Haskell Workshop

Sponsored by ACM SIGPLAN

Held in conjunction with ICFP97

Amsterdam

Saturday, June 7, 1997

Haskell Workshop

Sponsored by ACM SIGPLAN and held in conjunction with ICFP97, Amsterdam, Saturday, June 7, 1997

The definition of Haskell 1.4 has recently been released, but it is clear that there are many exciting opportunities ahead for developing and enhancing the language. Some of these will be in direct response to needs demonstrated by large-scale applications written in Haskell, or by Haskell being used in novel and interesting ways. Other developments will be driven by more theoretical considerations, whether they are moving in the direction of increased expressibility or, alternatively, simplifications to the language by recognizing unifying concepts. The purpose of the workshop is to provide a forum where possible future development directions for Haskell may be discussed.

The program committee consisted of

- Lennart Augustsson (Chalmers)
- Mark Jones (Nottingham)
- John Launchbury (OGI) - Chair
- Erik Meijer (Utrecht)
- John Peterson (Yale)
- Satnam Singh (Glasgow)
- Peter Thiemann (Tuebingen)

and I thank them for their work in reviewing and selecting the papers. I also thank the organizers of ICFP97 for taking care of the local arrangements.

J. Launchbury
May 1997

Workshop Program

Session 1. Chair: Lennart Augustsson

9:00-9:30. *Type Classes: an exploration of the design space.* Simon Peyton Jones, Mark Jones, and Erik Meijer

9:30-10:00. *Polymorphic Extensible Records for Haskell.* Benedict R. Gaster

10:00-10:30. *The Design and Implementation of Mondrian.* Erik Meijer and Koen Claessen

10:30-11:00. BREAK

Session 2. Chair: John Peterson

11:00-11:30. *Reactive Objects in a Functional Language.* Johan Nordlander and Magnus Carlsson

11:30-12:00. *Debugging Reactive Systems in Haskell.* Amr Sabry and Jan Sparud

12:00-12:30. *Bulk Types with Class.* Simon Peyton Jones

12:30-2:00. LUNCH

Session 3. Chair: Peter Thiemann

2:00-2:30. *Disposable Memo Functions.* Byron Cook and John Launchbury

2:30-3:00. *Green Card: a foreign language interface for Haskell.* Simon Peyton Jones, Thomas Nordin, and Alastair Reid

3:00-3:30. *Heap Compression and Binary I/O in Haskell.* Malcolm Wallace and Colin Runciman

3:30-4:00. BREAK

Session 4. Chair: John Launchbury

4:00-5:30. *Open-mike Session: What next for Haskell?* Participants are invited to make 5-minute statements on the future development of Haskell. These will be grouped by topic, and while the statements are limited to five minutes, any ensuing discussion is not. Anyone who wants to make a statement should indicate their desire in advance of the session.

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.

[3]This example was suggested by Sergey Mechveliani.

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.

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 $\{\texttt{Eq}\ \tau,\ \texttt{Eq}\ \tau\}$ to just $\{\texttt{Eq}\ \tau\}$.

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

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

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

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

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

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

Figure 1: Rules for entailment

   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:

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

   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:

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

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 $\tau\} \Vdash \{$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*

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

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

Foo (Tree α) => τ

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 Floats. 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 {` $op :: Q$ `=>` τ `; ... }`

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.

Polymorphic Extensible Records for Haskell

Benedict R. Gaster
Languages and Programming Group,
Department of Computer Science, University of Nottingham,
University Park, Nottingham NG7 2RD, England.
brg@cs.nott.ac.uk

Abstract

This paper describes an extension of Haskell that supports extensible records, with a full complement of polymorphic operations. It is a practical system which can be understood and implemented as a natural extension of Haskell. The proposed extensions have been implemented as part of the Hugs development system, and seem to work well in practice.

1 Introduction

Datatypes play an important role in all but the most trivial of programming tasks. For example, consider a program specification which requires that a selection of geometric shapes be transformed in variety of different ways. It is reasonable, even with only this informal description, to imagine a collection of new datatypes, one for each geometric shape, each of which may have a number of associated attributes. But how are such types represented in a program? In functional languages like Haskell [19], and Standard ML [15] products provide support for defining datatypes, allowing a selection of data items to be grouped together. For example, a datatype representing the geometric shape point, might be represented by the following Haskell definition:

$$\textbf{data}\ \mathit{Point} \ = \ \mathit{MkPoint}\ \mathit{Int}\ \mathit{Int}.$$

Although adequate, this definition is not particularly easy to work with in practice. For example, it is easy to confuse fields when they are accessed by position within a product.

