Restricted Data Types in Haskell

John Hughes

September 4, 1999

Abstract

The implementations of abstract type constructors must often restrict the type parameters: for example, one implementation of sets may require equality on the element type, while another implementation requires an ordering. Haskell has no mechanism to abstract over such restrictions, which can hinder us from replacing one implementation by another, or making several implementations instances of the same class. This paper proposes a language extension called *restricted data types* to address this problem.

A restricted data type definition specifies a condition which argument types must satisfy for the data type to be *well-formed*. Every type in a program must be well-formed, and we add an explicit notation to express such requirements. Thus programmers can simply state that a type must be well-formed, rather than repeat its restriction explicitly.

We explain our extension via a simulation using multi-parameter classes, which serves to specify its semantics. We show its application to the design of a collection class and to the class of monads, and we discuss extensions to compile-time context reduction needed to implement it.

1 Introduction

Suppose you are designing an abstract data type of sets, represented for example by lists:

```
data Set a = Set [a]
```

When you implement the methods, you are likely to need to make assumptions about the element type `a`. For example, a function to test for membership will need to test elements for equality, and its type will reflect this:

```
member :: Eq a => a -> Set a -> Bool
```

Should you later decide to change the representation of sets, for example to use ordered binary trees for greater efficiency, then the restrictions on the element type will change. For example, the type of `member` will become

```
member :: Ord a => a -> Set a -> Bool
```

Thus a change in the *representation* of the abstract data type is reflected by a change in the *interface* which it provides.

Such a change in the interface has unfortunate consequences. When the functions which implement sets change their types, so in general will functions which use them in the rest of the program. Explicit type signatures spread throughout the program will therefore need to be changed, even though the definitions they are attached to are unaffected[1]. If there are many type signatures, either for stylistic

[1] Assuming, of course, that all the sets used in the program have elements that do actually support an ordering.

reasons or because Haskell's infamous monomorphism restriction forced their insertion, then the work of revising them may dominate that of modifying the `Set` module itself. In the worst case, the programmer may even be dissuaded from making a desirable change in one module, because of all the consequential changes that must be made to type signatures elsewhere.

An even more acute problem arises if we try to define the interface of an abstract data type as a Haskell class, so that several different implementations can be provided as instances. For example, we might wish to define a class `Collection` whose instances are lists, ordered binary trees, hash tables, etc.

```
class Collection c where
  ...
  member :: ... => a -> c a -> Bool
  ...
```

But now, what should the type of the `member` function be in the class definition? We cannot know what requirements to place on the type of collection elements, because these requirements differ from instance to instance. As a result, we cannot write an appropriate class definition at all!

The main idea of this paper is to restrict the parameters of abstract data types *when we define the type*, rather than when we define the methods. We thus define sets represented by lists as follows

```
data Eq a => Set a = Set [a]
```

with the interpretation that types `Set t` are well-formed only if `t` supports equality[2]. We call types defined in this way *restricted data types*. Now, since we state in the *type* definition that the elements must support equality, it should no longer be necessary to state it in the types of the *methods*. For example, we might now give the `member` function the type

```
member :: a -> Set a -> Bool
```

It is clear that `member` can only be used at types that support equality, because `Set a` occurs in its type signature.

Now, if the implementation of sets is changed to ordered binary trees, then the new constraint on the element type need appear only on the type definition; the types of the methods remain unchanged. Consequently the problems discussed in this section disappear: the implementations of abstract datatypes can be changed without affecting type signatures in the rest of the program, and different implementations of the same abstract datatype can be made instances of the same class, even if they place different restrictions on the type parameters.

While the basic idea of a restricted data type is very intuitive, the details of the design are surprisingly subtle, and the implementation is even subtler. We will therefore focus on *simulating* restricted data types in Haskell (extended with multi-parameter classes). This simulation has been tested using `hugs98` with the `-98` flag. However, the simulation is a little tedious to use in practice, and so at the end of the paper we will propose a language extension whose semantics (and a possible implementation) is given by the simulation we describe.

2 Simulating Restricted Data Types: A Collection Class

We shall explain our idea with reference to a simplified `Collection` class, which might be defined using restricted data types as follows:

[2] Haskell already supports this syntax, but with a much weaker meaning.

```
class Collection c where
  empty :: c a
  singleton :: a -> c a
  union :: c a -> c a -> c a
  member :: a -> c a -> Bool
```

We will show how to simulate restricted data types, so that both sets and ordered lists can be made instances of this class.

