A first attempt at type class directives

Bastiaan Heeren

Jurriaan Hage

institute of information and computing sciences, utrecht university

technical report UU-CS-2004-039

www.cs.uu.nl

A first attempt at type class directives

Bastiaan Heeren 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

Building on earlier work on type inference directives for scripting a compiler to improve type error messages, we present extensions to those directives to deal with type classes. Our work is mainly motivated by the need for better type error messages, especially for domain specific languages. Type inference directives can bridge the gap between embedded domain specific languages and `Haskell` by their ability to lift error messages to the conceptual level of the domain, without a need to know anything about how the compiler works on the inside.

We consider both special type class directives, which help to improve type error messages in the presence of type classes, and we show how existing type inference directives can be extended to cope with overloading. We also describe a heuristic where type class information is used to pinpoint more precisely the most likely source of a unification error in a program.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Applicative (Functional) Programming; D.3.4 [**Programming Languages**]: Processors—*debuggers*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms

Languages

Keywords

constraints, type inference, type classes, type errors, 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. In a previous paper on the subject [HHS03b], we described a number of methods to improve the type error messages reported by the `Helium` compiler [HLvI03] (which implements a significant subset of the `Haskell 98` language) by the introduction of type inference directives which are placed in special files to accompany the usual `Haskell` source code.

The main advantage of the type inference directives is that people, and especially developers of (combinator) libraries in use by others, can tune the type inference process to their liking without needing to know anything about the compiler's internals: all modifications to the type inference process are described at the level of type inference rules, and are given in separate files. For example, we can define the following specialized type rule for applications of the `map` function from the Prelude[1].

```
  f :: t1;   xs :: t2;
--------------------------
    map f xs :: t3;

t2 == [t4]:       map needs @xs.pp@ to be a list
t1 == t5 -> t6: first argument to map should be a function
t4 == t5:         the function @f@ cannot be applied to
                  elements of @xs.pp@
t3 == [t6]:       result type should be a list of @t6@
```

The order in which the equality constraints are listed in the specialized type rule corresponds to the order of type inference. This reordering strongly determines which error message is reported for an ill-typed expression. Note that this also helps giving more precise, domain-specific messages. For instance, in the third error message we can use the fact that we already know `xs` to be a list and `f` to be a function. Needless to say, the lack of domain specificity in `map` avoids doing very interesting things, but it should suffice to illustrate the concept.

A second advantage of our setup is that specialized type rules cannot change the underlying type system in the sense that the same set of programs is accepted as type correct (and with the same types). User-defined type rules that do not correspond with the built-in type system are rejected.

In our previous work, both the type system and the implementation could not cope with type classes in any way. This is a large draw-

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

back since most Haskell programs use the predefined type classes Eq, Show etc., and some more advanced combinator languages define and use their own type classes. Since the type inference directives were designed especially for the purpose of improving the error messages in the presence of combinator languages, this is a serious inconvenience. Therefore, it seems natural to want to extend our work to encompass type classes as well.

However, the introduction of type classes has some serious consequences for the quality of the reported error messages. Ad-hoc polymorphism increases the problem to report concise error messages significantly, and not much attention has been devoted to this topic so far. A simple observation is that less errors will be exposed because the types become more general. This is illustrated by the following three well-typed[2] definitions.

```
f1 xs = map -1 xs
f2 xs = map (-1) xs
f3 xs = map (\x -> x-1) xs
```

Here, subtraction (f1 and f3), negation (f2), as well as the integer literal, are overloaded. The class assertions that appear in the inferred types for f1 and f2 will never be met, likely causing problems at the site where these functions are used.

Our work in this paper takes two forms: first of all, we have devised a number of type class directives to improve type error messages in the presence of type classes. An illustrative example is 7 + True, for which Hugs mentions “unresolved top-level overloading” and GHC (version 6.2) that it finds no instance for Num Bool. Such an error message can easily confuse a novice functional programmer, who might not even have heard of type classes, instances and overloading. For these programmers at least, one would like to be able to give the compiler some extra information so that it can come up with better error messages. A directive that specifies that Bool will never be a member of the type class Num can help the compiler to give more precise error messages, i.e., saying that a value of type Bool may not be used in an addition. In this paper, our main focus will be type classes according to the Haskell 98 specification, without considering extensions such as overlapping instances, multi-parameter type classes, and functional dependencies, although this would be an appealing direction for future research.

Second, we investigate how information about type classes can help improve the type error messages for unification errors. For example, suppose that function types cannot be part of the Eq type class. When a programmer writes (\x -> x) == "identity", then this could result in an error message which puts all the blame on the left operand of the equality, since the type of the right operand is a member of Eq. Although we shall refrain from doing so in this paper, one could go further and construct a heuristic which, based on the expected type String and the inferred type for the lambda abstraction, returns a hint how to modify the left operand such that it fits in this context.

The paper is structured as follows. We start with the introduction of various type class directives in Section 2. In Section 3, we consider extensions to earlier defined heuristics which make use of type class information. In the following sections, we describe our type inferencing framework and show where the various facilities discussed in this paper fit into the framework. We end with a discussion of related work, a summary of our results, and suggestions for future work.