To avoid these problems, the programming languages Haskell and Standard ML allow components of products to be identified using names drawn from some set of *labels*. Haskell 1.3 provides support for labelled products by allowing a datatype declaration to include field labels for components of the datatype. For example, the *Point* type described above might be defined more attractively as:

$$\textbf{data}\ \mathit{Point} \ = \ \mathit{MkPoint}\ \{x :: \mathit{Int},\ y :: \mathit{Int}\}.$$

Alternatively, Standard ML supports a more general notion of record types, which considers labelled products as separate entities from datatype declarations. In this setting, our *Point* example can be reformulated as[1]:

$$\textbf{type}\ \mathit{Point} \ = \ \mathit{Rec}\ \{x :: \mathit{Int},\ y :: \mathit{Int}\}.$$

Although Standard ML does not require that we predefine a type synonym for *Point*, in practice, it does provide a useful way of documenting one's intentions.

Both Haskell and Standard ML provide mechanisms allowing field names to be used in the construction and selection of record components without concern for the overall structure of the datatype. For example, Haskell ensures that, for each new label, a function working as a selector for that component is introduced at the top-level. Unfortunately, this has the undesirable side effect of forcing any two datatypes defined within the same scope, to use mutually exclusive field names. For example, returning again to the notion of geometric shapes, a datatype definition for circles including components x, y, and r representative of the circle's centre point and radius respectively, might be defined as:

$$\textbf{data}\ \mathit{Circle} \ = \ \mathit{MkCircle}\ \{x :: \mathit{Int},\ y :: \mathit{Int},\ r :: \mathit{Int}\}.$$

However, this definition is not valid if defined in the same scope as the *Point* shape described above. Moreover, datatypes defined in separate modules sharing common field names may only be used in the same namespace with careful use of qualified names. Standard ML avoids imposing similar restrictions on record fields, by requiring that the type of a record r is uniquely determined at compile-time. In effect, each different record

[1] To emphasize the notion of record types, we choose to incorporate a record constructor, *Rec*, where in fact the actual Standard ML definition is: **type** $\mathit{Point} = \{x : \mathit{Int},\ y : \mathit{Int}\}$.

type that includes an l field comes with its own method for extracting the value of that field. By requiring that the record type can be determined during type checking, the overloading that results from using the same notation for each of these operations is easily resolved.

An unfortunate consequence of the restrictions imposed on record types by both the Haskell and Standard ML type systems is that operations provided for manipulating records are less flexible than might be expected. For example, consider operations to extract the centre point of a given shape. We might reasonably expect that polymorphism would provide the ability to define a single definition for all shapes:

$$\mathit{centre}\ \mathit{shape} \;=\; (\mathit{shape}\,.\,x,\ \mathit{shape}\,.\,y),$$

where $(\_.x)$ denotes selection of the field x from some arbitrary record. However, although both Haskell and Standard ML provide support for polymorphic definitions, no support is provided for the analogous idea of polymorphism over fields, which allows unimportant labels to be ranged over by a single variable. It is the requirement that record types be completely determined at compile-time—enforced by the application of constructors in Haskell, and by user specified type annotations in Standard ML—that limit operations over records to monomorphic type. A further weakness of the Haskell and Standard ML record systems is that no support is provided for *extensibility*; there are no general operators for adding and removing fields in a record value, for example. The following definition, which is not legal in either Haskell or Standard ML, shows how extensibility might be applied to allow an additional colour field to be incorporated into arbitrary shape values:

$$\mathit{colour}\ c\ \mathit{shape} \;=\; (\mathit{colour} = c \,|\, \mathit{shape}),$$

where the operator $(\mathit{colour} = \_ | \_)$ denotes the extension of an arbitrary record with a new field *colour*.

1.1 This paper

This paper presents an alternative proposal for records in Haskell[2], by combining ideas that have been used in previous work to develop a practical type system. In particular, it supports extensible records, with a full complement of polymorphic operations. For example, the point and circle shapes described above can be reformulated as:

$$\begin{aligned}\textbf{type}\ \mathit{Shape}\ r &= \mathit{Rec}\ \{\!|x :: \mathit{Int},\ y :: \mathit{Int} \,|\, r|\!\}\\ \textbf{type}\ \mathit{Circle} &= \mathit{Shape}\ \{\!|\mathit{rad} :: \mathit{Int}|\!\}.\end{aligned}$$

[2]The record system discussed in this paper would be equally suitable for an extension of Standard ML. However, the type system is based on the notion of qualified types, which is the core type system of Haskell, and as such, Haskell seems an obvious choice.

Here record type extension, denoted by $\{\!|l :: \_ | \_|\!\}$, provides us with the ability to define *Circle* as an extension of the type *Shape* which includes at least the fields x and y; i.e., a value of type *Shape* is at least a value of type *Point* introduced above. Intuitively, a value of type *Circle* contains all the fields of a *Shape*, plus an extra component *rad*. In fact, we might have done equally well to define *Circle* as:

$$\textbf{type}\ \mathit{Circle} \;=\; \mathit{Rec}\ \{\!|x :: \mathit{Int},\ y :: \mathit{Int},\ r :: \mathit{Int}|\!\}.$$

The combination of extensibility, and the ability to define polymorphic operations over records provides us with the functionality to define, and type correctly, the definition of *centre* described above:

$$\begin{aligned}\mathit{centre} &:: \mathit{Rec}\ \{\!|x :: \mathit{Int},\ y :: \mathit{Int} \,|\, r|\!\} \\ &\qquad\qquad \to (\mathit{Int},\ \mathit{Int})\\ \mathit{centre}\ \mathit{shape} &= (\mathit{shape}\,.\,x,\ \mathit{shape}\,.\,y)\end{aligned}$$

Note, that unimportant fields (e.g., the radius component of a circle) are bound to the variable r.