Of course, the intention of this class definition is that the element type `a` is implicitly constrained to satisfy the restriction of the data type `c`. To simulate this in Haskell without restricted data types, we must declare a `Collection` class whose methods *do* explicitly restrict the element type, but we must parameterise the class definition on the particular restriction concerned, so that different instances can impose different restrictions. If we could parameterise classes on *classes*, then we might write

```
class Collection c cxt where
  empty :: cxt a => c a
  singleton :: cxt a => a -> c a
  union :: cxt a => c a -> c a -> c a
  member :: cxt a => a -> c a -> Bool
```

and declare instances

```
instance Collection Set Eq where ...
instance Collection OrdSet Ord where ...
```

However, Haskell classes can only be parameterised on types. We therefore *represent* class constraints such as `Eq` and `Ord` by a suitable type. It is natural to represent a class by the type of its associated dictionary, and so (simplifying somewhat) we define

```
data EqD a = EqD {eq :: a -> a -> Bool}
data OrdD a = OrdD {le :: a -> a -> Bool, eqOrd :: EqD a}
```

The idea is that `EqD a` contains an implementation of the equality test, while `OrdD a` contains an implementation of `<=` and an equality dictionary (since `Eq` is a superclass of `Ord`).

Now, the constraint `Eq a` is satisfied when we have an implementation of equality available at type `a`, which is equivalent to having a value of type `EqD a` available. We therefore define a class

```
class Sat t where dict :: t
```

which we will use to simulate other constraints. For example, the constraint `Eq a` is simulated by `Sat (EqD a)`; the former is satisfied precisely when we can construct a dictionary to satisfy the latter. We declare

```
instance Eq a => Sat (EqD a) where
  dict = EqD {eq= (==)}
instance Ord a => Sat (OrdD a) where
  dict = OrdD {le= (<=), eqOrd= dict}
```

Now we can redefine the `member` function for sets so that it no longer explicitly refers to the `Eq` class, but instead just requires that an appropriate dictionary exists:

```
member :: Sat (EqD a) => a -> Set a -> Bool
member x (Set xs) = any (eq dict x) xs
```

```
data Set cxt a = Set [a] | Unused (cxt a->()) deriving Show
type SetCxt a = EqD a
type WfSet a = Set SetCxt a

instance Collection Set EqD where
  empty = Set []
  singleton x = Set [x]
  union xset@(Set xs) (Set ys) =
    Set (xs++[y | y<-ys, not (member y xset)])
  member x (Set xs) = any (eq dict x) xs
```

Figure 1: Making `Set` an instance of `Collection`.

Now, at last, we can parameterise the `Collection` class definition both on the type of collections, and on the constraint that elements must satisfy, since both are now represented by types. We might expect to write

```
class Collection c cxt where
  empty :: Sat (cxt a) => c a
  singleton :: Sat (cxt a) => a -> c a
  union :: Sat (cxt a) => c a -> c a -> c a
  member :: Sat (cxt a) => a -> c a -> Bool
```

thus making the appropriate dictionary available in all the methods.

Unfortunately this definition is still not quite right, because the class parameter `cxt` does not appear in the *types* of the methods, and so cannot be inferred when the methods are used. An attempt to use this class would therefore lead to ambiguous overloading. In fact, since we are simulating restricted data types, there is only one possible type `cxt` for each collection type `c`, but compilers cannot know this.

The solution is simply to parameterise collection types on their context, so that each type carries with it the restriction that its elements must satisfy. We rewrite the class definition as

```
class Collection c cxt where
  empty :: Sat (cxt a) => c cxt a
  singleton :: Sat (cxt a) => a -> c cxt a
  union :: Sat (cxt a) => c cxt a -> c cxt a -> c cxt a
  member :: Sat (cxt a) => a -> c cxt a -> Bool
```

and it is now accepted.

We modify the definition of the `Set` type accordingly, and we can then define it as an instance of the generic `Collection` class. The new type and instance definitions appear in figure 1[3]. A similar implementation of collections as ordered lists appears in figure 2.