[2]GHC accepts these definitions without any warning. Hugs, on the other hand, reports an illegal Haskell 98 class constraint, when started in Haskell 98 mode. The monomorphism restriction will reject the first two definitions if map is only partially applied.

2 Motivating examples

In this section we consider each of the directives we have devised in turn. We stay at the level of examples, and show how these directives can be used to improve the accuracy of type error messages. We finally discuss possible interactions between the directives, and how a compiler can use the directives to modify its behavior in their presence.

2.1 The never directive

The simplest form of a type class directive is the never directive, which excludes a type from becoming a member of a type class. The benefit of this directive is twofold. During the normalization of class assertions we can reject certain assertions, and immediately generate an error message. Assertions that make no sense can no longer show up in types inferred by the compiler. The second advantage is that we can supply an error message for a directive, which is reported if the type class directive applies. We illustrate the never directive with an example. For the sake of brevity, we keep the error messages in our examples rather terse.

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

```
wrong = if id == id then 3 else 2 + True
```

The overloading in the definition of wrong cannot be resolved at two locations. In particular, instances for Eq (a -> b) and Num Bool are missing. In fact, this is the content of the message given by the Hugs interpreter.

```
ERROR "Ex1.hs":1 - Unresolved top-level overloading
*** Binding             : wrong
*** Outstanding context : (Eq (b -> b), Num Bool)
```

The error message emitted by GHC 6.2 is more detailed.

```
Ex1.hs:1:
    No instances for (Num Bool, Eq (a -> a))
      arising from the literal ‘3’ at Ex1.hs:1
    In the definition of ‘wrong’:
        wrong = if id == id then 3 else 2 + True

Ex1.hs:1:
    No instances for (Num Bool, Eq (a -> a))
      arising from use of ‘==’ at Ex1.hs:1
    In the predicate expression: id == id
    In the definition of ‘wrong’:
        wrong = if id == id then 3 else 2 + True
```

This message is definitely informative, although it is misleading that in both messages the two irreducible class assertions are mentioned. The two special purpose error messages would have been reported if we had taken the never directives into account.

The never directive is subjected 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). Haskell 98 does not allow overlapping instances, and similarly we prohibit overlapping nevers.

The main reason is that when we generate an error message, we want to explain which directive was responsible and give the corresponding error message. For example, if overlapping is allowed we do not know which message to generate for `wrong` in the following example.

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

wrong = (even == even)
```

The example illustrates that in this case we need some kind of prioritizing system to be added to the directives do determine which message is reported. Of course, this can be simply the order in which they are specified. Another way to cope with this situation is that we require a third directive, namely `never Eq (Int -> Bool)`, which is the overlapping case. This assures us that we can always find a most specific directive. Note that we have to postpone reporting a violating class assertion in the context of overlapping `never` directives since more information about a type variable in this assertion may make a more specific directive a better candidate.

2.2 The `close` directive

The second type class directive we consider is the `close` directive. It is stronger than the previous `never` directive, in that it forbids any new instance for a certain type class. To be more precise: if a class assertion for a closed type class cannot be reduced, then this will not be possible later on as a consequence of a new instance declaration that is provided. Many of the considerations applying to `never` also apply to `close`. An advantage of `close` is that the compiler can assume to know all instances of the given type class. Similar to the case-by-case directive, this allows the compiler to decide early on that some class assertion is in error.

The compiler should warn the programmer when a new instance is defined for a closed type class. Because the set of instances is known and fixed after the directive is encountered, a list of all possible instances which do exist can be presented. The advantage of `never` would be much higher if the `Haskell 98` restrictions on the context of a type are somewhat relaxed, as shown by the following example.

```
close Similar:
   the instances of Similar are @insts@.

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

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

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

Note that `GHC` (version 6.2, without extensions) accepts the program above, because it allows more general contexts in case the types are inferred (although not if they are explicitly given). In the example, the inferred type for `f` is `f :: forall a. (Similar [a]) => a -> [a] -> Bool`. To actually use this function in a different module, an instance `Similar [a]` still has to be provided. The `close` directive tells us, however, that this cannot be the case, so we can generate an error for `f` at this point.

In this fashion, 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 allowed, and benefiting from tailor made error message in the process. One could also choose to keep `Haskell 98` as a starting point, but to devise type class directives which allow some of the language restrictions to be selectively overruled. For instance, a directive such as `general X` could tell the compiler not to complain about certain class assertions that cannot be reduced to normal form. This would be a situation where the set of allowable programs grows due to a type directive. Which manner is chosen, depends solely on one's preference. The main lesson to learn is that type class directives give an easy and flexible way to specify these "local" extensions or restrictions.

We conclude our discussion of this directive by a more extensive example which shows in detail the error message attributes which we can use in a type error message that is associated with a `close` directive. These attributes, like `@insts@` in the previous example, help us to show context dependent information. Similar attributes can be defined for the other directives. Take a look at the following incorrect definition.

```
main = print (const 3)
```

When we generate an error message for an expression based on information from a `close` directive, we would like to have the following information available.