This paper provides an informal presentation of extensible records for Haskell and refrains from in depth discussion of related record calculi. In particular we do not consider proposals for extending Standard ML with similar record operations, for example Rémy [21, 22] and Ohori [18]. For an in depth, formal treatment of the record system proposed in this paper, including a discussion of related work, the interested reader might look at Gaster and Jones' paper introducing a record and variant calculi for qualified types [6].

We conclude this section by outlining the main subjects covered in the remaining parts of this paper:

- Section 2 provides an informal overview of our proposal for extensible records in Haskell. A number of basic record operations are considered, which are natural generalizations of operators that are already present in Haskell.
- Section 3 describes an implementation for extensible records. Analogous to the standard notion of class constraints representing implicit dictionary parameters, field constraints are considered as evidence for offsets into a given record.
- Section 4 considers a number of pragmatic issues concerning the integration of extensible records into Haskell. In particular, any serious proposal extending Haskell with new primitive datatypes, must consider the general framework of deriving instances for standard classes (e.g., equality), and must address questions of syntax, and pattern matching.
- Section 5 concludes, summarizing some possibilities for future work.

2 Basic record operations

Record types are defined by application of the constructor *Rec* to well-formed rows, which are themselves constructed by extension, starting from the empty row, $\{\!|\,|\!\}$. A row may be thought of as partial function from labels to simple types. It is convenient to introduce abbreviations for rows obtained in this way:

$$\begin{array}{lcl} \{\!| l_1{::}\tau_1, \ldots, l_n{::}\tau_n \mid r |\!\} & = & \{\!| l_1{::}\tau_1 \mid \ldots \{\!| l_n{::}\tau_n \mid r |\!\} \ldots |\!\} \\ \{\!| l_1{::}\tau_1, \ldots, l_n{::}\tau_n |\!\} & = & \{\!| l_1{::}\tau_1, \ldots, l_n{::}\tau_n \mid \{\!|\,|\!\} |\!\}. \end{array}$$

Note, however, that we treat rows, and hence record types, as equals if they include the same fields, regardless of the order in which those fields are listed. Intuitively, a record of type $Rec\ \{\!| l :: \alpha \mid r |\!\}$ is like a pair whose first component is a value of type α, and whose second component is a record of type $Rec\ r$. This motivates our choice of basic operations, which correspond closely to the projection and pairing functions of Haskell product types. There is, however, one complication; we do not allow repeated uses of any label within a particular row, so the expression $\{\!| l :: \alpha \mid r |\!\}$ is only valid if l does not appear in r. This is reflected by prefixing each of the types below with a predicate $(r \backslash l)$, pronounced "r lacks l":

- Selection: to extract the value of a field l:

  $$(\_.l) :: (r\backslash l) \Rightarrow Rec\ \{\!| l :: \alpha \mid r |\!\} \rightarrow \alpha.$$

- Restriction: to remove a field labelled l:

  $$(\_ - l) :: (r\backslash l) \Rightarrow Rec\ \{\!| l :: \alpha \mid r |\!\} \rightarrow Rec\ r.$$

- Extension: to add a field l to an existing record:

  $$(l = \_ \mid \_) :: (r\backslash l) \Rightarrow \alpha \rightarrow Rec\ r \rightarrow Rec\ \{\!| l :: \alpha \mid r |\!\}.$$

A predicate of the form $(r\backslash l)$ prevents a record r being extended with a field already present, but no analogous operation is provided at the level of types. However, although record types containing multiple occurrences of the same label are legal (e.g., $Rec\ \{\!| l :: Bool, l :: Int |\!\}$ is a valid type), they are uninhabited[3], and as such, it is impossible to construct proper values with the appropriate type. In practice a compiler may incorporate static checks to avoid record types with multiple labels within module boundaries. For example, consider the following type definitions:

$$\begin{array}{lcl} \textbf{type}\ Foo\ r & = & Rec\ \{\!| l :: Int \mid r |\!\} \\ \textbf{type}\ Foo' & = & Foo\ \{\!| l :: Bool |\!\}. \end{array}$$

[3]The primitive operation for record extension insures, by means of a predicate $r\backslash l$, that a record is only ever extended with a field not already present.

Expanding Foo' gives the type expression

$$Rec\ \{\!| l :: Bool, l :: Int |\!\},$$

which can be flagged as a error at compile-time.

