Type Class Directives

Bastiaan Heeren and Jurriaan Hage

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{bastiaan,jur}@cs.uu.nl

Abstract. The goal of this paper is to improve the type error messages in the presence of `Haskell 98` type classes, in particular for the non-expert user. As a language feature, type classes are very pervasive, and strongly influence what is reported and when, even in relatively simple programs. We propose four type class directives, and specialized type rules, to lend high-level support to compilers to improve the type error messages. Both have been implemented, and can be used to easily modify the behavior of the type inference process.

Keywords: type error messages, type classes, directives, domain-specific programming

1 Introduction

Improving the type error messages for higher-order, polymorphic, functional programming languages continues to be an area of activity [1–4]. Type classes have been studied thoroughly, both in theory and practice. In spite of this, very little attention has been devoted to compensate the effects type classes have on the quality of type error messages. In this paper, we present a practical and original solution to improve the quality of type error messages by scripting the type inference process.

To illustrate the problem type classes introduce, consider the following attempt to decrement the elements of a list.

```
f xs = map -1 xs
```

The parse tree for this expression does not correspond to what the spacing suggests: the literal `1` is applied to `xs`, the result of which is subtracted from `map`. Notwithstanding, `GHC` will infer the following type for `f`. (`Hugs` will reject `f` because of an illegal `Haskell 98` class constraint in the inferred type.)

```
f :: (Num (t -> (a -> b) -> [a] -> [b]),
      Num ((a -> b) -> [a] -> [b])) => t -> (a -> b) -> [a] -> [b]
```

Both subtraction and the literal `1` are overloaded in `f`'s definition[1]. Although the polymorphic type of `1` is constrained to the type class `Num`, this restriction does

[1] In `Haskell`, we have `1 :: Num a => a` and `(-) :: Num a => a -> a -> a`.

M. Hermenegildo and D. Cabeza (Eds.): PADL 2005, LNCS 3350, pp. 253–267, 2005.

not lead to a type error. Instead, the constraint is propagated into the type of `f`. Moreover, unifications change the constrained type into a type which is unlikely to be a member of `Num`. A compiler cannot reject `f` since the instances required could be given later on. This *open-world* approach for type classes is likely to cause problems at the site where `f` is used. One of our directives allows us to specify that function types will *never* be part of the `Num` type class. With this knowledge we can reject `f` at the site of definition.

In this paper, we will improve the type error messages for `Haskell 98` type classes [5], in particular for the non-expert user. (Extensions to the class system, like multi-parameter type classes, are outside the scope of this paper.) Our approach is to design a language for type inference directives to adapt the type inference process, and, in particular, to influence the error message reporting facility.

For a given module `X.hs`, the directives can be found in a file called `X.type`. If `X.hs` imports a module `Y.hs`, then all its directives are included as well. This applies transitively to the modules that `Y.hs` imports, which we achieve by storing directive information in object files. Our approach is similar to the one followed in an earlier paper [6], in which we did not consider overloading.

The use of compiler directives has a number of advantages: They are not part of the programming language, and can be easily turned off. The directives function as a high-level specification language for parts of the type inference process, precluding the need to know anything about how type inferencing is implemented in the compiler. In tandem with our automatic soundness and sanity checks for each directive, this makes them relatively easy to use. Also, it should be straightforward for other compilers to support our directives as well. Finally, although the focus of this paper is on the type inference process, the compiler directive approach lends itself to other program analyses as well.

This paper makes the following contributions.

1. We present four type class directives to improve the resolution of overloading (Section 2). With these directives we can report special purpose error messages, reject suspicious class contexts, improve inferred types, and disambiguate programs in a precise way.
2. We discuss how the proposed directives can be incorporated into the process of context reduction (Section 3).
3. We give a general language to describe invariants over type classes (Section 4). This language generalizes some of our proposed directives.
4. We extend the specialized type rules [6] to handle type classes (Section 5). As a consequence, we can report precise and tailor-made error messages for incorrect uses of an overloaded function.

The type directives proposed in Section 2 have been implemented in our type inference framework. The `Helium` compiler [7] is based on this framework, and supports the extended specialized type rules.

