It is already proven that extensible records can be encoded in terms of type classes with some common extensions such as multi-parameter type classes with functional dependencies [Kiselyov et al., 2004]. Therefore, it should also be possible to directly encode extensible records in the presented framework.

Non-termination

In this thesis we used simplification and propagation CHRs:

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

We assumed the following condition on CHRs:

$$fv\ (H_1, ..., H_i)\ \supseteq\ fv\ (B_1, ..., B_k)$$

The function *fv* computes the variables occurring in the constraints. The variables in the head of a CHR must always be a superset of the variables in the body. Not fulfilling this condition often results in non-termination. For that reason this condition was also dictated by the initial proposal for functional dependencies [Jones, 2000]. These conditions are called the *Coverage Condition* and the *Bound Variable Conditions* in the context of functional dependencies. However, these conditions can safely be relaxed [Duck et al., 2004, Sulzmann et al., 2007]. For example, consider the following class and instance declarations:

class $F\ a$
class $E\ a\ b \mid a \rightarrow b$
instance $E\ a\ b \Rightarrow F\ a$
instance $E\ Int\ Float$

The first instance declaration violates the Coverage Condition because the set $\{a\}$ is not a superset of $\{a, b\}$. However, the CHRs resulting from these declarations are confluent and terminating because the variable b is fixed by a using the functional dependency. The simplification CHR generated for the first instance declaration must introduce a fresh type variable for b:

$$Prove\ (F\ a) \Rightarrow b\ fresh \mid Prove\ (E\ a\ b)$$

Future work is to add support for generating fresh type variables in the body of a CHR.

Furthermore, compilers that allow users to experiment with advanced type class extensions usually fix the number of simplification and improvement steps performed to a certain depth. The reason for this is to avoid non-termination of the compiler. Such a mechanism is not present in the presented framework.

Improving substitutions

In Chapter 9 we have given a short introduction to improvement and explained why improvement is useful. Furthermore, we explained how we extended our framework to support improvement. However, improvement is in some sense an ad-hoc extension of our framework. We did not mention how improvement scales in combination with other extensions such as overlapping instances or local instances. It would be nice to express improvements in the simplification graphs. Improvements are then generated depending on the solution chosen by heuristics. This would be an interesting research topic and we think our framework provides a solid basis for this research.

Bibliography

Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 65–73. ACM Press, June 1993.

Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005a. ACM Press.

Manuel M. T. Chakravarty, Gabriele Keller, Simon L. Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the Symposium on Principles of Programming Languages*, pages 1–13, New York, NY, USA, 2005b. ACM Press.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Universiteit Utrecht, The Netherlands, November 2005.

Atze Dijkstra and S. Doaitse Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Institute of Information and Computing Sciences, Utrecht University, 2005.

Gregory J. Duck, Simon L. Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. In *ESOP '04: Proceedings of the 13th European Symposium on Programming*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5): 467–492, September 2001.

Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.

Thom W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37 (1-3):95–138, October 1998.

Kevin Glynn, Peter J. Stuckey, and Martin Sulzmann. Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*, July 2000.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

Bastiaan J. Heeren and Jurriaan Hage. Type class directives. In Daniel Hermenegildo, Manuel; Cabeza, editor, *PADL '05: Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, pages 253–267. PADL, Springer-Verlag, January 2005.

Bastiaan J. Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *ICFP'03: Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 3 – 13, New York, 2003a. ACM Press.

Bastiaan J. Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *Haskell'03: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 62 – 71, New York, 2003b. ACM Press.

Mark P. Jones. A theory of qualified types. In Bernd Krieg-Bruckner, editor, *ESOP '92: Proceedings of the 4th European Symposium on Programming*, volume 582, pages 287–306. Springer-Verlag, New York, N.Y., 1992.

Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 52–61, New York, 1993. ACM Press.

Mark P. Jones. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation*, 8(3): 229–248, 1995a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b.

Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 160–169, New York, NY, USA, June 1995c. ACM Press.

Mark P. Jones. Typing Haskell in Haskell. In *Haskell'99: Proceedings of the ACM SIGPLAN Workshop on Haskell*, New York, September 1999. ACM Press.

Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming*, volume 1782, pages 230–244, London, UK, March 2000. Springer-Verlag.

Mark P. Jones and Simon L. Peyton Jones. Lightweight extensible records for Haskell. In *Haskell'99: Proceedings of the ACM SIGPLAN Workshop on Haskell*, 1999.

Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, volume 300 of *LNCS*, pages 131–144. Springer, 1988.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell'04: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107. ACM Press, 2004.

Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

Jeffrey R. Lewis, Mark Shields, John Launchbury, and Erik Meijer. Implicit parameters: Dynamic scoping with static types. In *POPL '00: Proceedings of the Symposium on Principles of Programming Languages*, pages 108–118, 2000.

Simon Marlow and Simon L. Peyton Jones. The Glasgow Haskell Compiler (version 6.6). 2006.

Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 135–146, 1995.

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

Simon L. Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell'97: Proceedings of the ACM SIGPLAN Workshop on Haskell*, June 1997.

Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 167–178, New York, USA, 2002. ACM Press.

Martin Sulzmann. Extracting programs from type class proofs. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and Practice of Declarative Programming*, pages 97–108, New York, NY, USA, 2006. ACM Press.

Martin Sulzmann, Gregory J. Duck, Simon L. Peyton Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(1):83–129, January 2007.

Martin Sulzmann and Jeremy Wazny. Implementing overloading in Chameleon, http://taichi.ddns.comp.nus.edu.sg/taichiwiki/ChameleonHomePage, 2001.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL '89: Proceedings of the Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.