We can use the basic operations described above to implement a record update operation which, unlike datatypes with labelled fields, does not restrict the type of the updated field:

$$\begin{array}{lcl} (l := \_ \mid \_) & :: & (r\backslash l) \Rightarrow \alpha \rightarrow Rec\ \{\!| l :: \beta \mid r |\!\} \\ & & \quad \rightarrow Rec\ \{\!| l :: \alpha \mid r |\!\} \\ (l := x \mid r) & = & (l = x \mid r - l). \end{array}$$

As a concrete example of these operations, and highlighting the use of extensibility, consider a hierarchy of algebraic structures in which monoids (structures with a set and an associative binary operation) form the base of the hierarchy, and group and ring structures are defined as extensions of monoids and groups, respectively. A group supports all operations of a monoid plus an inverse, and a ring supports all operations of a group plus some of its own. Given, an appropriate implementation of this hierarchy, a user might reasonably expect to define operations, requiring only the functionality of monoids, over all algebraic structures. Figure 1 provides an implementation of this hierarchy in terms of extensible records, accompanied by sample implementations, for the integers.

$$\begin{array}{lcl} \textbf{type}\ Monoid\ v\ r & = & Rec\ \{\!|\ plus :: v \rightarrow v \rightarrow v, \\ & & \qquad id :: v \mid r |\!\} \\ \textbf{type}\ Group\ v\ r & = & Monoid\ v\ \{\!| inv :: v \rightarrow v \mid r |\!\} \\ \textbf{type}\ Ring\ v\ r & = & Group\ v\ \{\!|\ mult :: v \rightarrow v \rightarrow v, \\ & & \qquad one :: v \mid r |\!\} \\ iMonoid & :: & Monoid\ Int\ \{\!|\,|\!\} \\ iMonoid & = & (plus = (+), id = 0) \\ iGroup & :: & Group\ Int\ \{\!|\,|\!\} \\ iGroup & = & (inv = negate \mid iMonoid) \\ iRing & :: & Ring\ Int\ \{\!|\,|\!\} \\ iRing & = & (mult = (*), one = 1 \mid iGroup) \end{array}$$

Figure 1: Example algebraic hierarchy

The standard list function *sum*, for computing the sum of a list, can now be recast in terms of any monoid:

$$\begin{array}{lcl} sum & :: & Monoid\ \alpha\ r \rightarrow [\alpha] \rightarrow \alpha \\ sum\ mon & = & foldr\ (mon.plus)\ (mon.id) \end{array}$$

Here, r ranges over rows containing zero or more fields, which in the case when the function *sum* is applied to *iGroup*, r is bound to the single field *negate*. Thus extensibility captures a form of sub-typing that is also present, although in a slightly different form, in the Haskell class system. However, we believe that this notion of sub-typing is present in a number of different programming situations, many of which are more suited to extensibility than they are to obscure encodings using the class mechanism.

Extensibility provides a simple form of inheritance, more commonly found in object-oriented languages [23, 2, 1]. Hughes and Sparud [7], have shown that the Haskell class system provides an alternative form of inheritance, which can be utilized to encode object-oriented features. It remains to be seen whether records with extensibility will provide a practical platform for incorporating object-oriented features into Haskell.

3 Record implementation

This section explains how the data structures and operations described in the previous section can be implemented. We focus on the implementation of record extension, $(l = \_ \mid \_)$, which, aside from record selection, is probably the most frequently used basic operation. A naive approach would be to represent a record by an association list, pairing labels with values. This would allow simple implementations for each of the basic operations, with the type system providing a guarantee that the same field would never appear more than once in a given record. A major disadvantage is that it does not allow constant time access to record components.

To avoid these problems, we will assume instead that a record value is represented by a contiguous block of memory that contains a value for each individual field. To select a particular component $r.l$ from a record r, we need to know the offset of the l field in the block of memory representing r. Languages without polymorphic selection will usually only allow an expression of the form $r.l$ if the offset value, and hence the structure or even the full type of r, as in Haskell, is known at compile-time.

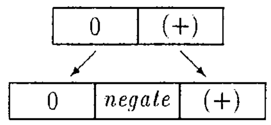
However, it is not actually necessary to know the position of every field at compile-time; instead, we can treat unknown offsets as implicit parameters whose values will be supplied at run-time when the full types of the records concerned are known. This is essentially the compilation method that was used by Ohori [18], and also suggested, independently, by Jones [8]. Assuming records are implemented as arrays of equally sized cells, in which record fields are stored consistently with respect to some total ordering on labels, the extension operator, $(l = \_ \mid \_)$, can be implemented by a function $\lambda i.\lambda v.\lambda r.pext\ i\ v\ r$, using the extra parameter i to supply the offset at which the value v is to be inserted into the record r. For example, the expression:

$$(inv = negate \mid iMonoid)$$

can be implemented by compiling it to

$$(\lambda i.\lambda v.\lambda r.pext\ i\ v\ r)\ 1\ negate\ iMonoid,$$

assuming a lexicographical ordering on labels. The resulting expression, *pext* 1 *negate iMonoid*, can be implemented by simple copying procedures, allowing the correct insertion of the value *negate*. This process is captured diagrammatically by the following diagram:



Note, that all fields with labels considered less than the label being inserted, remain in the same position in the array, while all fields following the inserted field are shifted up one position. Record restriction can be implemented in a similar fashion, where instead of larger field labels being shifted up one position they are shifted down one.