2 Type Class Directives

In `Haskell`, new type classes are introduced with a *class declaration*. If a list of superclasses is given at this point, then the instances of the type class must also be member of each superclass; this is enforced by the compiler. To make a type a member of a type class, we simply provide an *instance declaration*. Other means for specifying type classes do not exist in `Haskell`.

Therefore, some properties of a type class cannot be described: for example, we cannot exclude a type from a type class. To gain more flexibility, we propose type class directives to enrich the specification of type classes. Each of these have been implemented in our type inference framework.

The first directive we introduce is the `never` directive (Section 2.1), which excludes a single type from a type class. This is the exact opposite of an instance declaration, and limits the *open-world* character of that type class. Similar to this case-by-case directive, we introduce a second directive to disallow new instances for a type class altogether (Section 2.2). A closed type class has the advantage that we know its limited set of instances.

Knowing the set of instances of a type class opens the door for two optimizations. In the exceptional case that a type class is empty, we can reject every function that requires some instance of that class. If the type class `X` has only one member, say the type `t`, then a predicate of the form `X a` can improve `a` to `t`. This is, in fact, an improvement substitution in Jones' theory of qualified types [8]. If we have `(X a, Y a)`, and the set of shared instances is empty or a singleton, then the same reasoning applies. For example, if the instances of `X` are `Int` and `Bool`, and those of `Y` are `Bool` and `Char`, then `a` must be `Bool`. This is easily discovered for `Haskell 98` type classes by taking intersections of sets of instances.

Our next directive, the `disjoint` directive, specifies that the intersection of two type classes should be empty (Section 2.3). This is another instance of an invariant over sets of types, which is formulated by the programmer, and maintained by the compiler. In Section 4, we present a small language to capture this invariant, and many others besides.

Finally, Section 2.4 discusses a `default` directive for type classes, which helps to disambiguate in case overloading cannot be resolved. This directive refines the ad hoc default declarations supported by `Haskell`.

In the remainder of this section, we explore the directives in more detail, and conclude with a short section on error message attributes.

2.1 The `never` Directive

Our first directive lets us formulate explicitly that a type should never become a member of a certain type class. This statement can be accompanied with a special purpose error message, reported in case the forbidden instance is needed to resolve overloading. The main advantage of the `never` directive is the tailor-made error message for a particular case in which overloading cannot be resolved. In addition, the directive guarantees that the outlawed instance will not be given

in future. We illustrate the `never` directive with an example. For the sake of brevity, we keep the error messages in our examples rather terse. Error message attributes, which we will discuss in Section 2.5, can be used to create a more verbose message that depends on the actual program.

```
never Eq (a -> b): functions cannot be tested for equality
never Num Bool: arithmetic on booleans is not supported
```

These two directives should be placed in a `.type` file[2], which is considered prior to type inference, but after collecting all the type classes and instances in scope. Before type inference, we should check the validity of the directives. Each inconsistency between the directives and the instance declarations results in an error message or warning. For example, the following could be reported at this point.

```
The instance declaration for
   Num Bool at (3,1) in A.hs
is in contradiction with the directive
   never Num Bool defined at (1,1) in A.type
```

We proceed with type inference if no inconsistency is found. If arithmetic on booleans results in a `Num Bool` predicate, we report our special purpose error message. For the definition

```
f x = if x then x+1 else x
```

we simply report that arithmetic on booleans is not supported, and highlight the arithmetical operator `+`. An extreme of concision results in the following type error message.

```
(1,19): arithmetic on booleans is not supported
```

The `never` directive is subject to the same restrictions as any instance declaration in `Haskell 98`: a class name followed by a type constructor and a list of unique type variables (we took the liberty of writing function arrow infix in the example presented earlier). `Haskell 98` does not allow overlapping instances, and similarly we prohibit overlapping `never`s. This ensures that there is always at most one directive which we can use for constructing an error message. If we drop this restriction, then it becomes arbitrary which message is generated.

```
never Eq (Int -> a): message #1
never Eq (b -> Bool): message #2
```

In this example, it is unclear what will be reported for the type class predicate `Eq (Int -> Bool)`. One way to cope with this situation is to require a third directive for the overlapping case, namely `never Eq (Int -> Bool)`. This implies that we can always find and report a most specific directive. Note that in the context of overlapping `never` directives, we have to postpone reporting a violating class predicate since more information about a type variable in this assertion may make a more specific directive a better candidate.