- The name of the overloaded function, in this example `print`, and its type scheme, here `Show a => a -> IO ()`. Furthermore, we may also want to refer to the instantiated type of this specific occurrence, which is `Show (v1 -> v2) => (v1 -> v2) -> IO ()`.
- The type which should be a member of the `Show` type class, in this case `v1 -> v2`. Note that this type contains the same (internal) type variables as the instantiated type.
- The definition for which context reduction was taking place, which is `main = print (const 3)`.
- For all of these, we want their location (file name and line-column number), including the location of the `close` directive.

With this information we can write down the following type error message template.

```
close Show:
   Due to the close at @dir.loc@, the instances of
   Show are @insts@. In @expr.pp@, the identifier
   @id@ :: @id.gentype@ is responsible for the
   introduction of the assertion ...
```

A more advanced facility would also describe reductions which have already taken place before the error was discovered. For instance, if we have the class assertion `Eq [(String, Int -> Int)]`, then we get into trouble after we have reduced it to `Eq (Int -> Int)`. At this point, we would like to communicate this reasoning to the programmer as well, perhaps by showing some of the reduction steps.

2.3 The `disjoint` directive

The `disjoint` directive specifies that the instances of two type classes are disjoint, i.e., that no type in a certain class is shared with a certain other class. A typical example of two type classes that are intentionally disjoint are `Integral` and `Fractional`. If, after reduction, we end up with a type

`(Fractional a, Integral a) => ....`, 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 at a later stage. A programmer should be warned if new instance declarations make that a type becomes an inhabitant of both classes. Take a look at the following example which mixes fractions and integrals.

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

```
wrong = div 2 3 + 2/43
```

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`. Unfortunately, `GHC` reports an ambiguous type variable as a result of the monomorphism restriction. `Hugs` produces a similar error message.

```
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 class assertions about `Integral` and `Fractional` that make the defaulting mechanism fail (there is no numeral type instance of both classes), which in turn activates the monomorphism restriction rule.

2.4 The `default` directive

A default declaration is already included as special syntax in `Haskell 98`. This defaulting helps to disambiguate overloaded numeric operations (see Section 4.3.4 of the Revised `Haskell 98` Report[Has03]). A programmer can specify an ordering on numeric types in a default declaration (one per module). In this way we can specify that the overloaded literal constant `2` gets type `Int` or `Integer`, depending on our preference. 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. In fact, we could drop the restriction that we can only default to numeric types, and we could allow other (more complex) defaulting strategies as well.

2.5 A possible generalization

The previous directives are all instances of a more general approach, which considers a type class to be a set of types (its instances). Over this we may specify all kinds of predicates with some added (comparison) operators on sets such as equality, subset, union, intersection, and difference.

Even if a complete language is included for specifying type class directives to which all the other directives can be mapped (excluding possibly the `close` and `default` directive), then we expect to continue supporting sugared notation for often used directives, both from the viewpoint of user-friendliness as well the hope that they might be amenable to more efficient implementations.

We close this section with a small selection of interesting examples, which show what could be gained by extending the language.

```
Monad = {Maybe, [], IO}
intersect Integral Fractional = {}
intersect X Y <= union A B
RealFloat = intersect RealFrac Floating
intersect Eq Ord = Ord
Read = Show
```

The first directive prevents new instances for the `Monad` class, while the second is equivalent to `disjoint Integral Fractional`. The third expresses that all instances in the intersection of `X` and `Y` are in either `A` or `B` (or both). `RealFloat` is defined as the intersection of two other type classes. The directive `intersect Eq Ord = Ord` could also have been written as `Ord <= Eq`, while finally we demand that `Read` and `Show` instance declarations come in pairs. Note that a directive such as `Read = Show` demands that in this module (and all modules that import it) instances for exactly the same types are defined for `Show` and `Read`.

Obviously, there is a need for analyzing these general type class directives when we give the programmer such a rich language. It is, for instance, quite easy to pose different predicates which contradict each other, and it is also possible to phrase the same condition in many different ways, let alone directives which are mutually inconsistent and which are likely to burden the programmer with loads of broken directives. As a result, we are likely to take a conservative approach again, allowing what we deem necessary.

2.6 Discussion

Many of the type class directives we have seen thus far specify that the set of instances of a class has some property. For instance, in the case of the `close` directive, the set of instances is fixed. Essentially, such properties should be validated once when they are encountered. From that point on the properties become invariants of the program, and each time a certain property might become invalid (we need to be very precise in stating when a property is violated), we have to check whether it does. However, since every constraint explicitly mentions the type classes involved, we know which directives to check, for each class assertion that is considered at reduction time.

The main idea of the `never`, `close`, and `disjoint` directives is to detect some errors that involve overloading at an earlier stage, and to report precise messages for these errors. Contrary to what was the case for the type inference directives of [HHS03b], the type class directives presented in this paper can actually change the type system. Some of the errors would remain unnoticed without the presence of the type class directives. Furthermore, it remains debatable what to do if, for instance, a `never` directive is followed by the instance it forbids. Should we reject the instance declaration? Or perhaps ignore the directive?

These considerations lead to a design space of the type class directives for which any implementor should make some choices. A conservative approach would be to give warnings instead of error messages, having the advantage that the same set of programs is accepted compared to the situation without type class directives.

Another reasonable approach is to distinguish between the "defining" side of type classes and their instances, and the use of an overloaded function in an expression. We could raise a warning for a declared instance that breaks a class directive. If we then continue, we may choose to report an error if we use this instance during context reduction. For example, if we encounter `instance Num X` for some `X` after the directive `close Num`, then we can emit a warning saying that such an instance is in conflict with the type class directive (in which case one would expect to see the location of this directive). If we encounter a class assertion `Num X` later on, we issue an error, again referring to the type class directive which is responsible.

In any case, we expect that a compiler will be at least as stringent on the "usage" side as it is on the "defining" side: we do not expect a compiler to give errors for forbidden instances, and only warnings when these are used.

A final complication of the presented directives is that a single instance declaration, say `Num Bool`, may invalidate various directives, such as `close Num`, `never (Num Bool)`, or even weird directives such as `disjoint Num Num`. The bottom line is that a decision should be made about how to deal with such a situation: do we list all violations, do we prioritize and if so, how?

3 Extending existing directives and heuristics

Now that we have shown the new type class directives, we can similarly discuss the directives described in [HHS03b] when extended to cope with class assertions. We start with presenting a special heuristic for ill-typed applications and show how type class information can improve type error messages for unification errors.

3.1 The application heuristic

Consider an application of the following form:

```
wrong :: Bool
wrong = (\x -> x) == 2
```

The application as a whole is in error, while on the other hand, the various components, the operator `(==)` and its right and left argument are themselves type correct. A heuristic can be defined which first checks that the function or operator in question does not have more arguments than it can possibly need (for some functions, like the identity function, this number is infinite). If such is not the case, we line up two types: the type of the function (in this case an instantiation of `Eq a => a -> a -> Bool`, say `v9 -> v9 -> Bool` together with the class assertion `Eq v9`), and the inferred types of the arguments (here `v5 -> v5` and `Int`) combined with the type that is expected by the expression's surrounding context (in this case `Bool` from the explicit type). This results in the following situation.

```
function:              v9         -> v9  -> Bool
arguments + context:   (v5 -> v5)    Int    Bool
```

The function application is in error because the function type cannot be unified with `Int` via the type variable `v9`. The next best thing to do is to see if the blame can be laid on one of the two arguments. This is done by "forgetting" one column that corresponds to an argument, and test whether the other columns can be unified. In both cases unification will succeed, and as a result we cannot blame either of the two arguments. The resulting type error message effectively says that the application as a whole is at fault.

```
(2,9): Type error in infix application
 expression       : (\x -> x) == 2
 operator         : ==
   type           : a        -> a   -> Bool
   does not match : (b -> b) -> Int -> Bool
```

However, in the presence of type classes there is an extra demand on `v9`, which is that it should be in the `Eq` class. In the presence of a type class directive specifying that `Eq (a -> b)` is not allowed, there is more evidence that the left argument is at fault. Therefore, we could give a more precise error messages which focuses on the left operand of `==`. Note that if a directive is used to establish this bias, then it should be mentioned along with the unification error message.

```
(2,9): Type error in left operand of an infix application
 expression       : (\x -> x) == 2
 operator         : ==
   type           : Eq a => a -> a -> Bool
 left operand     : \x -> x
   type           : a -> a
   does not match : Int
```

This example shows that type class assertions can be used to give a more accurate error location. Observe also that the two conflicting types do not any longer contain irrelevant information, such as the result type.

3.2 Siblings

The siblings type inference directive can be used if it is discovered that programmers often mix up the names of two or more functions. This may be due to the fact that the names and functionality are similar, or the operators have similar functionality in different programming languages, or because the programmers in question are new to the language. The idea is that we locally replace a function which is involved in a type error with a sibling function to see if that solves the inconsistency. If a sibling directive is used, then a hint is added to the usual type error message, to the effect that he is advised to replace an identifier by a certain sibling.

An example is `(++)`, `(+)` and `(.)` for possible confusion among the (string) append operators in various programming languages.

```
siblings +, ++, .
```

```
f :: String
f = "2" . "1"
```

```
(2,9): Type error in identifier
 operator         : .
   type           : (a -> b) -> (c -> a) -> c -> b
   expected type  : String   -> String   -> String
 probable fix     : use ++ instead
```

After extending the compiler to cope with type classes, the siblings facility should be properly extended as well, in the sense that replacing an identifier with a sibling should not lead to invalid class assertions.

Consider the following code fragment.