Of course, there are run-time overheads in calculating and passing offset values as extra parameters. However, an attractive feature of our system is that these costs are only incurred when the extra flexibility of record extension and polymorphic selection is required. Operations like record extension and restriction will, in general, be implemented by copying. Optimizations can be used to combine multiple extensions or restrictions of records, avoiding unnecessary allocation and initialization of intermediate values. For example, a compiler can generate code that will allocate and initialize the storage for a record $(x = 1, y = 2, z = 3)$ in a single step, rather than a sequence of three individual allocations and extensions as a naive interpretation might suggest.

The typechecker gathers and simplifies the predicates generated by each use of an operator on records. For example, if *iMonoid* is a value of type *Monoid Int* $\{\!|\,|\!\}$, then an expression like *iMonoid.id* will generate a single constraint, $\{\!|plus :: Int|\!\}\backslash id$. Predicates like this, involving rows whose structure is known at compile-time, are easily discharged by calculating the appropriate offset value. Obviously, a compiler can use this information to produce efficient code by inlining and specializing the selector function, $(\_.id)$.

It is possible that our more general treatment of record operations could result in compiled programs that are littered with unwanted offset parameters; experience with our prototype implementation will help to substantiate or dismiss these concerns. In any case, there are simple steps that can be taken to avoid such problems. For example, a compiler might reject any definition with an inferred type containing predicates, unless an explicit type signature has been given to signal the programmer's acceptance. This is closely related to the *monomorphism restriction* in Haskell and to proposals for a *value restriction* in Standard ML [24, 13].

4 Pragmatic Issues

Previous sections highlighted a number of shortcomings with the current solution for records in Haskell, and proposed a system of polymorphic extensible records, naturally extending the Haskell type system. However, hitherto we have avoided considering the more pragmatic issues, which often arise with proposed extensions to non-trivial languages, such as Haskell. In this section we consider three particularly important practical concerns for extensible records in Haskell. Section 4.1 considers the question of pattern matching over records. Section 4.2 highlights the difficulty in selecting a suitable syntax. Section 4.3 presents extensions of the Haskell class mechanism, to allow for derived instances of equality and text operations over records.

4.1 Pattern matching

As with other datatypes in Haskell, pattern matching is often a natural way to extract the components of a record. For example, considering again the algebraic hierarchy of Section 2, pattern matching provides an alternative definition for the function *sum*:

$$\begin{array}{lcl} sum & :: & Monoid\ \alpha\ r \rightarrow [\alpha] \rightarrow \alpha \\ sum\ (plus = p, id = i \mid r) & = & foldr\ p\ i. \end{array}$$

Intuitively an expression of the form *sum e* is evaluated from left to right, first evaluating the pattern bindings for p and i, and then binding all other components to the pattern r. More generally we can explain the semantics of pattern matching over records using a translation of the form:

$$\begin{array}{l} \backslash(l = p \mid r) \rightarrow e \ \triangleq\ \backslash x \rightarrow \textbf{case}\ x\ .\ l\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad p \ \rightarrow\ \textbf{case}\ x - l\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad r \ \rightarrow\ e. \end{array}$$

Note that, although we propose that the source level use of record restriction be restricted to patterns only (see Section 4.2), our implementation of pattern matching requires record restriction as a primitive notion. However, such uses do not introduce any new syntactical problems.

4.2 Issues of Syntax

Unquestionably, choosing an appropriate syntax plays an important role in the success or failure of programming language features. For example, Haskell allows pattern matching on the left hand side of function bindings, which in turn provides a convenient mechanism for describing inductive definitions. If however, pattern matching was supported only within the case construct, then inductive definitions may not seem as attractive.

Section 2 introduced a syntax for record types and operations, which overlapped with that of Haskell. For example, record selection was written using the symbol (.), which is already used for function composition in Haskell. The fact that we use the symbols ⦃ and ⦄, which are not defined by Haskell's lexical syntax, complicates matters even further.

Any discussion of syntax for extensible records in Haskell, must consider whether records as described by the Haskell 1.3 report are to be retained. If not retained, then the syntax for record types can be simply that of Section 2, replacing the symbols ⦃ and ⦄ with { and } respectively. As the system presented in this paper supports the record operations of Haskell, we believe that it is not unreasonable to consider replacing one by the other.

We now turn our attention to the question of syntax for record values. Our main concern is that the syntax of Section 2 introduces ambiguities when considered with respect to Haskell's syntax. Unfortunately, although record extension integrates smoothly, this is not the case for either record selection or restriction. In practice we have found only a few applications for record restriction, and in such cases pattern matching over records has proven adequate. In contrast, record selection appears in all but the most trivial of record applications, and although pattern matching provides an alternative, we believe that this operation must be supported using a convenient notation.