[2] Our convention in this paper is to write all type inference directives on a light gray background.

2.2 The `close` Directive

With the `never` directive we can exclude one type from a type class. Similar to this case-by-case directive, we introduce a second type class directive which closes a type class in the sense that no new instances can be defined. As a result of this directive, we can report special error messages for unresolved overloading for a particular type class. A second advantage is that the compiler can assume to know all instances of the given type class since new instances are prohibited, which can be exploited when generating the error message.

One subtle issue is to establish at which point the type class should be closed. This can be either *before* or *after* having considered the instance declarations defined in the module. In this section we discuss only the former. A possible use for the latter is to close the `Num` type class in `Prelude.type` so that everybody who imports it may not extend the type class, but the `Prelude` module itself may specify new instances for `Num`.

Before we start with type inference, we check for each closed type class that no new instance declarations are provided. A special purpose error message is attached to each `close` directive, which is reported if we require a non-instance type to resolve overloading for the closed type class. Such a directive can live side by side with a `never` directive. Since the latter is strictly more informative, we give it precedence over a `close` directive if we have to create a message. As an example, we close the type class for `Integral` types, defined at the standard Prelude. Hence, this type class will only have `Int` and `Integer` as its members.

```
close Integral: the only instances of Integral are Int and Integer
```

The main advantage of a closed type class is that we know the fixed set of instances. Using this knowledge, we can influence the type inference process. As discussed in the introduction to Section 2, we can reject definitions early on (in case the set of instances for a certain type class is empty) or improve a type variable to a certain type (in case the set of instances is a singleton).

For example, consider a function `f :: (Bounded a, Num a) => a -> a`. The type class `Bounded` contains all types that have a minimal and maximal value, including `Int` and `Char`. However, `Int` is the only numeric type among these. Hence, if both `Bounded` and `Num` are closed, then we may safely improve `f`'s type to `Int -> Int`.

The advantages of the `close` directive would be even higher if we drop the restrictions of `Haskell 98`, because this directive allows us to reject incorrect usage of a type class early on. We illustrate this with the following example.

```
class Similar a where
  (~=) :: a -> a -> Bool

instance Similar Int where
  (~=) = (==)
```

Assume that the previous code is imported in a module that closes the type class `Similar`.

```
close Similar: the only instance of Similar is Int.
```

```
f x xs = [x] ~= xs
```

GHC version 6.2 (without extensions) accepts the program above, although an instance for `Similar [a]` must be provided to resolve overloading. The type inferred for `f` is

```
f :: forall t. (Similar [t]) => t -> t -> Bool
```

although this type cannot be declared in a type signature for `f`[3]. This type makes sense: the function `f` can be used in a different module, on the condition that the missing instance declaration is provided. However, if we intentionally close the type class, then we can generate an error for `f` at this point.

In this light, the `close` directive may become a way to moderate the power of some of the language extensions by specifying cases where such generality is not desired. An alternative would be to take `Haskell 98` as the starting point, and devise type class directives to selectively overrule some of the language restrictions. For instance, a directive such as `general X` could tell the compiler not to complain about predicates concerning the type class `X` that cannot be reduced to head-normal form. Such a directive would allow more programs. In conclusion, type class directives give an easy and flexible way to specify these local extensions and restrictions.

2.3 The `disjoint` Directive

Our next directive deliberately reduces the set of accepted programs. In other words: the programs will be subjected to a stricter type discipline. The `disjoint` directive specifies that the instances of two type classes are disjoint, i.e., no type is shared by the two classes. A typical example of two type classes that are intentionally disjoint are `Integral` and `Fractional` (see the `Haskell 98` Report [5]). If we end up with a type `(Fractional a, Integral a) => ....` after reduction, then we can immediately generate an error message, which can also explain that "fractions" are necessarily distinct from "integers". Note that without this directive, a context containing these two class assertions is happily accepted by the compiler, although it undoubtedly results in problems when we try to use this function. Acknowledging the senselessness of such a type prevents misunderstanding in the future. A `disjoint` directive can be defined as follows.

```
disjoint Integral Fractional:
   something which is fractional can never be integral
```

Because `Floating` is a subclass of `Fractional` (each type in the former must also be present in the latter), the directive above implies that the type classes `Integral` and `Floating` are also disjoint.