```
siblings f, g, h, i
```

```
f ::          a -> a -> Int
g :: Num a => a -> a -> Bool
h :: Eq a  => a -> a -> Bool
i ::          a -> a -> Bool

main = if (f True False) then 2 else 3
```

Obviously, the use of `f` results in a type error. One expects that sibling `g` fails as a replacement for this particular `f` since there is no instance `Num Bool` at this point. The other two siblings, namely `h` and `i`, both fit in this case. Note that checking whether a sibling fits is restricted to its binding group: if the inferred type of a definition is changed by following the probable fix, then things may go wrong in a subsequent binding group (explicit type annotations prevent this from happening).

A further complication arises when the erroneous expression is part of an explicitly typed definition. Care must be taken here to make sure that acting upon the hint does not result in an "explicit type too general" error. A way around this would be to push down (a skolemized version of) the explicit type into the expression so that this can be immediately verified. Take a look at the following example.

```
siblings f, g, h
```

```
f :: a -> a -> Int
g :: Bool -> Bool -> Bool
h :: Eq a => a -> a -> Bool

main :: a -> Bool
main x = f x x
```

Clearly, the identifier `f` in the definition of `main` is in error. We do not expect `g` to be suggested as a replacement, because `main` would then no longer be polymorphic in its first argument. Also the sibling `h` does not fit, because then `main`'s type should have been `Eq a => a -> Bool`.

3.3 The permutation heuristic

The case of the permutation heuristic is in many ways similar to that of the siblings, so we only consider the aspects in which it differs. For a function application that contributes to a type error, we look at all permutations of the arguments. If there is *one single* permutation of the arguments which resolves the type inconsistency, then we add this as a hint to the type error message. Again we have to take the class assertions that we come across into account.

```
f :: Num a => Bool -> a -> a

yes :: [a] -> Int
yes xs = f (length xs) True

no :: [a] -> [Int]
no xs = f [length xs] False
```

For the definition of `yes` we can suggest to flip the arguments. The class assertion `Num Int` that would arise as a result of permuting the arguments trivially holds. However, we should not suggest a permutation for the definition of `no` since we cannot validate the class assertion `Num [Int]` that would otherwise arise.

3.4 Specialized type rules

The major change to the specialized type rules is that we can now list class assertions among the equality constraints in such a type rule. The soundness check can be appropriately generalized to make sure that the underlying type system does not change (more on this in Section 5).

We continue with an example of a specialized type rule for the function `spread`. This function returns the difference between the smallest and largest value of a list.

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

We can give the following specialized type rule for this function, consisting of a deduction rule and a list of constraints.

```
    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
    Hint: make it an instance of the type class Ord.
Num t2: @t2@ should allow numerical operations
    Hint: make it an instance of the type class Num.
```

Although membership of the `Ord` or `Num` class for `t2` implies membership of `Eq`, we first check the latter, and give a more precise error message in case it fails. Only then do we consider the assertions `Ord` and `Num`, in which case we can use that `Eq t2` holds. Note that including the assertion `Eq t2` does not change the validity of the rule, and such should be discovered by the soundness check of this specialized type rule.

Context reduction amounts to reducing the class assertions to head normal form, along the way removing trivially satisfied and duplicate assertions, and generating error messages for unsatisfiable assertions. In many compilers, context reduction is postponed until we come to a point where generalization takes place. This approach implies that for a given expression the equality constraints have been handled before class assertions are considered. Such an order was also apparent in the specialized type rule given above.

Theoretically, there is no reason to keep the separation between equality constraints and class assertions. However, there is a practical reason not to lift the restriction. The complicating factor here is that contrary to equality constraints, a class assertion may be justified, it may not be justified, but it is also possible that it is too early to tell whether it is justified or not.

Compare for instance the constraint `Num v1`, followed by either `v1 == Bool` or `v1 == Int`. If nothing else is known about `v1`, then, at worst, we have to delay solving the class assertion until we have considered all the subsequent equality constraints.

This is in fact the reason why context reduction is delayed until the next generalization point: then we can be sure that all information necessary for reduction is available, while nothing is lost by postponing it until that point. During context reduction, some of the class assertions can be justified (e.g., `Num Int`), while we generate an error message for assertions that cannot be reduced (e.g., `Num Bool`). In the setting of `Haskell 98`, we know that the remaining assertions are in head normal form (see also “Typing Haskell in Haskell” [Jon99]). Some of these assertions will become a part of a type scheme as a result of generalization (to be more precise: the assertions about type variables over which we generalize). The other predicates still have to be justified at a later point. In the end, an ambiguity will be reported for each unjustified class assertion.

One of the main aspects of specialized type rules is that if one considers a given type constraint listed below the rule, then all constraints above it have indeed been satisfied. The information implicit in these preceding constraints is often used when constructing the tailor made error messages. The ability to put class assertions in between the equality constraints, so that they may be put in positions where their consistency cannot yet be determined, can easily lead to unexpected situations where a type error message uses information about class assertions, whose justification is not yet established. Misleading the programmer is the worst of crimes, so we tend to be conservative and advocate that class assertions *must* be listed below the equality constraints.

A concession which could be made is to allow class assertions in the type rule (also in this case, the user is not allowed to assume that the truthfulness of the assertion is established). An example of such an abbreviated rule follows.