Record selection in Standard ML [15] is represented by the appropriate label prefixed by the symbol #. Thus selection of the field l of a record r is denoted by the expression $\#l\ r$. However, having experimented with this notation, we found that important program details were often hard to visualize. With this in mind, we strongly believe that $(\_.l)$ is the correct notation for record selection. This leaves us with the important question of what to do about function composition, which is denoted by the symbol (.) in current versions of Haskell. It may come as a surprise to the reader that, in fact, we propose that function composition be represented by the symbol #, even though we felt it to be inappropriate for record selection. Moreover, since function composition is in fact an instance of the Haskell class *Functor*, at type $((\rightarrow)\alpha)$, it could be predefined as part of the *Functor* class:

$$\begin{aligned}&\textbf{class } \mathit{Functor}\ f \textbf{ where}\\ &\quad (\#) \ :: \ (\alpha \rightarrow \beta) \rightarrow (f\ \alpha \rightarrow f\ \beta)\end{aligned}$$

Function composition is defined simply as an instance of this class:

$$\begin{aligned}&\textbf{instance } \mathit{Functor}\ ((\rightarrow)\ \alpha) \textbf{ where}\\ &\quad (f \# g)\ x \ = \ f\ (g\ x).\end{aligned}$$

Fortunately, associativity for function composition is preserved. To see this, recall that the following equation is satisfied by any functor [12]:

$$(f\ \#\ g)\ \#\ h = f\ \#\ (g\ \#\ h),$$

for which f, g and h are of the appropriate types. Instantiating the different uses of $(\#)$ to function composition, of the appropriate types, gives the equality [4]:

$$(f\ .\ g)\ .\ h = f\ .\ (g\ .\ h),$$

which is precisely the required associativity law.

Figure 2 contains our proposed extensions for the Haskell grammar, which itself appears in appendix B of the Haskell report [19].

A more long term perspective for adopting an alternative record proposal for Haskell, might involve considering tuples of type $(\tau_1, \cdots, \tau_n)$ to be shorthand for records of type $\mathit{Rec}\ \{\!|1 :: \tau_1, \cdots, n :: \tau_n|\!\}$. A similar relationship between tuples and records has been adopted by Standard ML [15], and seems to provide a number of practical benefits, not least a general mechanism for selecting arbitrary components of tuples.

4.3 Records and the Haskell class system

Often, the design and implementation of a new datatype requires more than just specifying the datatype definition itself. For example, one must ask questions such as: Is equality defined over elements of the new datatype? Are elements of this type printable? And so on. Although many of these questions will be related to specific applications, there is a class of operations that arise for almost all datatypes (e.g., equality).

To ease the programming burden, Haskell provides a number of predefined type classes for operations such as equality and printing, for which instances can be derived automatically by the compiler. This section considers how an implementation might automatically derive instances of the *Eq* and *Show* classes over records.

[4] We return briefly to denoting function composition as (.), in order to help the discussion.

An obvious first attempt at defining equality over records might involve having the compiler generate instance declarations of the form:

$$\begin{aligned}&\textbf{instance } (\mathit{Eq}\ (\mathit{Rec}\ r),\ \mathit{Eq}\ \alpha) \Rightarrow (\mathit{Rec}\ \{\!|l :: \alpha \,|\, r|\!\}) \textbf{ where}\\ &\quad r == r' \ = \ (r.l == r'.l)\ \&\&\ (r - l == r' - l),\end{aligned}$$

for each record extension with a field l. However, even if we overlook the fact that Haskell does not support contexts of the form shown here, this does not give a well-defined notion of equality. To see this consider the expression:

$$(x = 10, y = \bot) \ == \ (x = 20, y = 30),$$

where $\bot$ is a diverging term of type *Int*. Evaluating this expression from left to right results in the boolean value *False*. However, we consider rows equal modulo reordering of fields, thus applying commutativity and evaluating from left to right gives $\bot$ for the above equality. Thus if we are not careful when deriving equality over records, then it is possible that different implementations may produce differing results. The problem lies in the fact that rows, and thus records, are considered equal modulo reordering of fields.

The clue to resolving this problem lies in Section 3, where the well-formedness of record compilation was guaranteed by considering a total ordering on labels. Intuitively, equality over records is well-defined if corresponding pairs of fields are compared in precisely the order determined by their labels in the record type that we are concerned with. Operationally, one can think of record equality as: given any two records of the same type, construct an ordered list of pairs, in which the first element of each pair is the string for a particular label and the second is a (delayed) boolean test of equality for the values associated with a given label. The following class definition captures this notion of equality over records:

$$\begin{aligned}&\textbf{class } \mathit{EqRecRow}\ r \textbf{ where}\\ &\quad \mathit{eqRecRow} \ :: \ \mathit{Rec}\ r \rightarrow \mathit{Rec}\ r \rightarrow [(\mathit{String},\ \mathit{Bool})].\end{aligned}$$

To ensure that the definition of equality, over records, is well-defined, an implementation must guarantee that instances of this class can only be generated internally, on application of the extension operator.