[3] In our opinion, it should be possible to include each type inferred by the compiler in the program. In this particular case, `GHC` suggests to use the Glasgow extensions, although these extensions are not required to infer the type.

In determining that two type classes are disjoint, we base our judgements on the set of instance declarations for these classes, and not on the types implied by the instances. Therefore, we reject instance declarations `C a => C [a]` and `D b => D [b]` if `C` and `D` must be disjoint. A more liberal approach is to consider the set of instance types for `C` and `D`, so that their disjointness depends on other instances given for these type classes.

Take a look at the following example which mixes fractions and integrals.

```
wrong = div 3 8 + 1/2
```

The `disjoint` directive helps to report an appropriate error message for the definition of `wrong`. In fact, without this directive we end up with the type `(Integral a, Fractional a) => a`. GHC reports an ambiguous type variable as a result of the monomorphism restriction.

```
Disjoint.hs:1:
 Ambiguous type variable ‘a’ in these top-level constraints:
   ‘Integral a’ arising from use of ‘div’ at Disjoint.hs:1
   ‘Fractional a’ arising from use of ‘/’ at Disjoint.hs:1
 Possible cause: the monomorphism restriction applied
                 to the following:
   wrong :: a (bound at Disjoint.hs:1)
 Probable fix: give these definition(s) an explicit type
               signature
```

Ironically, it is the combination of the two class predicates that makes the defaulting mechanism fail (no numeric type is instance of both classes), which in turn activates the monomorphism restriction rule.

2.4 The `default` Directive

One annoying aspect of overloading is that seemingly innocent programs are in fact ambiguous. For example, `show []` is not well-defined, since the type of the elements must be known (and showable) in order to display the empty list. This problem can only be circumvented by an explicit type annotation. A default declaration is included as special syntax in `Haskell` to help disambiguate overloaded numeric operations. This approach is fairly ad hoc, since it only covers the (standard) numeric type classes. Our example suggests that a programmer could also benefit from a more liberal defaulting strategy, which extends to other type classes. Secondly, the exact rules when defaulting should be applied are unnecessarily complicated (see the Haskell Report [5] for the exact specification). We think that a `default` declaration is nothing but a type class directive, and that it should be placed amongst the other directives instead of being considered part of the programming language. Taking this viewpoint paves the way for other, more complex defaulting strategies as well.

One might wonder at this point why the original design is so conservative. Actually, the caution to apply a general defaulting strategy is justified since it

changes the semantics of a program. Inappropriate defaulting unnoticed by a programmer is unquestionably harmful. By specifying `default` directives, the user has full control over the defaulting mechanism. A warning should be raised to inform the programmer that a class predicate has been defaulted. Although we do not advocate defaulting in large programming projects, it is unquestionably useful at times, for instance, to show the result of an evaluated expression in an interpreter. Note that `GHC` departs from the standard, and applies a more liberal defaulting strategy in combination with the emission of warnings, which works fine in practice.

Take a look at the following datatype definition for a binary tree with values of type `a`.

```
data Tree a = Bin (Tree a) a (Tree a) | Leaf   deriving Show
```

A function to show such a tree can be derived automatically, but it requires a show function for the values stored in the tree. This brings us to the problem: `show Leaf` is of type `String`, but it is ambiguous since the tree that we want to display is polymorphic in the values it contains. We define default directives to remedy this problem.

```
default Num (Int, Integer, Float, Double)
default Show ((), String, Bool, Int)
```

The first directive is similar to the original default declaration, the second defaults predicates concerning the `Show` type class. Obviously, the types which we use as default for a type class must be a member of the class.

Defaulting works as follows. For a given type variable `a`, let `P` = $\{$`X`$_1$ `a, X`$_2$ `a, ..., X`$_n$ `a`$\}$ be the set of all predicates in the context which contain `a`. Note that only these predicates determine which type may be selected for `a` as a default, and that other predicates in the context are not influenced by this choice. If at least one of the `X`$_i$ has a default directive, then we consider the default directives for each of the predicates in P in turn (if they exist). For each of these default directives, we determine the first type which satisfies all of `P`. If this type is the same for all default directives of `P`, then we choose this type for `a`. If the default directives cannot agree on their first choice, then defaulting does not take place.

