Type classes: an exploration of the design space

Simon Peyton Jones
University of Glasgow and Oregon Graduate Institute

Mark Jones
University of Nottingham

Erik Meijer
University of Utrecht and Oregon Graduate Institute

May 2, 1997

Abstract

When type classes were first introduced in Haskell they were regarded as a fairly experimental language feature, and therefore warranted a fairly conservative design. Since that time, practical experience has convinced many programmers of the benefits and convenience of type classes. However, on occasion, these same programmers have discovered examples where seemingly natural applications for type class overloading are prevented by the restrictions imposed by the Haskell design.

It is possible to extend the type class mechanism of Haskell in various ways to overcome these limitations, but such proposals must be designed with great care. For example, several different extensions have been implemented in Gofer. Some of these, particularly the support for multi-parameter classes, have proved to be very useful, but interactions between other aspects of the design have resulted in a type system that is both unsound and undecidable. Another illustration is the introduction of constructor classes in Haskell 1.3, which came without the proper generalization of the notion of a context. As a consequence, certain quite reasonable programs are not typable.

In this paper we review the rationale behind the design of Haskell's class system, we identify some of the weaknesses in the current situation, and we explain the choices that we face in attempting to remove them.

1 Introduction

Type classes are one of the most distinctive features of Haskell (Hudak et al. [1992]). They have been used for an impressive variety of applications, and Haskell 1.3[1] significantly extended their expressiveness by introducing *constructor classes* (Jones [1995a]).

All programmers want more than they are given, and many people have bumped up against the limitations of Haskell's class system. Another language, Gofer (Jones [1994]), that has developed in parallel with Haskell, enjoys a much more liberal and expressive class system. This expressiveness is definitely both useful and used, and transferring from Gofer to Haskell can be a painful experience. One feature that is particularly often missed is *multi-parameter* type classes — Section 2 explains why.

The obvious question is whether there is an upward-compatible way to extend Haskell's class system to enjoy some or all of the expressiveness that Gofer provides, and perhaps some more besides. The main body of this paper explores this question in detail. It turns out that there are a number of interlocking design decisions to be made. Gofer and Haskell each embody a particular set, but it is very useful to tease them out independently, and see how they interact. Our goal is to explore the design space as clearly as possible, laying out the choices that must be made, and the factors that affect them, rather than prescribing a particular solution (Section 4). We find that the design space is rather large; we identify nine separate design decisions, each of which has two or more possible choices, though not all combinations of choices make sense. In the end, however, we do offer our own opinion about a sensible set of choices (Section 6).

A new language feature is only justifiable if it results in a simplification or unification of the original language design, or if the extra expressiveness is truly useful in practice. One contribution of this paper is to collect together a fairly large set of examples that motivate various extensions to Haskell's type classes.

2 Why multi-parameter type classes?

The most visible extension to Haskell type classes that we discuss is support for multi-parameter type classes. The possibility of multi-parameter type classes has been recognised since the original papers on the subject (Kaes [1988]; Wadler & Blott [1989]), and Gofer has always supported them.

This section collects together examples of multi-parameter type classes that we have encountered. None of them are new, none will be surprising to the cognescenti, and many have appeared *inter alia* in other papers. Our purpose in collecting them is to provide a shared database of motivating examples. We would welcome new contributions.

[1] The current iteration of the Haskell language is Haskell 1.4, but it is identical to Haskell 1.3 in all respects relevant to this paper.

2.1 Overloading with coupled parameters

Concurrent Haskell (Peyton Jones, Gordon & Finne [1996]) introduces a number of types such as mutable variables `MutVar`, "synchronised" mutable variables `MVar`, channel variables `CVar`, communication channels `Channel`, and skip channels `SkipChan`, all of which come with similar operations that take the form:

```
newX :: a -> IO (X a)
getX :: X a -> IO a
putX :: X a -> a -> IO ()
```

where `X` ranges over `MVar` etc. Here are similar operations in the standard state monad:

```
newST :: a -> ST s (MutableVar s a)
getST :: MutableVar s a -> ST s a
putST :: MutableVar s a -> a -> ST s ()
```

These are manifestly candidates for overloading; yet a single parameter type class can't do the trick. The trouble is that in each case the monad type and the reference type come as a pair: (`IO`, `MutVar`) and (`ST s`, `MutableVar s`). What we want is a multiple parameter class that abstracts over both:

```
class Monad m => VarMonad m v where
  new :: a -> m (v a)
  get :: v a -> m a
  put :: v a -> a -> m ()

instance VarMonad IO MutVar where ....
instance VarMonad (ST s) (MutableVar s) where ...
```

This is quite a common pattern, in which a two-parameter type class is needed because the class signature is really over a *tuple* of types and where instance declarations capture direct relationships between specific tuples of type constructors. We call this *overloading with coupled parameters.*

Here are a number of other examples we have collected:

- The class `StateMonad` (Jones [1995]) carries the state around naked, instead of inside a container as in the `VarMonad` example:

  ```
  class Monad m => StateMonad m s where
     getS :: m s
     putS :: s -> m ()
  ```

  Here the monad `m` carries along a state of type `s`; `getS` extracts the state from the monad, and `putS` overwrites the state with a new value. One can then define instances of `StateMonad`:

  ```
  newtype State s a = State (s -> (a,s))
  instance StateMonad (State s) s where ...
  ```

  Notice the coupling between the parameters arising from the repeated type variable `s`. Jones [1995] also defines a related class, `ReaderMonad`, that describes computations that read some fixed environment

  ```
  class Monad m => ReaderMonad m e where
     env :: e -> m a -> m a
     getenv :: m e

  newtype Env e a = Env (e -> a)
  instance ReaderMonad (Env e) e where ...
  ```

- Work in Glasgow and the Oregon Graduate Institute on hardware description languages has led to class declarations similar to this:

  ```
  class Monad ct => Hard ct sg where
    const :: a -> ct (sg a)
    op1 :: (a -> b) -> sg a -> ct (sg b)
    op2 :: (a -> b -> c) -> sg a -> sg b -> ct (s c)

  instance Hard NetCircuit NetSignal where ...
  instance Hard SimCircuit SimSignal where ...
  ```

  Here, the circuit constructor, `ct` is a monad, while the signal constructor, `sg`, serves to distinguish values available at circuit-construction time (of type `Int`, say) from those flowing along the wires at circuit-execution time (of type `SimSignal Int`, say). Each instance of `Hard` gives a different interpretation of the circuit; for example, one might produce a net list, while another might simulate the circuit.

  Like the `VarMonad` example, the instance type come as a pair; it would make no sense to give an instance for `Hard NetCircuit SimSignal`.

- [2] The Haskell prelude defines defines the following two functions for reading and writing files

  ```
  readFile :: FilePath -> IO String
  writeFile :: FilePath -> String -> IO ()
  ```

  Similar functions can be defined for many more pairs of device handles and communicatable types, such as mice, buttons, timers, windows, robots, etc.

  ```
  readMouse  :: Mouse -> IO MouseEvent
  readButton :: Button -> IO ()
  readTimer  :: Timer -> IO Float

  sendWindow :: Window -> Picture -> IO ()
  sendRobot  :: Robot -> Command -> IO ()
  sendTimer  :: Timer -> Float -> IO ()
  ```

  These functions are quite similar to the methods `get :: VarMonad r m => r a -> m a` and `put :: VarMonad r m => r a -> a -> m ()` of the `VarMonad` family, except that here the monad `m` is fixed to `IO` and the choice of the value type `a` is coupled with the box type `v a`. So what we need here is a multi-parameter class that overloads on `v a` and `a` instead:

  ```
  class IODevice handle a where
     receive :: handle -> IO a
     send :: handle -> a -> IO a
  ```

  (Perhaps one could go one step further and unify `class IODevice r a` and `class Monad m => StateMonad m r` into a three parameter class `class Monad m => Device m r a`.)