The instance for the empty row can be predefined, in a suitable library, as:

$$\begin{aligned}&\textbf{instance } \mathit{EqRecRow}\ \{\!||\!\} \textbf{ where}\\ &\quad \mathit{eqRecRow}\ \_\ \_ \ = \ [\,]\end{aligned}$$

Now, providing that suitable implementations are constructed on each application of record extension, we

label	$\rightarrow$	*varid*	*(field names)*
rowvar	$\rightarrow$	*tyvar*	*(row variables)*
row	$\rightarrow$	$\{label_1 :: type_1, \cdots, label_n :: type_n\}$	$(n \geq 0)$
	$\mid$	$\{label_1 :: type_1, \cdots, label_n :: type_n \mid row\}$	$(n \geq 1)$
	$\mid$	*rowvar*	*(row variables)*
atype	$\rightarrow$	$\cdots$	
	$\mid$	*Rec row*	*(record type)*
fexp	$\rightarrow$	$[fexp]\, dexp$	*(function application)*
dexp	$\rightarrow$	$dexp.label$	*(record selection)*
	$\mid$	*aexp*	
aexp	$\rightarrow$	$\cdots$	
	$\mid$	$(label_1 = exp_1, \cdots, label_n = exp_n)$	$(n \geq 0)$
	$\mid$	$(label_1 = exp_1, \cdots, label_n = exp_n \mid exp)$	$(n \geq 1)$
apat	$\rightarrow$	$\cdots$	
	$\mid$	$(label_1 = pat_1, \cdots, label_n = pat_n)$	$(n \geq 0)$
	$\mid$	$(label_1 = pat_1, \cdots, label_n = pat_n \mid pat)$	$(n \geq 1)$

Figure 2: Proposed syntax for extensible records in Haskell

can safely define a single, general, instance for equality over records:

$$\begin{array}{l} \textbf{instance}\ EqRecRow\ r \Rightarrow Eq\ (Rec\ r)\ \textbf{where} \\ \quad x == y \ = \ all\ (map\ snd\ (eqRecRow\ x\ y)) \end{array}$$

Derivable instances may be defined similarly for the classes *Ord* and, *Show*. For example, Figure 3 contains an implementation for showing record values[5]. Analogous with the function *eqRecRow*, described above, the function *showRecRow* generates an ordered list of pairs for a given record. The second component of each pair represents the showable value associated with a given label l.

5 Conclusion

We have described a natural extension to Haskell, that is flexible enough to allow polymorphic extensible records. This system generalizes the current Haskell and Standard ML record systems, allowing polymorphic operations over records and extensibility. A prototype implementation has been incorporated into Hugs, an implementation of Haskell 1.3 [9]. Our experience to date shows that the implementation works well in practice. There are a number of areas for further work:

[5]Following the discussion of Section 4.2 the composition of functions is represented by $(\#) :: (\alpha \rightarrow \beta) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \gamma \rightarrow \beta$.

- *First class polymorphism and extensible records:* Recent work by Läufer and Odersky [17], Jones [11], and Garrigue and Rémy [5] has shown that a type system based upon let-polymorphism [14, 3, 4] can be extended to allow polymorphic values as first class citizens. A common theme in this work is the requirement that hints are provided to the type inference system (for example, polymorphic values can be constructed by application of constructor functions). It is our belief that such hints may be provided by the construction and deconstruction of record values and on going work is showing promising results.
- *First class modules for Haskell:* It has long been realized that the Standard ML module system provides a range of software engineering benefits over that of the Haskell module system. It is to this end, that we are interested in unifying and combining the record system described in this paper, with Jones' [10] mechanism for modules using parameterized signatures, with the long term objective of incorporating first class modules into Haskell.
- *A new approach to datatypes:* Gaster and Jones [6] have shown that the type system providing support for the operations introduced in Section 2 not only supports extensibility over records, but also provides this functionality for variants. To this

```
instance ShowRecRow r ⇒ Show (Rec r) where
  showsPrec d    =  showFields#showRecRow

showFields      ::  [(String, ShowS)] → ShowS
showFields []   =   showString "()"
showFields xs   =   showChar '('
                #   foldr1 comma (map fld xs)
                #   showChar ')'

comma a b       =   a#showString ", "#b
fld (s, v)      =   showString s#showChar '=' #v

class ShowRecRow r where
      showRecRow  ::  Rec r → [(String, ShowS)]

instance ShowRecRow {} where
  showRecRow _  =  []
```

Figure 3: Functions to "show" record values

end, we are interested in developing the ideas in this paper into a general framework for extensible datatypes in Haskell.

- *Object-oriented programming in Haskell:* Section 2 noted that extensibility is closely related to inheritance found in languages based on an object-oriented methodology. A general notion of extensibility is captured through our use of rows, which potentially may be suitable for object-oriented extensions of Haskell. For example, a constructor *Obj* might be used to describe objects with a given row of methods [16, 20].

Acknowledgements

This work was supported in part by an EPSRC studentship 9530 6293. I would also like to thank my colleagues in the functional programming group at Nottingham, in particular Mark P. Jones and Colin Taylor, for the valuable contributions that they have made to the work described in this paper.

References

[1] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[2] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[3] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] L. M. M. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, April 1985. Technical report CST-33-85.