If default directives are given for a type class and for its subclass, we should check that the two directives are coherent. For instance, `Integral` is a subclass of `Num`, and hence we expect that defaulting `Integral a` and `Num a` has the same result as defaulting only `Integral a`.

Considering defaulting as a directive allows us to design more precise defaulting strategies. For instance, we could have a specific default strategy for showing values of type `Tree a`: this requires some extra information about the instantiated type of the overloaded function `show`.

2.5 Error Message Attributes

The error messages given so far are context-insensitive, but for a real implementation this is not sufficient. Therefore, we use error message attributes, which

may hold context dependent information. For example, location information is present in an attribute `@range@` (attributes are written between `@` signs). Due to space limitations we restrict ourselves to an example for the `close` directive.

```
close Show:
   The expression @expr.pp@ at @expr.range@ has the type @expr.gentype@.
   This type is responsible for the introduction of the class predicate
   @errorpredicate@, which is not an instance of @typeclass@ due to
   the close directive defined at @directive.range@.
```

The attributes in the error message are replaced by information from the actual program. For instance, `@directive.range@` is changed into the location where the `close` directive is defined, and `@expr.pp@` is unfolded to a pretty printed version of the expression responsible for the introduction of the erroneous predicate. We can devise a list of attributes for each directive. These lists differ: in case of the `disjoint` directive, for instance, we want to refer to the origin of both class predicates that contradict.

A complicating factor is that the predicate at fault may not be the predicate which was introduced. Reducing the predicate `Eq [(String, Int -> Int)]` will eventually lead to `Eq (Int -> Int)`. We would like to communicate this reasoning to the programmer as well, perhaps by showing some of the reduction steps.

3 Implementation

Type inference for `Haskell` is based on the Hindley-Milner [9] type system. Along the way, types are unified, and when unification fails, an error message is reported. An alternative method is to collect *equality constraints* to encapsulate the relation between various types of parts of the program, and solve these afterwards. Type inference for `Haskell 98` is widely studied and well understood: we suggest "Typing Haskell in Haskell" [10] for an in-depth study. To support overloading, we extend the Hindley-Milner system and propagate sets of type class predicates. For each binding group we perform *context reduction*, which serves to simplify sets of type class predicates, and to report predicates that cannot be resolved. Context reduction can be divided into three phases.

In the first phase, the predicates are simplified by using the instance declarations until they are in head-normal form, that is, of the form `X (a` $\mathtt{t}_1 \ldots \mathtt{t}_n$`)` where `a` is a type variable. Typically, we get predicates where n is zero. Predicates that cannot be reduced to this form are reported as incorrect. For instance, `Eq [a]` can be simplified to `Eq a`, the predicate `Eq Int` can be removed altogether, and an error message is created for `Num Bool`.

Duplicate predicates are removed in the second phase, and we use the type class hierarchy to eliminate predicates entailed by assertions about a subclass. For instance, `Eq` is the superclass of `Ord`, and, hence, `Ord a` implies `Eq a`. If we have both predicates, then `Eq a` can be safely discarded.

In the final phase, we report predicates that give rise to an ambiguous type. For instance, the type `(Read a, Show a) => String -> String`, inferred for