[2] This example was suggested by Enno Scholz.

- [3] An appealing application of type classes is to describe mathematical structures, such as groups, fields, monoids, and so on. But it is not long before the need for coupled overloading arises. For example:

```
class (Field k, AdditiveGroup a)
   => VectorSpace k a where
      @* :: k -> a -> a
      ...
```

  Here the operator @* multiplies a vector by a scalar.

2.2 Overloading with constrained parameters

Libraries that implement sets, bags, lists, finite maps, and so on, all use similar functions (`empty`, `insert`, `union`, `lookup`, etc). There is no commonly-agreed signature for such libraries that usefully exploits the class system. One reason for this is that multi-parameter type classes are absolutely required to do a good job. Why? Consider this first attempt:

```
class Collection c where
  empty  :: c a
  insert :: a -> c a -> c a
  union  :: c a -> c a -> c a
  ...etc...
```

The trouble is that *the type variable* `a` *is universally quantified* in the signature for `insert`, `union`, and so on. This means we cannot use equality or greater-than on the elements, so we cannot make sets an instance of `Collection`, which rather defeats the object of the exercise. By far the best solution is to use a two-parameter type class, thus:

```
class Collection c a where
  empty  :: c a
  insert :: a -> c a -> c a
  union  :: c a -> c a -> c a
  ...etc...
```

The use of a multi-parameter class allows us to make instance declarations that constrain the element type *on a per-instance basis*:

```
instance Eq a => Collection ListSet a where
   empty = ...
   insert a xs = ...
   ...etc..

instance Ord a => Collection TreeSet a where
   empty = ...
   insert x t = ...
   ...etc...
```

The point is that different instance declarations can constrain the element type, `a`, in different ways. One can look at this as a variant of coupled-parameter overloading (discussed in the preceding section). Here, the second type in the pair is *constrained* by the instance declaration (e.g. "`Ord a =>...`"), rather than *completely specified* as in the previous section. In general, in this form of overloading, one or more of the parameters in any instance is a variable that serves as a hook, either for one of the other arguments, or for the instance context and member functions to use.

The *parametric type classes* of Chen, Hudak & Odersky [1992] also deal quite nicely with the bulk-types example, but their assymetry does not suit the examples of the previous section so well. A full discussion of the design choices for a bulk-types library is contained in Peyton Jones [1996].

2.3 Type relations

One can also construct applications for multi-parameter classes where the relationships between different parameters are much looser than in the examples that we have seen above. After all, in the most general setting, a multi-parameter type class `C` could be used to represent an arbitrary relation between types where, for example, (a, b) is in the relation if, and only if, there is an instance for (`C` a b).

- One can imagine defining an isomorphism relationship between types (Liang, Hudak & Jones [1995]):

```
class Iso a b where
  iso :: a -> b
  osi :: b -> a

instance Iso a a where iso = id
```

- One could imagine overloading Haskell's field selectors by declaring a class

```
class Hasf a b where
  f :: a -> b
```

  for any field label `f`. So if we have the data type `Foo = Foo{foo :: Int}`, we would get a class declaration `class Hasfoo a b where foo :: a -> b` and an instance declaration

```
instance Hasfoo Foo Int where
  foo (Foo foo) = foo
```

  This is just a cut-down version of the kind of extensible records that were proposed by Jones (Jones [1994]).

These examples are "looser" than the earlier ones, because the result types of the class operations do not mention all the class type variables. In practice, we typically find that such relations are too general for the type class mechanisms, and that it becomes remarkably easy to write programs whose overloading is ambiguous.

For example, what is the type of `iso 'a' == iso 'b'`? The `iso` function is used at type `Char -> b`, and the resulting values of `iso 'a'` and `iso 'b'` are compared with `(==)` used at type `b -> b -> Bool`. However this intermediate type is completely unconstrained and hence the resulting type, `(Eq b, Iso Char b) => Bool`, is ambiguous. One runs into similar problems quickly when trying to use overloading of field selectors. We discuss ambiguity further in Section 3.7.

[3] This example was suggested by Sergey Mechveliani.

2.4 Summary

In our view, the examples of this section make a very persuasive case for multi-parameter type classes, just as `Monad` and `Functor` did for constructor classes. These examples cry out for Haskell-style overloading, but it simply cannot be done without multi-parameter classes.

3 Background

In order to describe the design choices related to type classes we must briefly review some of the concepts involved.

3.1 Inferred contexts

When performing type inference on an expression, the type checker will infer (a) a monotype, and (b) a *context*, or set of *constraints*, that must be satisfied. For example, consider the expression:

```
\xs -> case xs of
      []     -> False
      (y:ys) -> y > z || (y==z &&  ys==[z])
```

Here, the type checker will infer that the expression has the following context and type:

Context: $\{$`Ord a`, `Eq a`, `Eq [a]`$\}$
Type: `[a] -> Bool`

The constraint `Ord a` arises from the use of `>` on an element of the list, `y`; the constraint says that the elements of the list must lie in class `Ord`. Similarly, `Eq a` arises from the use of `==` on a list element. The constraint `Eq [a]` arises from the use of `==` on the tail of the list; it says that lists of elements of type `a` must also lie in `Eq`.

These typing constraints have an operational interpretation that is often helpful, though it is not required that a Haskell implementation use this particular operational model. For each constraint there is a corresponding *dictionary*— a collection of functions that will be passed to the overloaded operator involved. In our example, the dictionary for `Eq [a]` will be a tuple of methods corresponding to the class `Eq`. It will be passed to the second overloaded `==` operator, which will simply select the `==` method from the dictionary and apply it to `ys` and `[z]`. You can think of a dictionary as concrete, run-time "evidence" that the constraint is satisfied.

3.2 Context reduction

Contexts can be simplified, or *reduced*, in three main ways:

1. *Eliminating duplicate constraints.* For example, we can reduce the context $\{$`Eq` τ, `Eq` $\tau\}$ to just $\{$`Eq` $\tau\}$.

2. *Using an instance declaration.* For example, the Haskell `Prelude` contains the standard instance declaration:

   ```
   instance Eq a => Eq [a] where ...
   ```

   This instance declaration specifies how we can use an equality on values of type `a` to define an equality on lists of type `[a]`. In terms of the dictionary model, the instance declaration specifies how to construct a dictionary for `Eq [a]` from a dictionary for `Eq a`. Hence we can perform the following context reduction:

   $$\{\texttt{Ord a}, \texttt{Eq a}, \texttt{Eq [a]}\} \longrightarrow \{\texttt{Ord a}, \texttt{Eq a}\}$$

   We say that a constraint *matches* an instance declaration if there is a substitution of the type variables in the instance declaration head that makes it equal to the constraint.

3. *Using a* `class` *declaration.* For example, the class declaration for `Ord` in the Haskell `Prelude` specifies that `Eq` is a superclass of `Ord`:

   ```
   class Eq a => Ord a where ...
   ```

   What this means is that every instance of `Ord` is also an instance of `Eq`. In terms of the dictionary model, we can read this as saying that each `Ord` dictionary contains an `Eq` dictionary as a sub-component. So the constraint `Eq a` is implied by `Ord a`, and it follows that we can perform the following context reduction:

   $$\{\texttt{Ord a}, \texttt{Eq a}\} \longrightarrow \{\texttt{Ord a}\}$$

$$\frac{TV(P) \subseteq dom(\theta) \qquad \texttt{instance } C \texttt{ => } P \texttt{ where } \ldots}{\theta(C) \Vdash \theta(P)} \quad \text{(inst)}$$

$$\frac{TV(P) \subseteq dom(\theta) \qquad \texttt{class } C \texttt{ => } P \texttt{ where } \ldots}{\theta(P) \Vdash \theta(C)} \quad \text{(super)}$$

$$\frac{Q \subseteq P}{P \Vdash Q} \quad \text{(mono)}$$

$$\frac{P \Vdash Q \qquad Q \Vdash R}{P \Vdash R} \quad \text{(trans)}$$

Figure 1: Rules for entailment

More precisely, we say that Q *entails* P, written $Q \Vdash P$, if the constraints in P are implied by those in Q. We define the meaning of class constraints more formally using the definition of the entailment relation defined in Figure 1. The first two rules correspond to (2) and (3) above[4]. The substitution θ maps type variables to types; it allows class and instance declarations to be used at substitution-instances of their types. For example, from the declaration

```
instance Eq a => Eq [a] where ...
```

[4]Notice that in (*inst*), C and P appear in the same order on the top and bottom lines of the rules, whereas they are reversed in (*super*). This suggest an infelicity in Haskell's syntax, but one that it is perhaps too late to correct!

we can deduce that {`Eq` τ} $\Vdash$ {`Eq [`τ`]`}, for an arbitrary type τ[5]. The remaining rules explain that entailment is monotonic and transitive as one would expect.