[3] There is one unpleasant hack in the figure: the constructor `Unused` in the data type definition for `Set`. It is there purely in order to force the compiler to assign the parameter `cxt` the correct kind: without it, `cxt` does not appear at all in the right hand side of the definition, and is therefore assigned the (incorrect) kind `*`. The application `cxt a` forces the correct kind `*->*` to be assigned, and embedding it in the type `cxt a->()` prevents the type of the context from interfering with the derivation of a `Show` instance.

```
data OrdSet cxt a = OrdSet [a] | Unused (cxt a->()) deriving Show
type OrdSetCxt a = OrdD a
type WfOrdSet a = OrdSet OrdSetCxt a

instance Collection OrdSet OrdD where
  empty = OrdSet []
  singleton x = OrdSet [x]
  union (OrdSet xs) (OrdSet ys) = OrdSet (merge xs ys)
    where
      merge [] ys = ys
      merge xs [] = xs
      merge (x:xs) (y:ys) = if eq (eqOrd dict) x y then x:merge xs ys
                            else if le dict x y then x:merge xs (y:ys)
                            else y:merge (x:xs) ys
  member x (OrdSet xs) = any (eq (eqOrd dict) x) xs
```

Figure 2: Making `OrdSet` an instance of `Collection`.

3 Restricted Monads

Restricted data types would be of limited interest if they were useful only for defining collection types, but in fact they are generally useful in connection with constructor classes, classes whose instances are parameterised types. We will discuss one more example, the class of *monads*.

The `Monad` class is defined (slightly simplified) as follows:

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

One interesting monad in the mathematical sense is the type `Set`, but our implementation of `Set` above cannot be made an instance of this class because `(>>=)` requires equality on the set elements, and the type given for `(>>=)` in the class declaration does not provide it. Categorically speaking, `Set` is a monad over a *subcategory* of the category of Haskell types and functions, but Haskell gives us no way to express that.

However, just as we parameterised the `Collection` class on a condition that elements must satisfy, so we can parameterise the `Monad` class in an analogous way. We define

```
class WfMonad m cxt where
  unit :: Sat (cxt a) => a -> m cxt a
  bind :: (Sat (cxt a), Sat (cxt b)) =>
            m cxt a -> (a -> m cxt b) -> m cxt b
```

Notice that the context for `bind` requires that *both* types `m cxt a` and `m cxt b` are well-formed.

Now we can indeed make `Set` an instance of `WfMonad`:

```
instance WfMonad Set EqD where
  unit a = Set [a]
  Set as ‘bind‘ f = foldr union empty (map f as)
```

The `union` operation in `bind` requires equality, which is available since the type `Set EqD b` is well-formed.

Another interesting example is the monad which represents computations as strings:

```
data StringM cxt a = StringM String | Unused (cxt a->())
```

Naturally, we must restrict this monad to types which can be read and shown, so we define a type to represent this constraint:

```
data TextD a = TextD {rd :: String -> a, sh :: a -> String}
instance (Read a, Show a) => Sat (TextD a) where
  dict = TextD {rd= read, sh= show}
```

Now we can make `StringM` an instance of `WfMonad` as follows:

```
instance WfMonad StringM TextD where
  unit x = StringM (sh dict x)
  StringM s ‘bind‘ f = f (rd dict s)
```

This monad is related to a library for CGI programming which I am developing, which saves computations in hidden fields of HTML forms. (In fact, the technique is applied to yet another constructor class, the class of *arrows* [Hug99]).

4 Improving the Simulation

The technique we have discussed certainly lets us use different properties of collection elements in different instances of the collection class, but not really very conveniently. In figure 2, for example, we know that the element type supports an equality operation, but we have to refer to it as `eq (eqOrd dict)`. Of course we would prefer to use the usual symbol `(==)`, not only because it is syntactically more appealing, but also because we could then use other overloaded functions that depend on equality internally. However, using the equality symbol would require a context `Eq a` rather than `Sat (OrdD a)`.

Let us therefore define new instances of `Eq` in terms of `Sat`, which extract the equality function from an available dictionary:

```
instance Sat (EqD a) => Eq a where
  (==) = eq dict

instance Sat (OrdD a) => Sat (EqD a) where
  dict = eqOrd dict
```

Given these definitions, and a similar one for `Ord`, we should be able to use `(==)` and `(<=)` freely in Figures 1 and 2.

Unfortunately, these instance declarations are rejected by Hugs, because they *overlap* with the existing instances of `Eq` and `Sat (EqD a)`. In their exploration of the design space for type classes [JJM97], Peyton-Jones, Jones and Meijer discuss the possibility of allowing overlapping instances, but conclude that it is undesirable to do so.

The main problem is that the meaning of a program may depend on which of two overlapping instances is chosen, and so it is important to specify precisely how the choice is made, but it seems to be very difficult to give a specification which is both simple and precise. However, in this particular case, the meanings of programs do *not* depend on which instance is chosen. There is only one implementation of equality for any particular type; the new overlapping instance declaration above simply provides another way of accessing it. Thus whenever there is a choice of how to obtain an implementation of equality, then we know that the choice does not affect the semantics of the program, and the compiler is free to choose the instance declaration to apply on other grounds, such as efficiency.