```
    xs :: [t3];    Eq t2;
---------------------------
      spread xs :: t2;

t3 == t2: @expr.pp@ should return a value of type @t3@
Ord t2: @t2@ should have a linear ordering imposed on it
    Hint: make it an instance of the type class Ord.
Num t2: @t2@ should allow numerical operations,
    Hint: make it an instance of the type class Num.
```

The ordering of constraints is in fact the same as in the previous version. The only difference is that we do not give special error messages for two of the constraints, since they have been included into the type rule. A default error message is associated with these “implicit” constraints.

In summary, two separate lists of constraints are kept: one for equality constraints and one for class assertions. We expect to start on the class assertions of a binding group only after all the equality constraints for the same binding group have been solved.

Note, though, that it is very well possible to use the idea of phasing introduced in our previous paper [HHS03b] so that context reduction might be done at the end of each phase, refining our approach somewhat. Then class assertions belonging to a certain phase can be reduced, before the constraints of a later phase are considered. Since this can only be done explicitly, and hence consciously, we feel that the chances for making mistakes is quite a bit smaller.

4 The type inference framework

In this section, we describe our type inference framework in which all of the above can be incorporated with relatively little effort. In the next section, we show how the various aspects of the type (class) inference directives can be implemented.

In our view, the type inference process takes the following form.

i. *Generate* the constraints in a syntax directed fashion in the abstract syntax tree.
ii. *Order* the constraints into a list.
iii. *Solve* the list of constraints.

The strength and flexibility of the approach lies in the separation of these concerns, which is hard-wired in the framework. How these various concerns are instantiated is largely a matter of choice. In fact, our `Helium` compiler, which is implemented in the described fashion, has different implementations for the various phases, which can be regarded as plug-ins, enabling us to easily experiment with different instantiations of the framework. Moreover, many of the plug-ins (such as the type graph constraint solver) themselves have plug-ins, e.g., for choosing a collection of heuristics to determine which type error message is reported.

We now consider each of the three phases in turn.

Generation

The generation phase is a type-rule-by-type-rule translation of a collection of bottom-up, constraint based type rules [HHS03a], which essentially defines the type system.

We formulate the type inference process using three sorts of constraints [HHS03a]: equality constraints, which also appear in specialized type rules and which can be solved by unification, and the implicit and explicit instance constraints. The latter is used in dealing with explicit type annotations from a program, while the former is used to deal with let-polymorphism: to obtain a full decoupling of the generation and solving of constraints, we cannot yet know the polymorphic type of a let-defined identifier before generating the constraints for the body.

There are other ways of dealing with let-polymorphism such as duplicating the constraints of a let-definition for each of its occurrences. We want to avoid duplication of constraints (to avoid duplication of effort and type errors). Therefore we insist that there are generalization points in the solving process, where we can be sure that we have the polymorphic (generalized) type, such that we can instantiate it. The information necessary to determine whether that point has arrived is present in the implicit instance constraint (see [HHS03a] for more details).

Ordering

The result of the generation phase is a constraint tree that follows the shape of the abstract syntax tree. Each node is decorated with a set of constraints generated for that node.

The ordering of the constraints in the abstract syntax tree can now be formulated as a walk over the constraint tree. In principle, any walk is allowed, except for the restriction that constraints arising from a let-definition always come before constraints generated for the body of a let-expression. The transition between these two is what is referred to as a *generalization point*: this is the point where all the information necessary to compute the generalized type of a let-defined identifier is known. More specifically, we know of each type variable whether it is monomorphic or polymorphic.

The ordering phase allows a programmer to choose his favourite type of inferencing: bottom-up, top-down and so on, simply by choosing a certain walk over the tree to convert the various sets of constraints into a list. For instance, a pre-order treewalk roughly corresponds to algorithm $\mathcal{M}$, and a post-order treewalk to algorithm $\mathcal{W}$ (except that we have to mimic that types of identifiers are passed down via an environment, instead of collected upwards as is the case in the bottom-up type inference rules; this is called *spreading* of type constraints in [HHS03a]).

Solving

A solver is an algorithm which, when given a list of constraints, returns a substitution that satisfies a subset of the constraints in the list, and for each of the constraints which it does not satisfy, a suitable error message.