The connection between entailment and context reduction is this: to reduce the context P to P' it is necessary (but perhaps not sufficient) that $P' \Vdash P$. The reason that entailment is not sufficient for reduction concerns overlapping instances: there might be more than one P' with the property that $P' \Vdash P$, so which should be chosen? Overlapping instance declarations are discussed in Section 3.6 and 4.4.

3.3 Failure

Context reduction *fails*, and a type error is reported, if there is no instance declaration that can match the given constraint. For example, suppose that we are trying to reduce the constraint `Eq (Tree` τ`)`, and there is no instance declaration of the form

```
instance ... => Eq (Tree ...) where ...
```

Then we can immediately report an error, even if τ contains type variables that will later be further instantiated, because no further refinement of τ can possibly make it match. This strategy conflicts slightly with separate compilation, because one could imagine that a separately-compiled library might not be able to "see" all the instance declarations for `Tree`.

Arguably, therefore, rather than reporting an error message, context reduction should be deferred (see Section 4.3), in the hope that an importing module will have the necessary instance declaration. However, that would postpone the production of even legitimate missing-instance error messages until the "main" module is compiled (when no further instance declarations can occur), which is quite a serious disadvantage. Furthermore, it is usually easy to arrange that the module that needs the instance declaration is able to "see" it. If this is so, then failure can be reported immediately, regardless of the context reduction strategy.

3.4 Tautological constraints

A *tautological* constraint is one that is entailed by the empty context. For example, given the standard instance declarations, `Ord [Int]` is a tautological constraint, because the instance declaration for `Ord [a]`, together with that for `Ord Int` allow us to conclude that {} $\Vdash$ {`Ord [Int]`}.

A *ground* constraint is one that mentions no type variables. It is clear that a ground constraint is erroneous (that is, cannot match any instance declaration), or is tautological. It is less obvious that a tautological constraint does not have to be ground. Consider

```
instance Eq a => Foo (a,b) where ...
```

and let us assume for the moment that overlapping instance declarations are prohibited (Section 4.4). Now suppose that the context {`Foo (Int,t)`} is subject to context reduction. *Regardless of the type* `t`, it can be simplified to {`Eq Int`} (using the instance declaration above), and thence to {} (using the `Int` instance for `Eq`). Even if `t` contains type variables, the constraint `Foo (Int,t)` can still be reduced to {}, so it is a tautological constraint.

Another example of one of these tautological constraints that contain type variables is given by this instance declaration:

```
instance Monad (ST s) where ...
```

This declares the state transformer type, `ST s`, to be a monad, regardless of the type `s`.

If, on the other hand, overlapping instance declarations *are* permitted, then reducing a tautological constraint in this way is not legitimate, as we discuss in Section 4.4.

3.5 Generalisation

Suppose that the example in Section 3.1 is embedded in a larger expression:

```
let
   f = \xs -> case xs of
        []     -> False
        (y:ys) -> y > z ||
                    (y==z &&  ys==[z])
in
....
```

Having inferred a type for the right-hand side of `f`, the type checker must *generalise* this type to obtain the polymorphic type for `f`. Here are several possible types for `f`:

```
f :: (Ord a) => [a] -> Bool
f :: (Ord a, Eq a) => [a] -> Bool
f :: (Ord a, Eq a, Eq [a]) => [a] -> Bool
```

Which of these types is inferred depends on how much context reduction is done before generalisation, a topic we discuss later (Section 4.3). For the present, we only need note (a) that there is a choice to be made here, and (b) that the time that choice is crystallised is at the moment of generalisation.

What we mean by (b) is that it makes no difference whether context reduction is done just before generalising `f`, or just after inferring the type of the sub-expression `(ys==[z])`, or anywhere in between; all that matters is how much is done before generalisation.

3.6 Overlapping instance declarations

Consider these declarations:

```
class MyShow a where
  myShow :: a -> String
```

[5] In Gofer, an instance declaration `instance P => C where ...` brings about the axiom $C \Vdash P$, because the representation in Gofer of a dictionary for C contains sub-dictionaries for P. In retrospect, this was probably a poor design decision because it is not always very intuitive. Moreover, it was later discovered that this is incompatible with overlapping instances: while either one is acceptable on its own, the combination results in an unsound type system. The Gofer type system still suffers from this problem today because of concerns that removing support for either feature would break a lot of existing code.

```
instance MyShow a => MyShow [a] where
  myShow = myShow1
instance MyShow [Char] where
  myShow = myShow2
```

Here, the programmer wants to use a different method for `myShow` when used at `[Char]` than when used at other types. We say that the two instance declarations *overlap*, because there exists a constraint that matches both. For example, the constraint `MyShow [Char]` matches both declarations. In general, two instance declarations

```
instance P1 => Q1 where ...
instance P2 => Q2 where ...
```

are said to *overlap* if `Q1` and `Q2` are unifiable. This definition is equivalent to saying that there is a constraint `Q` that matches both `Q1` and `Q2`. Overlapping instance declarations are illegal in Haskell, but permitted in Gofer.

When, during context reduction, a constraint matches two overlapping instance declarations, which should be chosen? We will discuss this question in Section 4.4, but for now we address the question of whether or not overlapping instance declarations are useful. We give two further examples.

3.6.1 "Default methods"

One application of overlapping instance declarations is to define "default methods". Haskell has the following standard classes:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a

class Functor f where
  map :: (a -> b) -> f a -> f b
```

Now, in any instance of `Monad`, there is a sensible definition of `map`, an idea we could express like this:

```
instance Monad m => Functor m where
  map f m = [f x | x <- m]
```

These instance declarations overlap with all other instances of `Functor`. (Whether this is the best way to explain that an instance of `Monad` has a natural definition of `map` is debatable.)

3.6.2 Monad transformers

A second application of overlapping instance declarations arises when we try to define *monad transformers*. The idea is given by Jones [1995]:

> "In fact, we will take a more forward-thinking approach and use the constructor class mechanisms to define different families of monads, each of which supports a particular collection of simple primitives. The benefit of this is that, later, we will want to consider monads that are simultaneously instances of several different classes, and hence support a combination of different primitive features. This same approach has proved to be very flexible in other recent work (Jones [1995a]; Liang, Hudak & Jones [1995])."

To combine the features of monads we introduce a notion of a monad transformer; the idea is that a monad transformer `t` takes a monad `m` as an argument and produces a new monad `(t m)` as a result that provides all of the computational features of `m`, *plus* some new ones added in by the transformer `t`.