However, even if overlapping instances are permitted, they do introduce a risk that type inference may loop. Indeed, Hugs provides a flag to allow instances to overlap, but if it is turned on then the type checker loops given the declarations above. To understand why, we must explain the process of *context reduction.* Suppose a programmer uses an equality test to compare two lists within a polymorphic function. The compiler infers that the comparison is well typed in a context `Eq [a]`, where `a` is the element type. Given an instance declaration `instance Eq a => Eq [a] where ...`, then the compiler can construct an implementation of equality for `[a]` from an implementation for `a`, thus reducing the problem of satisfying `Eq [a]` to the simpler problem of satisfying the context `Eq a`. This process is called context reduction, and is implemented by using instance declarations 'backwards' as rewrite rules on contexts.

Now we can see that if instance declarations overlap, then context reduction becomes non-deterministic. Worse, it may easily loop. In our example, the new instance declaration

```
instance Sat (EqD a) => Eq a
```

enables a context `Eq a` to be reduced to `Sat (EqD a)`, but the instance declaration we gave earlier

```
instance Eq a => Sat (EqD a)
```

enables this to be reduced back to `Eq a` again. Hence context reduction loops. In this case the compiler would need to search for a terminating context reduction, and this is something which existing implementations of overlapping instances do not do.

In general, it may be undecidable whether a path in the tree of possible context reductions is terminating or not. But in our particular case, loops are introduced only via instance declarations involving the `Sat` class, and by inspection, context reduction using these instances does not increase the *depth* of terms in the context. Thus even infinite context reductions using these instances will contain only a finite number of terms. A compiler can abort an attempted context reduction when a term reduced earlier appears again, since such a reduction is never helpful: to discover, for example, that an `Eq a` dictionary can be constructed from an `Eq a` dictionary is pointless. In our application, this strategy will cut off all infinite context reductions, and so type checking will terminate.

To summarise, to make our simulation of restricted datatypes convenient to use we must be able to define overlapping instance. This is normally dangerous, since in general it leads to ill-defined semantics and undecideable type checking, but in our particular application these problems do not arise. We do require, though, that compilers make an easy test to detect and avoid looping context reductions.

5 Other Approaches

The simulation we have described is certainly non-trivial, and of course, similar problems have been solved in other ways before. In this section we will review two other ways of designing a `Collection` class, which could be taken as alternative ways of simulating restricted data types, and we will argue that the approach we take in this paper is superior to both.

5.1 Peyton-Jones' Multiparameter `Collection` Class

Peyton-Jones proposed a different design for a multi-parameter `Collection` class [PJ96]. His idea was to parameterise the class on the type of elements, as well as the type of collections, thus letting instances constrain both. Applying his idea to our simplified class, we would define it as

```
class Collection c a where
  empty :: c a
  singleton :: a -> c a
  union :: c a -> c a -> c a
  member :: a -> c a -> Bool
```

and could now define instances such as

```
instance Eq a => Collection Set a where ...
```

in which the assumption `Eq a` can of course be used in the implementations of the methods.

Peyton-Jones' idea works well when there is a *single* element type which appears in all occurrences of the collection type. But it works much less well when the methods operate on collections of different types. If the `Collection` class included a method for mapping over collections,

```
mapC :: (a -> b) -> c a -> c b
```

then it would be unclear what the parameters of the `Collection` class should be.

- If only one element type appears as a parameter, for example as in

  ```
  class Collection c a where
    ...
    mapC :: (a -> b) -> c a -> c b
  ```

  then instances can constrain only one of `a` and `b`, and so an implementation of `mapC` which required equality on both types could not be made an instance of this class.

- On the other hand, if both type variables are made parameters of the class, as in

  ```
  class Collection c a b where
    ...
    mapC :: (a -> b) -> c a -> c b
  ```

  then any attempt to use the other methods of the class leads to ambiguous overloading, since the variable `b` does not occur in their types.

This problem arises also if we try to use Peyton-Jones' idea to define a restricted monad class, since the type of `return` involves only `m a`, while the type of `(>>=)` involves both `m a` and `m b`; what should the class parameters be?

Our approach, in contrast, works regardless of how many different occurrences of the restricted type constructor there are.