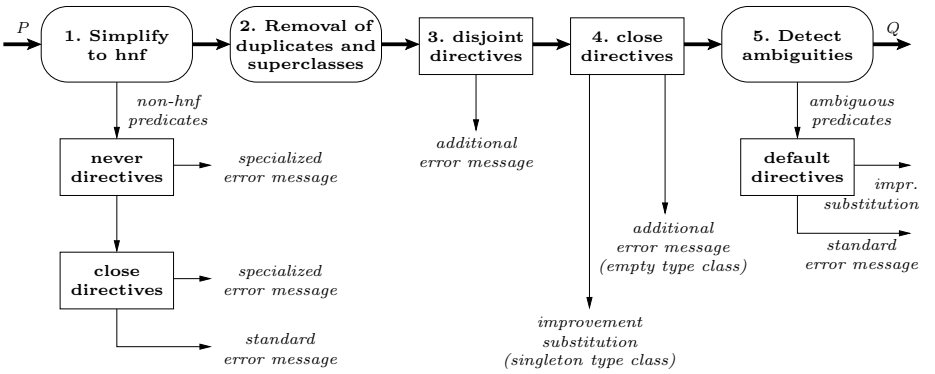

Fig. 1. Context reduction with type class directives for `Haskell 98`

the famous `show . read` example, is ambiguous since there is no way we can determine the type of `a`, which is needed to resolve overloading. Note that we can postpone dealing with predicates containing monomorphic type variables.

We continue with a discussion on how the four type class directives can be incorporated into context reduction. Figure 1 gives an overview. The first, second, and fifth step correspond to the three phases of the traditional approach. The thickened horizontal line reflects the main process in which the set of predicates P is transformed into a set of predicates Q.

The first modification concerns the predicates that cannot be simplified to head-normal form. If a `never` or `close` directive is specified for such a predicate, then we report the specialized error message that was declared with the directive. Otherwise, we proceed as usual and report a standard error message.

The `disjoint` directives and closed type classes are handled after removal of duplicates and super-classes. At this point, the predicates to consider are in head-normal form. A `disjoint` directive creates an error message for a pair of predicates that is in conflict. Similarly, if we can conclude from the closed type classes that no type meets all the requirements imposed by the predicates for a given type variable, then an error message is constructed. If we, on the other hand, discover that there is a single type which meets the restrictions, then we assume this type variable to be that particular type. This is an improvement substitution [8]. Because we consider all predicates involving a certain type variable at once, the restrictions of `Haskell 98` guarantee that improvement substitutions cannot lead to more reduction steps.

Finally, we try to avoid reporting ambiguous predicates by inspecting the given `default` directives, as described in Section 2.4. Defaulting a type variable by applying these directives results again in an improvement substitution.

The use of improvement substitutions leads to more programs being accepted, while others are now rejected. The sum of their effects can be hard to predict, and not something to rely on in large programming projects. Even without improvement substitutions, the `never`, `close`, and `disjoint` directives can be quite useful.

4 Generalization of Directives

In this section, we sketch a generalization of the first three directives described in Section 2. This part has not been implemented, but gives an idea how far we expect type class directives can go, and what benefits accrue.

Essentially, a type class describes a (possibly infinite) set of types, and most of the proposed directives can be understood as constraints over such sets. In fact, they describe invariants on these sets of types, enriching the means of specification in Haskell, which is limited to membership of a type class (instance declaration), and a subset relation between type classes (class declaration).

We present a small language to specify invariants on the class system. The language is very expressive, and it may be necessary to restrict its power for reasons of efficiency and decidability, depending on the type (class) system to which it is added.

Constraint	::=	*Type EltOp Set* | *Set SetOp Set*
Set	::=	*BinOp Set Set* | *SetLiteral* | *Class*
SetLiteral	::=	{} | { *Type* (, *Type*)* }
EltOp	::=	`isin` | `isnotin`
SetOp	::=	<= | == | >=
BinOp	::=	`intersect` | `union` | `difference`

Each constraint can be followed by an error message. If necessary, syntactic sugar can be introduced for special directives such as `never` and `disjoint`.

```
Monad  ==  {Maybe, [], IO}: only Maybe, [], and IO are monads today.
Read   ==  Show
intersect Egglayer Mammal  <=  {Platypus}
```

The first example directive prevents new instances for the `Monad` class, while `Read == Show` demands that in this module (and all modules that import it) the instances for `Show` and `Read` are the same. A nice example of an invariant is the third directive, which states that only the duckbilled platypus can be both in the type class for egg layers and in `Mammal`. This directive might be used to obtain an improvement substitution (as discussed in Section 2): if we have the predicates `Mammal a` and `Egglayer a`, then `a` must be `Platypus`. This example shows that the directives can be used to describe domain specific invariants over class hierarchies.