```
class MonadT t where
  lift :: Monad m => m a -> t m a
```

For example, the state monad transformer that can add state to any monad:

```
newtype StateT s m a = StateT (s -> m (a,s))

instance MonadT (StateT s) where ...
instance Monad m
      => StateMonad (StateT s m) s where ...
```

Critically, we also need to know that any properties enjoyed by the original monad, are also supported by the transformed monad. We can capture this formally using:

```
instance (MonadT t, StateMonad m s)
      => StateMonad (t m) s where
  update f = lift (update f)
```

Note the overlap with the previous instance declaration, which plays an essential role. Defining monad transformers in this way allows us to build up composite monads, with automatically generated liftings of the important operators. For example:

```
f :: (StateMonad m Int, StateMonad m Char)
  => Int -> Char -> m (Int,Char)
f x y  =  do  x' <- update (const x)
              y' <- update (const y)
              return (x',y')
```

Later, we might call this function with an integer and a character argument on a monad that we've constructed using the following:

```
type M = StateT Int (ErrorT (State Char))
```

Notice that the argument of the `StateT` monad transformer is not `State Char` but rather the enriched monad `(ErrorT (State Char))`, assuming that `ErrorT` is another monad transformer. Now, the overloading mechansims will automatically make sure that the first call to `update` in `f` takes place in the outermost `Int` state monad, while the second call will be lifted up from the depths of the innermost `Char` state monad.

3.7 The ambiguity problem

As we observed earlier, some programs have *ambiguous* typings. The classic example is `(show (read s))`, where different choices for the intermediate type (the result of the `read` might lead to different results). Programs with ambiguous typings are therefore rejected by Haskell.

Preliminary experience, however, is that multi-parameter type classes give new opportunities for ambiguity. Is there any way to have multi-parameter type classes without risking ambiguity? Our answer here is "no". One approach that has been suggested to the ambiguity problem in single-parameter type classes is to insist that all class operations take as their first argument a value of the class's type (Odersky, Wadler & Wehr [1995]). Though it is theoretically attractive, there are too many useful classes that disobey this constraint (`Num`, for example, and overloaded constants in general), so it has not been adopted in practice. It is also not clear what the rule would be when we move to constructor classes, so that the class's "type" variable ranges over type constructors.

If no workable solution to the ambiguity problem has been found for single parameter classes, we are not optimistic that one will be found for multi-parameter classes.

4 Design choices

We are now ready to discuss the design choices that must be embodied in a type-class system of the kind exemplified by Haskell. Our goal is to describe a design space that includes Haskell, Gofer, and a number of other options beside. While we express opinions about which design choices we prefer, our primary goal is to give a clear description of the design space, rather than to prescribe a particular solution.

4.1 The ground rules

Type systems are a huge design space, and we only have space to explore part of it in this paper. In this section we briefly record some design decisions currently embodied in Haskell that we do not propose to meddle with. Our first set of ground rules concern the larger setting:

- We want to retain Haskell's type-inference property.
- We want type inference to be decidable; that is, the compiler must not fail to terminate.
- We want to retain the possibility of separate compilation.
- We want all existing Haskell programs to remain legal, and to have the same meaning.
- We seek a *coherent* type system; that is, every different valid typing derivation for a program leads to a resulting program that has the same dynamic semantics.

The last point needs a little explanation. We have already seen that the way in which context reduction is performed affects the dynamic semantics of the program *via* the construction and use of dictionaries (other operational models will experience similar effects). It is essential that the way in which the typing derivation is constructed (there is usually more than one for a given program) should not affect the meaning of the program.

Next, we give some ground rules about the form of class declarations. A class declaration takes the form:

$$\texttt{class}\ P\ \texttt{=>}\ C\ \alpha_1 \ldots \alpha_n\ \texttt{where}\ \{\ op :: Q\ \texttt{=>}\ \tau\ ;\ \ldots\ \}$$

(If multi-parameter type classes are prohibited, then $n = 1$.) If $S\ \beta_1 \ldots \beta_m$ is one of the constraints appearing in the context P, we say that S is a *superclass* of C. We insist on the following:

- There can be at most one `class` declaration for each class C.
- Throughout the program, all uses of C are applied to n arguments.
- $\alpha_1 \ldots \alpha_n$ must be distinct type variables.
- $TV(P) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. That is, P must not mention any type variables other than the α_i.
- The superclass hierarchy defined by the set of `class` declarations must be acyclic. This restriction is not absolutely necessary, but the applications for cyclic class structures are limited, and it helps to keep things simple.

Next, we give rules governing instance declarations, which have the form:

$$\texttt{instance}\ P\ \texttt{=>}\ C\ \tau_1 \ldots \tau_n\ \texttt{where}\ \ldots$$

We call P the *instance context*, $\tau_1, \ldots, \tau_n$ the *instance types*, and $C\ \tau_1 \ldots \tau_n$ the *head* of the instance declaration. Like Haskell, we insist that:

- $TV(P) \subseteq \bigcup TV(\tau_i)$; that is, the instance context must not mention any type variables that are not mentioned in the instance types.

We discuss the design choices related to instance declarations in Sections 4.5 and 4.7.

Thirdly, we require the following rule for types:

- If P `=>` τ is a type, then $TV(P) \subseteq TV(\tau)$. If the context P mentions any type variables not used in τ then any use of a value with this type is certain to be ambiguous.

Fourthly, we will assume that, despite separate compilation, instance declarations are globally visible. The reason for this is that we want to be able to report an error if we encounter a constraint that cannot match any instance declaration. For example, consider

```
f x = 'c' + x
```

Type inference on `f` gives rise to the constraint `(Num Char)`. If instance declarations are not globally visible, then we would be forced to defer context reduction, in case `f` is called in another module that has an instance declaration for `(Num Char)`. Thus we would have to infer the following type for `f`:

```
f :: Num Char => Char -> Char
```

Instead, what we really want to report an immediate error when type-checking `f`.

So, if instance declarations are not globally visible, many missing-instance errors would only be reported when the main module is compiled, an unacceptable outcome. (Explicit type sigatures might force earlier error reports, however.) Hence our ground rule. In practice, though, we can get away with something a little weaker than insisting that every instance declaration is visible in every module — for example, when compiling a standard library one does need instance declarations for unrelated user-defined types.

Lastly, we have found it useful to articulate the following principle:

- Adding an instance declaration to well-typed program should not alter either the static or dynamic semantics of the program, except that it may give rise to an overlapping-instance-declaration error (in systems that prohibit overlap).

The reason for this principle is to support separate compilation. A separately compiled library module cannot possibly "see" all the instance declarations for all the possible client modules. So it must be the case that these extra instance declarations should not influence the static or dynamic semantics of the library, except if they conflict with the instance declarations used when the library was compiled.

4.2 Decision 1: the form of types

Decision 1: *what limitations, if any, are there on the form of the context of a type?* In Haskell 1.4, types (whether inferred, or specified in a type signature) must be of the form P `=>` τ, where P is a *simple context.* We say that a context is simple if all its constraints are of the form $C\ \alpha$, where C is a class and α is a type variable.

This design decision was defensible for Haskell 1.2 (which lacked constructor classes) but seems demonstrably wrong for Haskell 1.4. For example, consider the definition:

```
g = \xs -> (map not xs) == xs
```

The right hand side of the definition has the type `f Bool -> Bool`, and context {`Functor f`, `Eq (f Bool)`}[6]. Because of the second constraint here, this cannot be reduced to a simple context by the rules in Figure 1, and Haskell 1.4 rejects this definition as ill-typed. In fact, if we insist that the context in a type must be simple, the function `g` has many legal types (such as `[Bool] -> Bool`), but no *principal*, or most general, type. If, instead, we allow non-simple contexts in types, then it has the perfectly sensible principal type:

```
g :: (Functor f, Eq (f Bool)) => f Bool -> Bool
```

In short, Haskell 1.4 lacks the principal type property, namely that any typable expression has a principal type; but it can be regained by allowing richer contexts in types. This is not just a theoretical nicety — it directly affects the expressiveness of the language.

[6] The definition of the class `Functor` was given in Section 3.6.1.

Similar problems occur with multi-parameter classes if we insist that the arguments of each constraint in a context must be variables — a natural generalization of the single-parameter notion of a simple context. For example, one can imagine inferring a context such as {`StateMonad IO` α}, where α is a type variable. If we then want to generalise over α, we would obtain a function whose type was of the form `StateMonad IO` α `=>` τ. If such a type was illegal then, as with the previous example, we would be forced to reject the program even though it has a sensible principal type in a slightly richer system.

The choices for the allowable contexts in types seem to be these:

Choice 1a (Haskell): the context of a type must be simple (with some extended definition of "simple").

Choice 1b (Gofer): there are no restrictions on the context of a type.

Choice 1c: something in between these two. For example, we might insist that the context in a type is reduced "as much as possible". But then a legal type signature might become illegal if we introduced a new instance declaration (because then the type signature might no longer be reduced as much as possible).

4.3 Decision 2: How much context reduction?

Decision 2: *how much context reduction should be done before generalisation?* Haskell and Gofer make very different choices here. Haskell takes an eager approach to context reduction, doing as much as possible before generalisation, while Gofer takes a lazy approach, only using context reduction to eliminate tautological constraints.

It turns out that this choice has a whole raft of consequences, as Jones [1994, Chapter 7] discusses in detail. These consequences mainly concern pragmatic matters, such as the complexity of types, or the efficiency of the resulting program. *It is highly desirable that the choice of how much context reduction is done when should not affect the meaning of the program.* It is bad enough that the meaning of the program inevitably depends on the resolution of overloading (Odersky, Wadler & Wehr [1995]). It would be much worse if the program's meaning depended on the exact *way* in which the overloading was resolved — that is, if the type system were incoherent (Section 4.1).

Here, then, are the issues affecting context reduction.

1. *Context reduction usually leads to "simpler" contexts*, which are perhaps more readily understood (and written) by the programmer. In our earlier example, `Ord a` is simpler than {`Ord a`, `Eq a`, `Eq [a]`}.

   Occasionally, however, a "simpler" context might be less "natural". Suppose we have a data type `Set` with an operation `union`, and an `Ord` instance (Jones [1994, Section 7.1]):

```
data Set a = ...

union :: Eq a => Set a -> Set a -> Set a

instance Eq a => Ord (Set a) where ...
```

Now, consider the following function definition:

```
f x y = if (x<=y) then y else x `union` y
```

With context reduction, `f`'s type is inferred to be

```
f :: Eq a => Set a -> Set a -> Set a
```

whereas without context reduction we would infer

```
f :: Ord (Set a) => Set a -> Set a -> Set a
```

One can argue that the latter is more "natural" since it is clear where the `Ord` constraint comes from, while the former contains a slightly surprising `Eq` constraint that results from the unrelated instance declaration.

2. *Context reduction often, but not always, reduces the number of dictionaries passed to functions.* In the running example of Section 3, doing context reduction before generalisation allowed us to pass one dictionary to `f` instead of three.

   Sometimes, though, a "simpler" context might have more constraints (i.e. more dictionaries to pass in a dictionary-passing implementation). For example, given the instance declaration:

   ```
   instance (Eq a, Eq b) => Eq (a,b) where ...
   ```

   the constraint `Eq (a,b)` would reduce to {`Eq a`,`Eq b`}, which may be "simpler", but certainly is not shorter.

3. *Context reduction eliminates tautological constraints.* For example, without context reduction the function

   ```
   double = \x -> x + (x::Int)
   ```

   would get the type

   ```
   double :: Num Int => Int -> Int
   ```

   This type means that a dictionary for `Num Int` will be passed to `double`, which is quite redundant. It it invariably better to reduce {`Num Int`} to {}, using the `Int` instance of `Num`. The "evidence" that `Int` is an instance of `Num` takes the form of a global constant dictionary for `Num Int`. (This example uses a ground constraint, but the same reasoning applies to any tautological constraint.)

4. *Delaying context reduction increases sharing of dictionaries.* Consider this example:

   ```
   let
     f xs y = xs > [y]
   in
   f xs y && f xs z
   ```

   Haskell will infer the type of `f` to be:

   ```
   f :: Ord a => [a] -> a -> Bool
   ```

   A dictionary for `Ord a` will be passed to `f`, which will construct a dictionary for `Ord [a]`. In this example, though, `f` is called twice, at the same type, and the two calls will independently construct the same `Ord [a]` dictionary. We could obtain more sharing (i.e. efficiency) by postponing the context reduction, inferring instead the following type for `f`:

   ```
   f :: Ord [a] => [a] -> a -> Bool
   ```

   Now `f` is passed a dictionary for `Ord [a]`, and this dictionary can be shared between the two calls of `f`.

   Because context reduction is postponed until the top level in Gofer, this sharing can encompass the whole program, and *only one dictionary for each class/type combination is ever constructed.*

5. *Type signatures interact with context reduction.* Haskell allows us to specify a type signature for a function. Depending on how context reduction is done, and what contexts are allowed in type signatures, this type might be more or less reduced than the inferred type. For example, if full context reduction is normally done before generalisation, then is this a valid type signature?

   ```
   f :: Eq [a] => ...
   ```

   That is, can a type signature decrease the amount of context reduction that is performed? In the other direction, if context reduction is *not* usually done at generalisation, then is this a valid type signature?

   ```
   f :: Eq a => ...
   ```

   where `f`'s right-hand side generates a constraint `Eq [a]`? That is, can a type signature increase the amount of context reduction that is performed?

6. *Context reduction is necessary for polymorphic recursion.* One of the new features in Haskell 1.4 is the ability to define a recursive function in which the recursive call is at a different type than the original call, a feature that has proved itself useful in the efficient encoding of functional data structures (Okasaki [1996]).

   For example, consider the following non-uniformly recursive function:

   ```
   f :: Eq a => a -> a -> Bool
   f x y = if x == y then True
             else f [x] [y]
   ```

   It is not possible to avoid all runtime dictionary construction in this example, because each call to recursive `f` must use a dictionary of higher type, and there is no static bound to the depth of recursion. It follows that the strategy of defering all context reduction to the top level, thereby ensuring a finite number of dictionaries, cannot work. The type signature is necessary for the type checker to permit polymorphic recursion, and it in turn forces reduction of the constraint `Eq [a]` that arises from the recursive call to `f`.

7. *Context reduction affects typability.* Consider the following (contrived) program:

```
data Tree a = Nil | Fork (Tree a) (Tree a)

f x = let silly y = (y==Nil)
      in  x + 1
```

If there is no `Eq` instance of `Tree`, then the program is arguably erroneous, since `silly` performs equality at type `Tree`. But if context reduction is deferred, `silly` will, without complaint, be assigned the type

```
silly :: Eq (Tree a) => a -> Bool
```

Then, since `silly` is never called, no other type error will result. In short, the definition of which programs are typable and which are not depends on the rules for context reduction.

8. *Context reduction conflicts with the use of overlapping instances.* This is a bigger topic, and we defer it until Section 4.4.

Bearing in mind this (amazingly large) set of issues, there seem to be the following possible choices:

Choice 2a (Haskell, eager): reduce every context to a simple context before generalisation. However, as we have seen, this may mean that some perfectly reasonable programs are rejected as being ill-typed.

Choice 2b (lazy): do no context reduction at all until the constraints for the whole program are gathered together; then reduce them. This is satisfyingly decisive, but it gives rise to pretty stupid types, such as:

```
(Eq a, Eq a, Eq a) => a -> Bool
(Num Int, Show Int) => Int -> String
```

Choice 2c (Gofer, fairly lazy): do context reduction before generalisation, but refrain from using rule (*inst*) except for tautological constraints. If overlapping instances are permitted, then change "tautological" to "ground". A variant would be to refrain from using (*super*) as well.

Choice 2d (Gofer + polymorphic recursion): like 2c, but with the added rule that if there is a type signature, the inferred context must be entailed by the context in the type signature, and the variable being defined is assigned the type in the signature throughout its scope. This is enough to make the choice compatible with polymorphic recursion, which 2c is not.

Choice 2e (relaxed): leave it un-specified how much context reduction is done before generalisation! That is, if the actual context of the term to be generalised is P, then the inferred context for the generalised term is P or any context that P reduces to. The same rule for type signatures must apply as in 2d, for the same reason. To avoid the problem of item 7 we can require that an error is reported as soon as a generalisation step encounters a constraint that cannot possibly be satisfied (even if that constraint is not reduced).

We should note that 2b-e rule out Choice 1a for type signatures. Furthermore (as we shall see in Section 4.4), Choices 2a and 2e rule out overlapping instance declarations.

The intent in Choice 2e is to leave as much flexibility as possible to the compiler (so that it can make the most efficient choice) while still giving a well-defined static and dynamic semantics for the language:

- So far as the static semantics is concerned, when context reduction is performed does not change the set of typable programs.
- Concerning the dynamic semantics, in the absence of overlapping instance declarations, a given constraint can only match a unique instance declaration.

4.4 Decision 3: overlapping instance declarations

Decision 3: *are instance declarations with overlapping (but not identical) instance types permitted? (See Section 3.6.)*

If overlapping instances are permitted, we need a rule that specifies which instance declaration to choose if more than one matches a particular constraint. Gofer's rule is that the declaration that matches most closely is chosen. In general, there may not be a unique such instance declaration, so further rules are required to disambiguate the choice — for example, Gofer requires that instance declarations may only overlap if one is a substitution instance of the other.

Unfortunately, this is not enough. As we mentioned above, there is a fundamental conflict between eager (or unspecified) context reduction and the use of overlapping instances. To see this, consider the definition:

```
let
  f x = myShow (x++x)
in
(f "c", f [True,False])
```

where `myShow` was defined in Section 3.6. If we do (full) context reduction before generalising `f`, we will be faced with a constraint `MyShow [a]`, arising from the use of `myShow`. Under eager context reduction we must simplify it, presumably using the instance declaration for `MyShow [a]`, to obtain the type

```
f :: MyShow a => a -> String
```

If we do so, then every call to `f` will be committed to the `myShow1` method. However, suppose that we first perform a simple program transformation, inlining `f` at both its call sites, to obtain the expression:

```
(myShow "c", myShow [True,False] [True,False])
```

Now the two calls distinct calls to `myShow` will lead to the constraints `MyShow [Char]` and `MyShow [Bool]` respectively; the first will lead to a call of `myShow2` while second will lead to a call of `myShow1`. A simple program transformation has changed the behaviour of the program!

Now consider the original program again. If instead we *deferred* context reduction we would infer the type:

```
f :: MyShow [a] => a -> String
```

Now the two calls to `f` will lead to the constraints `MyShow [Char]` and `MyShow [Bool]` as in the inlined case, leading to calls to `myShow2` and `myShow1` respectively. In short, eager context reduction in the presence of overlapping instance declarations can lead to premature committment to a particular instance declaration, and consequential loss of simple source-language program transformations.

Overlapping instances are also incompatible with the reduction of non-ground tautological constraints. For example, suppose we have the declaration

```
instance Monad (ST s) where ...
```

and we are trying to simplify the context {`Monad` (`ST` τ)}. It would be wrong to reduce it to {} because there might be an overlapping instance declaration

```
instance Monad (ST Int) where ...
```

This inability to simplify non-ground tautological constraints has, in practice, caused Gofer some difficulties when implementing lazy state threads (Launchbury & Peyton Jones [1995]). Briefly, `runST` insists that its argument has type $\forall\alpha.$`ST` α τ, while the argument type would be inferred to be `Monad` (`ST` α) `=>` `ST` α τ.

To summarise, if overlapping instances are permitted, then the meaning of the program depends in detail on when context reduction takes place. To avoid loss of coherence, we must specify when context reduction takes place as part of the type system itself.

One possibility is to defer reduction of any constraint that can possibly match more than one instance declaration. That restores the ability to perform program transformations, but it interacts poorly with separate compilation. A separately-compiled library might not "see" all the instances of a given class that a client module uses, and so must conservatively assume that no context reduction can be done at all on any constraint involving a type variable.

So the only reasonable choices are these:

Choice 3a: prohibit overlapping instance declarations.

Choice 3b: permit instance declarations with overlapping, but not identical, instance types, provided one is a substitution instance of the other; but restrict all uses of the (*inst*) rule (Figure 1) to ground contexts C, P. This condition identifies constraints that can match at most one instance declaration, regardless of what further instance declarations are added.

4.5 Decision 4: instance types

Decision 4: *in the instance declaration*

$$\texttt{instance } P \texttt{ => } C\ \tau_1 \ldots \tau_n \texttt{ where } \ldots$$

what limitations, if any, are there on the form of the instance types, $\tau_1 \ldots \tau_n$*?*

Haskell 1.4 has only single-parameter type classes, hence $n = 1$. Furthermore, Haskell insists that the single type τ is a *simple type*; that is, a type of the form $T\ \alpha_1 \ldots \alpha_m$, where T is a type constructor and $\alpha_1 \ldots \alpha_n$ are distinct type variables. This decision is closely bound up with Haskell's restriction to simple contexts in types (Section 4.2). Why? Because, faced with a constraint of the form $(C\ (T\ \tau))$ there is either a unique instance declaration that matches it (in which case the constraint can be reduced), or there is not (in which case an error can be signaled). If τ were allowed to be other than a type variable then more than one instance declaration might be a potential match for the constraint. For example, suppose we had:

```
instance Foo (Tree Int) where ...
instance Foo (Tree Bool) where ...
```

(Note that these two do not overlap.) Given the constraint (`Foo` (`Tree` α)), for some type variable α, we cannot decide which instance declaration to use until we know more about α. If we are generalising over α, we will therefore end up with a function whose type is of the form

$$\texttt{Foo (Tree } \alpha\texttt{) => } \tau$$

Since Haskell does not allow such types (because the context is not simple), it makes sense for Haskell also to restrict instance types to be simple types. If types can have more general contexts, however, it is not clear that such a restriction makes sense.

We have come across examples where it makes sense for the instance types not to be simple types. Section 3.6.1 gave examples in which the instance type was just a type variable, although this was in the context of overlapping instance declarations. Here is another example[7]:

```
class Liftable f where
  lift0 :: a -> f a
  lift1 :: (a->b) -> f a -> f b
  lift2 :: (a->b->c) -> f a -> f b -> f c

instance (Liftable f, Num a) => Num (f a) where
  fromInteger = lift0 . fromInteger
  negate      = lift1 negate
  (+)         = lift2 (+)
```

The instance declaration is entirely reasonable: it says that any "liftable" type constructor `f` can be used to construct a new numeric type `(f a)` from an existing numeric type `a`. Indeed, these declarations precisely generalises the `Behaviour` class of Elliott & Hudak [1997], and we have encountered other examples of the same pattern. (You will probably have noticed that `lift1` is just the `map` from the class `Functor`; perhaps `Functor` should be a superclass of `Liftable`.) A disadvantage of `Liftable` is that now the Haskell types for `Complex` and `Ratio` must be made instances of `Num` indirectly, by making them instances of `Liftable`. This seems to work fine for `Complex`, but not for `Ratio`. Incidentally, we could overcome this problem if we had overlapping instances, thus:

```
instance (Liftable f, Num a) => Num (f a) where ...
instance Num a => Num (Ratio a) where ...
```

Another reason for wanting non-simple instance types is

[7] Suggested by John Matthews.

when using old types for new purposes. For example[8], suppose we want to define the class of moveable things:

```
class Moveable t where
  move :: Vector -> t -> t
```

Now let us make points moveable. What is a point? Perhaps just a pair of `Float`s. So we might want to write