5.2 The ‘Object-Oriented’ Approach

An alternative way to simulate restricted data types is to build in the appropriate dictionary into the objects of the type. For example, if we define

```
data Set a = Set (EqD a) [a]
```

then it is clear that we can manipulate values of type `Set a` without requiring that `Eq a` hold in the context: we can obtain an implementation of equality directly from the `Set` we are working on. For example,

```
member x (Set dict xs) = any (eq dict x) xs
union xset@(Set dict xs) (Set _ ys) =
  Set dict (xs++[y | y<-ys, not (member y xset)])
```

The problem with this approach is that we cannot always guarantee that the arguments of a function will provide a suitable dictionary to construct its result. For example, `empty` and `singleton` construct `Set`s, in which they must place an equality dictionary, but they have no `Set` argument to extract it from. Likewise,

```
mapC :: (a->b) -> Set a -> Set b
```

should place an `EqD b` dictionary in its result, but can obtain only an `EqD a` dictionary from its argument. So none of these functions can be implemented.

The ‘object-oriented’ approach only works under strong restrictions on the types of the methods we want to implement. Our approach, on the other hand, works for all method types.

6 The Case for a Language Extension

We have now argued that restricted data types are useful, and that our simulation of them works better than other proposals. But given that restricted data types can be simulated in several dialects of Haskell already, why make a new language extension? We see three main reasons for doing so.

Firstly, our simulation requires that the designer of a constructor *class* anticipate whether or not it need support restricted datatypes as instances. In the case of a `Collection` class, it is fairly clear that it should, but in the case of the `Monad` class, for example, the class designer may not anticipate the need. The programmer who later wishes to declare `Set` to be a monad is then powerless to do so. But if restricted datatypes are built into the language, then the compiler can transform *all* class definitions appropriately, thus guaranteeing that a restricted datatype can be used anywhere an unrestricted one can.

Secondly, our simulation requires the programmer to declare the types of dictionaries for each class used in a datatype restriction. But these types are constructed internally by the compiler anyway, as part of the compilation of the class mechanism. Building restricted datatypes into the language spares the programmer from the need to duplicate the compiler's work.

Thirdly, to work really well, our simulation requires support for overlapping instances, which are in general a dangerous feature. Yet in our particular application, the overlap is safe. It is better to extend Haskell with a safe feature (restricted datatypes) than with a dangerous feature which can be used to simulate it.

We propose the following extension therefore. We introduce a new kind of context, `wft t`, to mean that the type `t` is *well-formed.* The built-in types are always well-formed; that is, there are instances

```
instance wft Int
instance wft (a, b)
instance wft (a -> b)
```

```
class Collection c where
  empty :: wft (c a) => c a
  singleton :: wft (c a) => a -> c a
  union :: wft (c a) => c a -> c a -> c a
  member :: wft (c a) => a -> c a -> Bool

data Eq a => Set a = Set [a]

instance Collection Set where
  empty = Set []
  singleton x = Set [x]
  union (Set xs) (Set ys) =
    Set (xs++[y | y<-ys, not (y ‘elem‘ xs)])
  member x (Set xs) = any (==x) xs
```

Figure 3: `Collection` and `Set` defined using restricted datatypes.

and so on.

A *restricted datatype definition*

$$\texttt{data (}\mathtt{C}_1\ \overline{a},\ldots,\mathtt{C}_n\ \overline{a}\texttt{) => T }\overline{a}\texttt{ = ...}$$

introduces instances

$$\begin{gathered}\texttt{instance (}\mathtt{C}_1\ \overline{a},\ldots,\mathtt{C}_n\ \overline{a}\texttt{) => wft (T }\overline{a}\texttt{)}\\ \texttt{instance wft (T }\overline{a}\texttt{) => }\mathtt{C}_1\ \overline{a}\\ \vdots\\ \texttt{instance wft (T }\overline{a}\texttt{) => }\mathtt{C}_n\ \overline{a}\end{gathered}$$

Unrestricted datatype definitions are just the special case where $n = 0$.

Now, we insist that *every type appearing in a program must be well-formed.* That is, every expression must appear in a context which guarantees that every sub-expression has a well-formed type. Likewise, every type signature must carry a context which guarantees that the type itself is well-formed; every data type, newtype, and type synonym definition must carry a context which guarantees that the types on the right hand side are well-formed; and every instance declaration must carry a context which guarantees that the instance type and the types occurring in the instance methods are well-formed. Type constructors may only be applied to parameters which are themselves well-formed[4]. However, we can assume that `wft a` holds for every polymorphic type variable `a`, (since in any instance of a polymorphic type the instantiating type must be well-formed), and so such constraints need not appear in contexts.