5 Specialized Type Rules

In an earlier paper [6], we introduced specialized type rules to improve type error messages. The main benefits of this facility are that for certain collections of expressions, the programmer can

1. change the order in which unifications are performed, and
2. provide special type error messages, which can exploit this knowledge,
3. with the guarantee that the underlying type system is unchanged.

This facility is especially useful for domain specific extensions to a base language (such as `Haskell`), because the developer of such a language can now specify error messages which refer to concepts in the domain to replace the error messages phrased in terms of the underlying language. We present an extension of these type rules which allows class assertions among the equality constraints to deal with overloading. This extension has been implemented in the `Helium` compiler [7].

Consider the function `spread`, which returns the difference between the smallest and largest value of a list, and a specialized type rule for this function, given that it is applied to one argument.

```
spread :: (Ord a, Num a) => [a] -> a
spread xs = maximum xs - minimum xs
```

```
    xs :: t1;
--------------------
  spread xs :: t2;

t1 == [t3]: @xs.pp@ must be a list
t3 == t2: @expr.pp@ should return a value of type @t3@
Eq t2: @t2@ is not an instance of Eq, let alone Ord or Num
Ord t2: @t2@ should have a linear ordering imposed on it
Num t2: @t2@ should allow numerical operations
```

A specialized type rule consists of a deduction rule, followed by a list of constraints. In the consequent of the deduction rule, `spread xs :: t2`, we describe the expressions of interest. Since `xs` also occurs above the line, it is considered to be a meta-variable which functions as a placeholder for an arbitrary expression (with a type to which we can refer as `t1`).

The deduction rule is followed by a number of constraints. The first of these states that the type `t1` is a list type, with elements of type `t3` (`t3` is still unconstrained at this point). The next equality constraint constrains the type `t3` to be the same as the type of `spread xs`. Note that the listed constraints are verified from top to bottom, and this fact can be exploited to yield very precise error messages.

For class assertions we can also exploit the order of specification. Although membership of `Ord` or `Num` implies membership of `Eq`, we can check the latter first, and give a more precise error message in case it fails. Only when `Eq t2` holds, do we consider the class assertions `Ord t2` and `Num t2`. Note that the assertion `Eq t2` does not change the validity of the rule.

Context reduction takes place after having solved the unification constraints. This implies that listing class assertions before the unification constraints makes little sense, and only serves to confuse people. Therefore, we disallow this.

Equality constraints can be moved into the deduction rule, in which case it is given a standard error message. This facility is essential for practical reasons: it should be possible to only list those constraints for which we expect special treatment. Similarly, we may move a class assertion into the deduction rule. Notwithstanding, this assertion is checked *after* all the unification constraints.

All specialized type rules are automatically examined in that they leave the underlying type system unchanged. This is an essential feature, since a mistake is easily made in these rules. We compare the set of constraints implied by the specialized type rule (say S) with the set that would have been generated by the standard inference rules (say T). A type rule is only accepted if S equals T under an entailment relation. This relation is a combination of entailment for class predicates and for equality constraints.

6 Related Work

A number of papers address the problem of improving the type error messages produced by compilers for functional programming languages. Several approaches to improve on the quality of error messages have been suggested. One of the first proposals is by Wand [11], who suggests to modify the unification algorithm such that it keeps track of reasons for deductions about the types of type variables. Many papers that followed elaborate on his idea. At the same time, Walz and Johnson [12] suggested to use maximum flow techniques to isolate and report the most likely source of an inconsistency. This can be considered the first heuristic-based system.

Recently, Yang, Michaelson, and Trinder [2] have reported on a human-like type inference algorithm, which mimics the manner in which an expert would explain a type inconsistency. This algorithm produces explanations in plain English for inferred (polymorphic) types. McAdam [1] suggested to use unification of types modulo linear isomorphism to automatically repair ill-typed programs. Haack and Wells [3] compute a minimal set of program locations (a type error slice) that contribute to a type inconsistency. The `Chameleon` type debugger is developed by Stuckey, Sulzmann, and Wazny [4], and helps to locate type errors in an interactive way.

Elements of our work can be found in earlier papers: closed type classes were mentioned by Shields and Peyton Jones [13], while the concepts of disjoint type classes and type class complements were considered by Glynn et al. [14]. Type class directives lead to improvement substitutions which are part of the framework as laid down by Jones [8]. All these efforts are focused on the type system, while we concentrate on giving good feedback by adding high-level support to compilers via compiler directives. Moreover, we generalize these directives to invariants over type classes.