```
instance Moveable (Float, Float) where ...
```

or even

```
type Point = (Float, Float)
instance Moveable Point where ...
```

Unlike the `Liftable` example, it is possible to manage with simple instance types, by making `Point` a new type:

```
newtype Point = MkPoint Float Float
instance Moveable Point where ...
```

but that might be tiresome (for example, `unzip` split a list of points into their x-coordinates and y-coordinates).

Choice 4a (Haskell): the instance type(s) τ_i must all be simple types.

Choice 4b: each of the instance types τ_i is a simple type or a type variable, and at least one is not a type variable. (The latter restriction is necessary to ensure that context reduction terminates.)

Choice 4c: at least one of the instance types τ_i must not be a type variable.

Choice 4c would permit the `Liftable` example above. It would also permit the following instance declarations

```
instance D (T Int a) where ...
instance D (T Bool a) where ...
```

even if overlapping instances are prohibited (provided, of course, there was no instance for `D (T a b)`). It would also allow strange-looking instance declarations such as

```
instance C [[a -> Int]] where ...
```

which in turn make the matching of a candidate instance declaration against a constraint a little more complicated (although not much).

If overlapping instances are permitted, then it is not clear whether choices 4b and 4c lead to a decideable type system. If overlapping instances are not permitted then, seem to be no technical objections to them, and the examples given above suggest that the extra expressiveness is useful.

4.6 Decision 5: repeated type variables in instance heads

Decision 5: *in the instance declaration*

instance P => $C\ \tau_1 \ldots \tau_n$ where ...

can the instance head τ_i contain repeated type variables? This decision is really part of Decision 4 but it deserves separate treatment.

Consider this instance declaration, which has a repeated type variable in the instance type:

```
instance ... => Foo (a,a) where ...
```

In Haskell this is illegal, but there seems no technical reason to exclude it. Furthermore, it is useful: the `VarMonad` instance for `ST` in Section 2.1 used repeated type variables, as did the `Iso` example in Section 2.3.

Permitting repeated type variables in the instance type of an instance declaration slightly complicates the process of matching a candidate instance declaration against a constraint, requiring full matching (i.e. one-way unification, a well-understood algorithm). For example, when matching the instance head `Foo` (α,α) against a constraint `Foo` (τ_1,τ_2) one must first bind α to τ_1, and then check for equality between the now-bound α and τ_2.

Choice 5a: permit repeated type variables in an instance head.

Choice 5b: prohibit repeated type variables in an instance head.

4.7 Decision 6: instance contexts

Decision 6: *in the instance declaration*

instance P => $C\ \tau_1 \ldots \tau_n$ where ...

what limitations, if any, are there on the form of the instance context, P?

As mentioned in Section 4.1, we require that $TV(P) \subseteq \bigcup TV(\tau_i)$. However, Haskell has a more drastic restriction: it requires that each constraint in P be of the form $C\ \alpha$ where α is a type variable. An important motivation for a restriction of this sort is the need to ensure termination of context reduction. For example, suppose the following declaration was allowed:

```
instance C [[a]] => C [a] where ...
```

The trouble here is that for context reduction to terminate it must reduce a context to a *simpler* context. This instance declaration will "reduce" the constraint (`C [`τ`]`) to (`C [[`τ`]]`), which is more complicated, and context reduction will diverge. Although they do not seem to occur in practical applications, instance declarations like this are permitted in Gofer—with the consequence that its type system is in fact undecidable.

In short, it is essential to place enough constraints on the instance context to ensure that context reduction converges. To do this, we need to ensure that something "gets smaller" in the passage from $C\ \tau_1 \ldots \tau_n$ to P. Haskell's restriction to simple contexts certainly ensures termination, because the argument types are guaranteed to get smaller. In principle, instance declarations with irreducible but non-simple contexts might make sense:

[8] Suggested by Simon Thompson.

```
instance Monad (t m) => Foo t m where ...
```

We have yet to find any convincing examples of this. However, if context reduction is deferred (Choices 2b,c) then we *must* permit non-simple instance contexts. For example:

```
data Tree a = Node a [Tree a]
instance (Eq a, Eq [Tree a]) => Eq (Tree a) where
  (==) (Node v1 ts1) (Node v2 ts2)
        = (v1 == v2) && (ts1 == ts2)
```

Here, if we are not permitted to reduce the constraint `Eq [Tree a]`, it must appear in the instance context.

Lastly, if the constraints in P involve only type variables, when multi-parameter type classes are involved we must also ask whether a single constraint may contain a repeated type variable, thus:

```
instance Foo a a => Baz a where ...
```

There seems to be no technical reason to prohibit this.

Choice 6a: constraints in the context of an instance declaration must be of the form $C\ \alpha_1 \ldots \alpha_n$, with the α_i distinct.

Choice 6b: as for Choice 6a, except without the requirement for the α_i to be distinct.

Choice 6c: something less restrictive, but with some way of ensuring decidability of context reduction.

4.8 Decision 7: what superclasses are permitted

Decision 7: *in a class declaration,*

`class` P `=>` $C\ \alpha_1 \ldots \alpha_n$ `where {` $op :: Q$ `=>` $\tau\ ; \ldots$ `}`

what limitations, beyond those in Section 4.1, are there on the form of the superclass context, P? Haskell restricts P to consist of constraints of the form $D\ \beta_1 \ldots \beta_m$, where β_i must be a member of $\{\alpha_1, \ldots, \alpha_n\}$, and all the β_i must be distinct. But what is wrong with this?

```
class Foo (t m) => Baz t m where ...
```

Also in this case, there seems to be no technical reason to prohibit this.

Choice 7a: constraints in the superclass context must be as in Haskell, i.e. the constraints are of the form $D\ \alpha_1 \ldots \alpha_n$, with the α_i distinct, and a subset of the type variables that occur in the class head.

Choice 7b: no limitations on superclass contexts, except those postulated in Section 4.1.

4.9 Decision 8: improvement

Suppose that we have a constraint with the following properties:

- it contains free type variables;
- it does not match any instance declaration[9]
- it can be made to match an instance declaration by instantiating some of the constraint's free type variables;
- no matter what other (legal) instance declarations are added, there is only one instance declaration that the constraint can be made to match in this way.

If all these things are true, an attractive idea is to *improve* the constraint by instantiating the type variables in the constraint so that it does match the instance declaration. This makes some programs typable that would not otherwise be so. It does not compromise any of our principles, because the last condition ensures that even adding new instance declarations will not change the way in which improvement is carried out.

Improvement was introduced by Jones [1995b]. A full discussion is beyond the scope of this paper. The conditions are quite restrictive, so it is not yet clear whether it would improve enough useful programs to be worth the extra effort.

Choice 8a: no improvement.

Choice 8b: allow improvement in some form.

Choice 8b would obviously need further elaboration before this design decision is crisply formulated.

4.10 Decision 9: Class declarations

Decision 9: *what limitations, if any, are there on the contexts in class-member type signatures?* Presumably class-member type signatures should obey the same rules as any other type signature, but Haskell adds an additional restriction. Consider:

```
class C a where
  op1 :: a -> a
  op2 :: Eq a => a -> a
```

In Haskell, the type signature for `op2` would be illegal, because it further constrains the class type variable `a`. There seems to be no technical reason for this restriction. It is simply a nuisance to the Haskell specification, implementation, and (occasionally) programmer.

Choice 9a (Haskell): the context in a class-member type signature cannot mention the class type variable; in addition, it is subject to the same rules as any other type signature.

Choice 9b: the type signature for a class-member is subject to the same rules as any other type signature.

[9] Recall that matching a constraint against an instance declaration is a one-way unification: we may instantiate type variables from the instance head, but not those from the constraint.

5 Other avenues

While writing this paper, a number of other extensions to Haskell's type-class system were suggested to us that seem to raise considerable technical difficulties. We enumerate them in this section, identifying their difficulties.

5.1 Anonymous type synonyms

When exposed to multi-parameter type classes and in particular higher order type variables, programmers often seek a more expressive type language. For example, suppose we have the following two classes `Foo` and `Bar`:

```
class Foo k1 where f :: k1 a -> a
class Bar k2 where g :: k2 b -> b
```

and a concrete binary type constructor

```
data Baz a b = ...
```

Then we can easily write an instance declaration that declares `(Bar a)` to be a functor, thus:

```
instance Functor (Baz a) where
      map = ...
```

But suppose `Baz` is really a functor in its *first* argument. Then we really want to say is:

```
type Zab b a = Baz a b
instance Functor (Zab b) where
      map = ...
```

However, Haskell prohibits partially-applied type synonmyms, and for a very good reason: a partially-applied type synonym is, in effect, a lambda abstraction at the type level, and that takes us immediately into the realm of higher-order unification, and minimises the likelihood of a decidable type system (Jones [1995a, Section 4.2]). It might be possible to incorporate some form of higher-order unification (e.g. along the lines of Miller [1991]) but it would be a substantial new complication to an already sophisticated type system.

5.2 Relaxed superclass contexts

One of our ground rules in this paper is that the type variables in the context of a class declaration must be a subset of the type variables in the class head. This rules out declarations like:

```
class Monad (m s) => StateMonad m where
    get :: m s s
    set :: s -> m s ()
```

The idea here is that the context indicates that `m s` should be a monad for any type `s`. Rewriting this definition by overloading on the state as well

```
class Monad (m s) => StateMonad m s where
    get :: m s s
    set :: s -> m s ()
```

is not satisfactory as it forces us to pass several dictionaries, say `(StateMonad State Int, StateMonad State Bool)` where they are really the same. What we really want is to use universal quantification:

```
class (forall s. Monad (m s))
   => StateMonad m where
    get :: m s s
    set :: s -> m s ()
```

but that means that the type system would have to handle constraints with universal quantification — a substantial complication.

Another ground rule in this paper is the restriction to acyclic superclass hierarchies. Gofer puts no restriction on the form of predicates that may appear in superclass contexts, in particular it allows mutually recursive class hierarchies. For example, the `Iso` class example of Section 2.3 can be written in a more elegant way if we allow recursive classes:

```
class Iso b a => Iso a b where iso :: a -> b
```

The superclass constraint ensures that when a type `a` is isomorphic to `b`, then type `b` is isomorphic to `a`. Needless to say that such class declarations easily give lead to an undecidable type system.

5.3 Controling the scope of instances

One sometimes wishes that it was possible to have more than one instance declaration for the *same* instance type, an extreme case of overlap. For example, in one part of the program one might like to have an instance declaration

```
instance Ord T where { (<) = lessThanT }
```

and elsewhere one might like

```
instance Ord T where { (<) = greaterThanT }
```

As evidence for this, notice that several Haskell standard library functions (such as `sortBy`) take an explicit comparison operator as an argument, reflecting the fact that the `Ord` instance for the data type involved might not be the ordering you want for the sort. Having multiple instance declarations for the same type is, however, fraught with the risk of losing coherence; at the very least it involves strict control over which instance declarations are visible where. It is far from obvious that controlling the scope of instances is the right way to tackle this problem — functors, as in ML, look more appropriate.

5.4 Relaxed type signature contexts

In programming with type classes it is often the case that we end up with an ambiguous type while we know that in fact it is harmless. For example, knowing all instance declarations in the program, we might be sure that the ambiguous example of Section 2.3 `iso 2 == iso 3 :: (Eq b, Iso Int b) => Bool` has the same value, irrespective of the choice for `b`. Is it possible to modify the type system to deal with such cases?

6 Conclusion

Sometimes a type system is so finely balanced that virtually any extension destroys some of its more desirable properties. Haskell's type class system turns out not to have the property – there seems to be sensible extensions that gain expressiveness without involving major new complications.

We have tried to summarise the design choices in a fairly un-biased manner, but it is time to nail our colours to the mast. The following set of design choices seems to define an upward-compatible extension of Haskell without losing anything important:

- Permit multi-parameter type classes.
- Permit arbitrary constraints in types and type signatures (Choice 1b).
- Use the (*inst*) context-reduction rule only when forced by a type signature, or when the constraint is tautological (Choice 2d). Choice 2e is also viable.
- Prohibit overlapping instance declarations (Choice 3a).
- Permit arbitrary instance types in the head of an instance declaration, except that at least one must not be a type variable (Choice 4c).
- Permit repeated type variables in the head of an instance declaration (Choice 5a).
- Restrict the context of an instance declaration to mention type variables only (Choice 6b).
- No limitations on superclass contexts (Choice 7b).
- Prohibit improvement (Choice 8a).
- Permit the class variable(s) to be constrained in class-member type signatures (Choice 9b).

Our hope is that this paper will provoke some well-informed debate about possible extensions to Haskell's type classes. We particularly seek a wider range of examples to illustrate and motivate the various extensions discussed here.

Acknowledegements

We would like to thank Koen Claessen, Benedict Gaster, Thomas Hallgren, John Matthews, Sergey Mechveliani, Alastair Reid, Enno Scholz, Walid Taha, Simon Thompson, and Carl Witty for helpful feedback on earlier drafts of this paper. Meijer and Peyton Jones also gratefully acknowledge the support of the Oregon Graduate Institute during our sabbaticals, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References

K Chen, P Hudak & M Odersky [June 1992], "Parametric type classes," in *ACM Symposium on Lisp and Functional Programming, Snowbird*, ACM.

C Elliott & P Hudak [June 1997], "Functional reactive animation," in *Proc International Conference on Functional Programming, Amsterdam*, ACM.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27.

MP Jones [Jan 1995a], "A system of constructor classes: overloading and implicit higher-order polymorphism," *Journal of Functional Programming* 5, 1–36.

MP Jones [June 1995b], "Simplifying and improving qualified types," in *Proc Functional Programming Languages and Computer Architecture, La Jolla*, ACM.

MP Jones [May 1994], "The implementation of the Gofer functional programming system," YALEU/DCS/RR-1030, Department of Computer Science, Yale University.

MP Jones [May 1995], "Functional programming with overloading and higher-order polymorphism," in *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, Springer-Verlag LNCS 925.

MP Jones [Nov 1994], *Qualified types: theory and practice*, Cambridge University Press.

S Kaes [Jan 1988], "Parametric overloading in polymorphic programming languages," in *15th ACM Symposium on Principles of Programming Languages*, ACM, 131–144.

J Launchbury & SL Peyton Jones [Dec 1995], "State in Haskell," *Lisp and Symbolic Computation* 8, 293–342.

S Liang, P Hudak & M Jones [Jan 1995], "Monad transformers and modular interpreters," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM.

D Miller [1991], "A logic programming language with lambda abstraction, function variables, and simple unification," *Journal of Logic and Computation* 1.

M Odersky, PL Wadler & M Wehr [June 1995], "A second look at overloading," in *Proc Functional Programming Languages and Computer Architecture, La Jolla*, ACM.

C Okasaki [Sept 1996], "Purely functional data structures," PhD thesis, CMU-CS-96-177, Department of Computer Science, Carnegie Mellon University.

SL Peyton Jones [Sept 1996], "Bulk types with class," in *Electronic proceedings of the 1996 Glasgow Functional Programming Workshop* (`http://-www.dcs.gla.ac.uk/fp/workshops/fpw96/-Proceedings96.html`).

SL Peyton Jones, AJ Gordon & SO Finne [Jan 1996], "Concurrent Haskell," in *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, ACM, 295–308.

PL Wadler & S Blott [Jan 1989], "How to make ad-hoc polymorphism less ad hoc," in *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, ACM.