With this extension, we could define the `Collection` class and its `Set` instance as shown in Figure 3. As we see from this example, the `wft` constraints correspond to `Sat` constraints in our simulation; for example, `wft (Set a)` corresponds to `Sat (SetCxt a)`. Notice that we can freely use operations that depend on `Eq a`, such as `(==)` and `elem`, in the methods of the `Set` instance, thanks to the generated `instance wft (Set a) => Eq a`. The instances generated from a restricted

[4] This applies only to parameters of kind `*`. Type parameters of other kinds are not restricted, but of course if they are used then their applications must be well-formed. *Example:* if `A` is defined by `data wft (c a) => A c a = A (c a)`, then in any use `A` κ τ the type τ must be well-formed by this rule, while the type constructor κ must satisfy `wft (`κ τ`)` because of the context on the definition of `A`. This context must be present, since the type `(c a)` appears on the right hand side.

datatype definition are of course implemented just like the Sat instances we saw earlier:

$$\texttt{instance (C}_1\ \overline{a},\ldots,\texttt{C}_n\ \overline{a}\texttt{) => wft (T }\overline{a}\texttt{)}$$

constructs a dictionary for $\texttt{wft (T }\overline{a}\texttt{)}$ which is a tuple of the dictionaries on the left hand side, and the instances

$$\texttt{instance wft (T }\overline{a}\texttt{) => C}_i\ \overline{a}$$

just select the appropriate dictionary from the tuple. As in our simulation, overlapping instances force the compiler to search for a successful context reduction, and avoid detectable loops.

Since the compiler knows that the well-formedness constraint for Set is Eq and no other, we do not need to parameterise Set on Eq, or parameterise the Collection class separately on the well-formedness constraint.

Well-formedness of type variables. We assume that constraints wft a (where a is a type variable) are always satisfied, since a can only be instantiated to a well-formed type. Consequently such constraints do not appear in contexts, and no corresponding dictionary is passed at run-time. But is this really safe? Even if we know that a dictionary for wft a must exist, we cannot construct one should it prove to be needed, without knowing the type a. Thus we must convince ourselves that such a dictionary can never be needed, and so passing it as a parameter is unnecessary.

To see this, note that the dictionary corresponding to wft a is of an unknown shape: it could indeed be any dictionary at all depending on which type a is instantiated to. Dictionaries are used to implement calls of class methods, and any such use requires that one know the dictionary shape. Thus a wft a dictionary can never be used.

To prove this formally, we should specify the translation of our extended language into F^ω (which would in any case be desirable, since this translation is used in the Glasgow Haskell compiler). In this translation, context constraints are translated into the types of dictionary parameters, and a constraint wft a would be translated into another type variable. Thus the translated code would be polymorphic in the type of the dictionary, which implies by parametricity that the dictionary is unused.

A more subtle argument is needed for type variables which are parameters of classes, because even if the method types are 'polymorphic' in these variables, their implementations are not. For example, given the class declaration

```
class BinOp a where
  binop :: a -> a -> a
```

then an instance of binop at the type Set b might very well use the fact that Set b is well-formed. But since no wft a constraint appears in the type of the class method, then calls of binop will pass no dictionary.

However, recall that the *instance declaration* must require that the instance type is well-formed. In this example, we would be forced to write

```
instance wft (Set b) => BinOp (Set b) where
  binop = ...
```

Thus the dictionary for wft (Set b) is supplied when that for BinOp (Set b) is created; there is no need to pass it each time the binop method is called.

7 Discussion

7.1 On well-formedness

The major surprise in the design we have presented is the introduction of a new kind of constraint in contexts, the `wft` constraints. Our design requires programmers to understand and write this new kind of constraint, which may seem unsatisfactory in (for example) the `Collection` class definition in Figure 3, given our initial motivation that it is 'obvious' that `Set` elements must have an equality. One might argue that it is 'obvious' from the types of the `Collection` class methods that `wft (c a)` must hold, and therefore there is no need for the programmer to write it. More generally, we might implicitly add `wft` constraints to each type signature to require that all the types occurring in it are well-formed, and thereby spare the programmer the need to know about them.

We have chosen not to follow this route, because it is not possible in general to infer which `wft` constraints must hold for the *body* of a function to be well-typed, just from its *type signature*. For example, suppose we extend the `Collection` class with a `mapC` method as in section 5.1, and define

```
existsC :: (Collection c, wft (c a), wft (c Bool)) =>
  (a -> Bool) -> c a -> Bool
existsC p c = member True (mapC p c)
```

Notice that for `existsC` to be well-typed, then `Collections` of `Bool` must be well-formed, since such a `Collection` is used internally in the definition. But this type does not appear in the type of `existsC`, and so it is impossible to implicitly insert the constraint `wft (c Bool)` just given the type signature. In general we cannot use the body of a function to decide which constraints to add to its type signature, because the type signature might appear in a class definition, for example, while the associated bodies appear scattered throughout the program in the corresponding instance declarations.