A different approach to tackle language extensions is followed in the `DrScheme` project [15], which introduces language levels (syntactically restricted variants) to gradually become familiar with a language.

7 Conclusion and Future Work

This paper offers a solution to compensate the effect that the introduction of overloading (type classes) has on the quality of reported error messages. In general, the types of overloaded functions are less restrictive, and therefore some

errors may remain undetected. At the same time, a different kind of error message is produced for unresolved overloading, and these errors are often hard to interpret.

To remedy the loss of clarity in error messages, a number of type class directives have been proposed, and we have indicated how context reduction can be extended to incorporate these directives. The directives have the following advantages.

- Tailor-made, domain-specific error messages can be reported for special cases.
- Functions for which we infer a type scheme with a suspicious class context can be detected (and rejected) at an early stage.
- An effective defaulting mechanism assists to disambiguate overloading.
- Type classes with a limited set of instances help to improve and simplify types.

Furthermore, we have added type class predicates to the specialized type rules, and the soundness check has been generalized accordingly.

We see several possible directions for future research. A small language to specify invariants on the class system (see Section 4) seems to be a promising direction, and this requires further investigation. The more expressive such a language becomes, the more need there is for some form of analysis of these invariants. Another direction is to explore directives for a number of the proposed extensions to the type class system [16], and to come up with new directives to alleviate the problems introduced by these extensions.

In both these cases, we see the need for a formal approach, so that the effects of our (more general) directives on our (extended) language can be fully understood. The constraint handling rules (used in [14]) are a good starting point for such an approach.

Furthermore, we would like to lift our ideas on directives to Jones' theory of qualified types [8]. As a result, we want to look for directives to support other *qualifiers* that fit in his framework. Besides the type class predicates discussed in this paper, we plan to investigate predicates for extensible records, and for subtyping.

Acknowledgements

We like to thank Daan Leijen, Alexey Rodriguez Yakushev, Doaitse Swierstra, and the anonymous reviewers for their suggestions and comments on this paper.

References

1. McAdam, B.: How to repair type errors automatically. In: 3rd Scottish Workshop on Functional Programming, Stirling, U.K. (2001) 121–135
2. Yang, J., Michaelson, G., Trinder, P.: Explaining polymorphic types. The Computer Journal **45** (2002) 436–452

3. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. In: Proceedings of the 12th European Symposium on Programming. (2003) 284–301
4. Stuckey, P., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: Haskell Workshop, New York, ACM Press (2003) 72 – 83
5. Peyton Jones, S., ed.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003) http://www.haskell.org/onlinereport/.
6. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Eighth ACM Sigplan International Conference on Functional Programming, New York, ACM Press (2003) 3 – 13
7. Heeren, B., Leijen, D., van IJzendoorn, A.: Helium, for learning Haskell. In: ACM Sigplan 2003 Haskell Workshop, New York, ACM Press (2003) 62 – 71 http://www.cs.uu.nl/helium.
8. Jones, M.P.: Simplifying and improving qualified types. In: International Conference on Functional Programming Languages and Computer Architecture. (1995) 160–169
9. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
10. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop. (1999)
11. Wand, M.: Finding the source of type errors. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, FL (1986) 38–43
12. Walz, J.A., Johnson, G.F.: A maximum flow approach to anomaly isolation in unification-based incremental type inference. In: Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, FL (1986) 44–57
13. Shields, M., Peyton Jones, S.: Object-oriented style overloading for Haskell. In: Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01). (2001)
14. Glynn, K., Stuckey, P., Sulzmann, M.: Type classes and constraint handling rules. In: First Workshop on Rule-Based Constraint Reasoning and Programming. (2000)
15. Findler, R.B., Clements, J., Cormac Flanagan, M.F., Krishnamurthi, S., Steckler, P., Felleisen, M.: DrScheme: A programming environment for Scheme. Journal of Functional Programming **12** (2002) 159–182
16. Peyton Jones, S., Jones, M., Meijer, E.: Type classes: an exploration of the design space. In: Haskell Workshop. (1997)