The most straightforward solver is the *greedy* solver, which computes a substitution by considering the constraints one by one. If the next constraint is consistent with the result so far, then it is incorporated into the substitution. If it is not, then it is assumed to be incorrect and an error message is generated. (Note that a constraint has some extra information associated with it to generate such an error message.) It is a matter of taste whether one wants to continue after an error has been found to search for more inconsistencies. A greedy solver can be quite fast, but it suffers from a bias introduced by the treewalk chosen in the ordering phase. Although there is a certain freedom in choosing a preferred treewalk, a more flexible way of dealing with order is to let the constraints decide amongst themselves who is the likely culprit.

A type graph was invented for exactly this reason: it is a data structure which can represent substitutions, i.e. consistent sets of constraints, but also inconsistent ones. A graph can be constructed from a list of equality constraints in which everything which should be equivalent ends up in the same component of the graph. Some of these equivalences are inherited from others, e.g., if `a -> b` and `c -> d` are equivalent, then `a` and `c` as well as `b` and `d` are also equivalent. After we have added the constraints to the graph, we search for components which contain two different type constructors. If this is the case, a number of heuristics have been defined which try to pinpoint the most likely edge(s) (in other words, constraints) to cut out of the graph. Repeatedly doing this should result in a consistent type graph. Each such graph represents a substitution, while each of the cut-edges corresponds to an invalid equality constraint, which is used to generate appropriate error messages. In some cases, the heuristics can influence the message, for instance by adding a hint.

There may be various heuristics, each with their own weight, and each with their own measure of trust in the correctness (or invalidity) of a constraint. A tunable voting mechanism is present in the compiler, to obtain the best mix of heuristics for a given programmer.

After processing all the generated constraints for a program, we proceed to verify that for each explicitly typed function, its explicit type is an instance of its inferred type. If this is not the case, then a "declared type too general" error message is displayed. Finally, if there are no error messages, then we may generate a list of warnings, such as missing explicit types.

Summarizing, the final outcome of the type inferencing process is a substitution, a list of warnings and a list of errors for the constraints which were found to be inconsistent with the others. When giving feedback, we usually start with the unifications errors, followed by the type class errors, then the "declared type too general" errors, and if we have none of either of these, the warnings.

5 Implementation issues

The concern of this section lies in explaining how our implementation deals with the extensions described in the first part of this paper, by pointing out where these fit into the type inference process described in the previous section. In some cases, we can be quite concrete, because of our experiences with the `Helium` compiler [HLvI03].

We do not discuss the implementation of the type class directives here. Once some design decisions have been made, we expect the implementation to be rather straightforward. The extent and consequences of these design decisions were explained at the end of Section 2.

5.1 The specialized type rules

There are two important aspects to the specialized type rules. The first is, how can they be elegantly implemented, and, secondly, how can we avoid that a given specialized type rule changes the underlying type system?

We consider the former question first. The idea behind the specialized type rule is, conceptually, that certain expressions are treated in special way. For instance, an application of `map` is dealt with differently from an ordinary application of a binary function to its arguments, in the sense that the ordering of the constraints for this expression (and the generated error messages) differ from the default.

During a top-down traversal of the abstract syntax tree, we look for expressions (which may in fact be arbitrarily complicated, as long as they do not change the scope) for which a specialized type rule has been defined. If this is the case, then we make sure that the constraints are generated in the specified order, and that each constraint "knows" its specialized error message (if one is given). Obviously, specialized type rules are dealt with in the generation phase, and locally override the treewalk chosen for the tree as whole.

Note that the fact that a specialized type rule uses the same type environment above and below the type rule (in other words, new bindings may not be introduced in such a rule) simplifies matters greatly, also notationally for the programmer.

The second part concerns the sanity check on specialized type rules, which verifies that the underlying type system is not changed. This is done by defining an entailment relation on sets of constraints. Consider the following type rule again.