Our conclusion is that `wft` constraints should always be explicit. A half-way house would be to implicitly add the constraints which are obviously needed from the type signature, but let the programmer write (hopefully rare) additional constraints explicitly. We consider it wiser to let programmers become used to `wft` constraints by writing them often.

Indeed, we claim that `wft` constraints are naturally associated with restricted datatypes: when we declare that sets may only be built from elements with equality, we are stating that some types are well-formed and others are not. It can hardly be surprising that we then need to reason explicitly about well-formedness.

It is interesting to note that similar issues arise in Jones and Peyton Jones' proposal for extensible records [JJ99]. There a record type `{r | x::t}`, denoting record type `r` extended with the field `x`, is well-formed only if `r` 'lacks' `x`, written `r\x`. These 'lacks' constraints clutter the types of functions, and just as we do, Jones and Peyton Jones consider introducing (some of) them automatically, when their necessity can be inferred from the type of the function alone. Thus there is an interaction between these two proposals, and in a final design the same decision should be made in both cases.

7.2 On abstraction

We began this article by bemoaning the fact that the type of the `member` function reveals too much about the way that `Sets` are implemented. Specifically, replacing an implementation in terms of lists by one in terms of ordered trees will probably change the type of `member` from

```
member :: Eq a => a -> Set a -> Bool
```

to

```
member :: Ord a => a -> Set a -> Bool
```

Consequential changes to the types of other functions could force the modification of many type signatures in other modules.

With the extension we propose, both these types can be replaced by

```
member :: wft (Set a) => a -> Set a -> Bool
```

When Set is implemented by lists, then wft (Set a) is equivalent to Eq a, and when Set is implemented by ordered trees, then wft (Set a) is equivalent to Ord a.

However, while our extension *enables* the programmer to write type signatures which are robust across changes to the representation of Sets, it does not *force* him to do so. For example, the function

```
isOneOf x y z = member x (singleton y `union` singleton z)
```

can be given the robust type

```
isOneOf :: wft (Set a) => a -> a -> a -> Bool
```

But it can also be given the type

```
isOneOf :: Eq a => a -> a -> a -> Bool
```

when Sets are represented by lists, since Eq a implies wft (Set a) in that case.

Equally, the function

```
maxMember x y s = member (x `max` y) s
```

can be given the robust type

```
maxMember :: (Ord a, wft (Set a)) => a -> a -> Set a -> Bool
```

But if Sets are represented by ordered trees, then it can also be given the type

```
maxMember :: wft (Set a) => a -> a -> Set a -> Bool
```

since wft (Set a) implies Ord a in that case.

If the programmer chooses to write type signatures such as these, whose validity depends on the conditions under which wft (Set a) holds, then of course a change to the representation of Sets will invalidate them. Our proposal makes it possible to write robust type signatures, but does not guarantee that all type signatures are robust.

In a sense, the constraint wft (Set a) behaves like a *context synonym* for the context on the definition of type Set, and the synonym is not abstract. It would be interesting to consider ways to restrict the scope of the synonym, for example to the same scope as the datatype constructors, to force programmers to write robust type signatures elsewhere.

7.3 Overhead of dictionary passing

A major problem with the implementation we have suggested is that it requires passing many more dictionaries than usual, leading to a potentially high overhead. In particular, the definition of class Monad must be revised as follows:

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

The thought of passing two dictionaries to every call of `(>>=)` is probably enough to put any implementor off.

The problem here is that programs which do not use restricted datatypes would still pay a heavy cost for their inclusion in the language, by passing a large number of empty dictionaries around. Our proposal here is to generate two versions of the code for each function whose type signature involves `wft`, one to be used when all the dictionaries involved are empty, and the other when genuinely restricted datatypes are involved. This should only affect constructor-polymorphic functions, so the amount of code duplication should be small, while the performance penalty for programs which do not use restricted datatypes is completely removed.

A related difficulty under our proposal is that adding restricted datatypes to the language might make some definitions overloaded which were not overloaded before — if the context they require contains only `wft` constraints. Such definitions would be subject to the monomorphism restriction after the extension, but not before, and this could cause type-checking to fail. We believe the correct solution here is to revise the monomorphism restriction.