```
    xs :: [t3];  Eq t2;
------------------------
  spread xs :: t2;

t3 == t2
Ord t2
Num t2
```

Now we can compute two sets of type constraints (including class assertions). One, say S_1, is based on the set of constraints in and below the rule, where we have to unfold the typing `xs :: [t3]` in the type rule to `xs :: v1` (for some fresh `v1`) and a constraint `v1 == [t3]`.

A second set of constraints, say S_2, is based on the expression `spread xs` where we use the default type rule for application.

Here, we make sure that each of the meta-variables (xs), the result type (t2), and the instantiated type of the identifiers taken from the environment (spread), are the same for the two sets (or, if not, that the renaming between these type variables in the two sets is known).

Now, a set of constraints S_1 entails a set of constraints S_2, if every substitution which satisfies S_1 also satisfies S_2. Entailment is easily decided by computing a most general substitution satisfying S_1, and to verify that it also satisfies S_2. If S_1 entails S_2 and vice versa, then we can be sure that the underlying type system remains unchanged. If this is not the case, then we expect an error message explaining this fact. Note that we insist that the type system is unchanged. This could be relaxed by insisting only on soundness: in that case every substitution which satisfies S_1 should satisfy S_2, but not the other way around.

5.2 Siblings, permutation and application heuristics

Many of the heuristics we have described in this paper are possible because we use a type graph in which it is easy to experiment with slightly different sets of constraints. Consider the permutation heuristic. Assume we are given a type graph for a certain ill-typed expression, and we suspect that this may be due to a permutation of the arguments of a certain function. We can now easily modify the graph by rearranging some edges in the graph such that it represents the same expression, except that the arguments to the suspect application are permuted in some way. Then, we can check that the result becomes consistent, and if no other reordering has the same effect, then we have some confidence that this is the preferred correction of the program.

The siblings directive is performed in the same way: if we replace a function f by its sibling g, then we remove the edges/constraints for which the type of f was responsible, add in those of g, and see whether that solves the problem. If so, then the sibling directive nominates the constraint which determines that f is the function to be applied, and adds a hint to the error message.

Note that the permutation heuristic and sibling heuristic are competitors: their outcome is weighted in some way to obtain the most likely candidate for removal. The weighting is subject to change, depending on the preferences of the programmer.

The application heuristic was described in some detail in Section 3, which already corresponds quite closely to its implementation.

6 Related work

A number of papers address the problem of poor type error messages that are produced by most modern compilers for functional programming languages. We believe that this is a serious obstacle to value such a language, and that it results in a steep learning curve. Various propositions to extend Haskell, such as higher-ranked types, implicit arguments, and various extensions to the class system, complicate the type system and increase the size of the problem to report concise and helpful error messages.

Different approaches to improve on the quality of error message have been suggested. One of the first proposals is by Wand [Wan86], 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 [WJ86] 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.

More recently, Yang, Michaelson and Trinder [YMT02] 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 [McA01] suggested to use unification of types modulo linear isomorphism to automatically repair ill-typed programs. To our knowledge, this is the only work that addresses constructive advice to correct a program. Haack and Wells [HW03] compute a minimal set of program locations (a type error slice) that contribute to a type inconsistency. Stuckey, Sulzmann, and Wazny [SSW03] present the Chameleon type debugger, which helps to locate type errors in an interactive way.

Despite all the attention to improve the type error messages, far too few ideas have been suggested to tackle the complications that extensions to the language (such as type classes) introduce.

Several type inference frameworks follow a constraint-based approach. Aiken and Wimmers [AW93] propose a constraint solving system that can handle inclusion constraints over type expressions. The Hindley-Milner type system, including let-polymorphism, can be formulated in terms of these constraints. Another interesting direction is the HM(X) framework by Odersky, Sulzmann, and Wehr[OSW99]. This framework is a constraint-based formulation of the Hindley-Milner type system including let-polymorphism, which is parameterized over some constraint system X. All kinds of extensions to the type system can be captured via this abstraction.

7 Conclusion and future work

This paper discusses 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. Type inference directives that have been described in an earlier paper [HHS03b] are generalized to cope with class constraints. Summarizing, this paper offers the following contributions and improvements over work on type inference directives.

- We have devised type class directives that help to reject programs that contain a class context that is unlikely to be fulfilled. For instance, an instance for `Eq (a -> b)` will only be supplied in exceptional cases. Because these directives prevent some programs to compile that could otherwise be legal, we also considered the question how one may deal with this fact. Special domain-specific error messages can be reported by such a directive.
- Some type class directives can help to disambiguate overloading, which leads to more programs being accepted (or well-typed). In particular, the default directive helps to resolve overloading, and could be a directive instead of some special syntax of the language.
- Class assertions have been added to the specialized type rules

and the soundness check has been appropriately generalized. The implementation of the other inference directives has been generalized to cope with class constraints, and to prevent misleading suggestions to correct a program.

- We have hinted how type classes and class assertions can be added to a constraint-based type inference framework. Here, we have followed the `Haskell 98` standard [Has03].
- We have presented a heuristic which uses class information about type variables to decide which part of the program is the most likely cause of error. This is a generalization of the minority isolation proposed by Walz and Johnson [WJ86].

We see several possible directions for future research. The most obvious direction is to explore directives for a number of the proposed extensions to the type class system (see Peyton Jones, Jones and Meijer [JJM97] for an exploration of the design space of type classes). More specifically, we want to investigate how directives may remove the reason for not having those extensions.

The more expressive the directives become, the more need there is for some form of analysis for these directives. In particular, the generalization of type class directives described in Section 2.5 necessitate a closer look in this direction. It may be possible to come up with a nice language for specifying predicates over sets of instances for type classes. We need more practical experience to see what functionality is necessary and helpful.

Notwithstanding all our efforts, we hope that every compiler also supports the possibility to turn type classes off altogether.

Acknowledgements

We thank Doaitse Swierstra for his suggestions and comments on an earlier version of this paper.

8 References

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[Has03] Haskell 98: The revised report, January 2003. Simon Peyton Jones (editor), `http://haskell.org/onlinereport/`.

[HHS03a] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.

[HHS03b] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

[HLvI03] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.

[HW03] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*, pages 284–301, April 2003.

[JJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[Jon99] Mark Jones. Typing Haskell in Haskell. In *Haskell Workshop*, September 1999.

[McA01] Bruce McAdam. How to repair type errors automatically. In *3rd Scottish Workshop on Functional Programming*, pages 121–135. Stirling, U.K., August 2001.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.

[SSW03] Peter Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *Haskell Workshop*, pages 72 – 83, New York, 2003. ACM Press.

[Wan86] M. Wand. Finding the source of type errors. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, St. Petersburg, FL, January 1986.

[WJ86] J. A. Walz and G. F. Johnson. 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*, pages 44–57, St. Petersburg, FL, January 1986.

[YMT02] Jun Yang, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.