7.4 Lazy vs eager context reduction

While we have explained that context reduction must search for a suitable reduction path, we have said little about *when* context reduction should occur. In Haskell as it stands, context reduction is performed *eagerly*, as part of inferring the type signature of each function. When the programmer states a context explicitly in a type signature, then it is clear that the compiler can search for a way to reduce the context that the function body requires to the given one. When the programmer leaves type signatures to be inferred by the compiler, then it is much less clear which reduction the compiler should choose. That leads us to suggest that context reduction should instead be *lazy*, that is, performed only when necessary to match contexts given explicitly by the programmer (or when a context is *tautologous*, that is can be reduced to an empty context). Non-tautologous contexts in inferred type signatures would not be reduced at all.

Peyton-Jones, Jones and Meijer come to the same conclusion in their exploration of the design space for classes [JJM97]. They point out in particular that, in combination with overlapping instances, eager context reduction can type fewer programs than lazy can. This is also true in our situation. Consider:

```
allpalin :: wft (Set [a]) => Set [a] -> Bool
allpalin (Set xss) = all palindrome xs
palindrome xs = xs==reverse xs
```

With lazy context reduction, the type signature inferred for `palindrome` is

```
palindrome :: Eq [a] => [a] -> Bool
```

and `allpalin` is well-typed, since `Eq [a]` is implied by `wft (Set [a])`. But if eager context reduction is used instead, then the type inferred for `palindrome` is

```
palindrome :: Eq a => [a] -> Bool
```

and `allpalin` becomes ill-typed — a good reason to prefer the former.

7.5 An alternative: abstracting over contexts

The extension we have proposed is not the only possible way to support restricted data types. An alternative would be to allow type and class definitions to be parameterised not just on types, but also on contexts. We could then carry out our simulation much more easily, without needing to represent classes by types — one would declare the Collection class, for example, as:

```
class Collection c cxt where
  empty :: cxt a => c cxt a
  ...
```

Of course, the ability to abstract over contexts might be useful in other ways too.

A natural complement would be to give contexts names via *context synonyms*. Interestingly, we can almost do so in Haskell already. A 'synonym' for the context (A a, B a) can be modelled by a new class and instance

```
class (A a, B a) => AB a
instance (A a, B a) => AB a
```

The instance declaration allows a context AB a to be reduced to (A a, B a), while the class declaration enables both A a and B a to be reduced to AB a. While context reduction might in principle loop here also, in fact the loop is avoided by treating class declarations specially.

However, this approach does have some important disadvantages. Firstly, even if we can abstract over a context in the Collection class, there is no obvious way to associate the type Set with the context Eq. We would need to make *both* the collection type, and the associated context, into parameters of the Collection class. To avoid ambiguous overloading, we could make sure that the context parameter appears in the method types (as we have done above). Alternatively, we might restrict instance declarations so that no two instances of the same class may have the same type parameters. Such a restriction would guarantee that the type parameters determine the context parameters, thus in effect creating an association between Set and Eq. In any case, there are subtle design choices to be made here.

Perhaps a more serious objection is that this approach would still require the class designer to *anticipate* that another programmer might later wish to make a restricted datatype into an instance. Many class designers would fail to do so, and the frustration of using restricted datatypes would remain.

8 Conclusion

It is common for the implementation of an abstract datatype to place some restrictions on its argument types. The consequent loss of abstraction when the implementation is changed, and the difficulties of making such implementations instances of more general classes, are recurring topics on the Haskell mailing list. We believe there is a crying need for a restricted datatype construct with much more power than Haskell's present sham.

The simulation in the first part of this paper enables us to explore the semantics and implementation of restricted datatypes. We argue that our solution is significantly more useful than either Peyton-Jones' approach to Collection classes or a more 'object-oriented' approach.

Finally, we propose a language extension based on our simulation, and argue that it is both natural, and can be implemented with reasonable efficiency. We believe the extension would be invaluable, and live in hope that Haskell implementors will take up the challenge!

Acknowledgements

Simon Peyton-Jones, Mark Jones, and Lennart Augustsson have all made very useful comments on this work at various stages. I am grateful to all of them, while

the remaining errors are of course my own.

References

[Hug99] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, to appear, 1999.

[JJ99] Mark P Jones and Simon Peyton Jones. Lightweight Extensible Records for Haskell. In *Haskell Workshop*, Paris, September 1999.

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

[PJ96] Simon Peyton-Jones. Bulk types with class. In *Electronic proceedings of the Glasgow Functional Programming Workshop*, Ullapool, July 1996.