[5] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. In *International Symposium on Theoretical Aspects of Computer Software*, September 1997.

[6] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Computer Science, University of Nottingham, November 1996.

[7] J. Hughes and J. Sparud. Haskell++: An object-oriented extension of Haskell. In *Proceedings of Haskell Workshop, La Jolla, California*. YALE Research Report DCS/RR-1075, 1995.

[8] M. P. Jones. *Qualified Types Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[9] M. P. Jones. The Hugs 1.3 distribution. Available from the University of Nottingham: `http://www.cs.nott.ac.uk/Department/Staff/mpj/`, August 1996.

[10] M. P. Jones. Using parameterized signatures to express module structure. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, pages 68–78. ACM, January 1996.

[11] M. P. Jones. First-class polymorphism with type inference. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 483–496. ACM, January 1997.

[12] S. M. Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer-Verlag, 1972.

[13] X. Leroy. Polymorphism by name for references and continuations. In *Principles of Programming Languages*, pages 220–231. ACM press, 1993.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[15] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* The MIT Press, 1990.

[16] J. C. Mitchell, F. Honsell, and K. Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 1993.

[17] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 24th Symposium on Principles of Programming Languages*, pages 65–67. ACM, January 1996.

[18] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov. 1995. Preliminary version in Proceedings of ACM Symposium on Principles of Programming Languages, 1992, under the title, A compilation method for ML-style polymorphic record calculi.

[19] J. Peterson and K. Hammond. Report on the Programming Language Haskell, A Non-strict, Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, May 1996.

[20] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, Apr. 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[21] D. Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM press.

[22] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Type, Semantics, and Language Design*, Foundations of Computing Series. MIT Press, 1994. Early version appeared in Sixteenth Annual Symposium on Principles of Programming Languages. Austin, Texas, January 1989.

[23] M. Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[24] A. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

The Design and Implementation of Mondrian

Erik Meijer and Koen Claessen
OGI and Utrecht University
{erik, koen}@cse.ogi.edu

Abstract

The Haskell dialect Mondrian is designed using the explicit philosophy of keeping things simple and consistent. Mondrian generalizes some of Haskell's (too) complex constructs, and adds a few simple new ones. This results in a small, intuitively comprehensible language with an object oriented flavor.

In this paper, we will present the design decisions we made for Mondrian. Furthermore, some of Mondrian's language constructs will be defined by translations into Haskell.

1 Introduction

In the preface of the Haskell report [13] the hope is expressed that extensions or variants of the language will appear that incorporate experimental features. The language Mondrian is such an experiment. Mondrian evolved from Haskell by deleting and combining some of Haskell's more complicated constructs and adding a few simple new ones. Unlike the language Pizza [29], which is designed as a strict superset of Java [8], the major concern in designing Mondrian has been to keep the language as basic and down to the bare essence as possible, even if this implied breaking compatibility with Haskell. We do define Mondrian by translating it into Haskell.

The hype about object-oriented programming is not without cause. Viewing the world as a collection of interacting objects is conceptually very natural. When it comes down to programming the behavior of objects in an imperative language, however, we miss the abstraction mechanisms offered by functional languages, and we are less enthousiastic. Mondrian combines algebraic data types and type classes into a naive (no real subtyping) object-oriented type system, so that it becomes possible to program in an object-oriented style within the functional paradigm. Mondrian does *not* pretend to be a real object-oriented language. All type-checking remains covariant and "message not understood" runtime errors are not prevented by the type checker. Fundamentally though, Haskell suffers from the very same problem in the sense that the compiler does not reject partial functions defined using pattern matching, something that is not normally perceived as a *typing* issue.

Types are significant in the programming process. Most of us share the experience that once we have the types right, the program is correct. Therefore it is a pity that in Haskell the abstraction mechanisms on the level of types fall short when compared to those on the level of expressions. On the term level we have let-expressions to name program fragments, and λ-expressions and application to parametrize over program fragments. We should be able to use these on the type level as well. Instead of Haskell's primitive mechanism of type synonyms, Mondrian has a full functional language available on the level of types, but we push the borders just a little bit to keep things as simple as possible.

If we think that it is a good idea that programmers document their code by supplying explicit type signatures for function definitions, their programming language should encourage this by providing convenient mechanisms to give type annotations. In particular, the programmer should be able to specify types of locally defined functions that are polymorphic in some type variables and monomorphic in others. Mondrian follows SML by introducing scoped type variables.

As an experiment the syntax of Mondrian modules is made consistent with that of value definitions. Identifiers that are exported must be marked as such at their declaration instead of writing them as arguments in the module heading. From a software engineering point of view this leads to more weakly coupled code. In the current situation, changing the name of an identifier requires changing a name in the export list as well. We do not consider the lack of explicit module interfaces to be a problem. In contrary, module interfaces should be generated automatically by a program browser.

At the implementation level, a novel aspect of Mondrian is the use of Pure Type Systems [10] as a typed intermediate language [24]. Instead of having three different little languages for terms, types and kinds, all of which must support similar operations, it is more convenient to have a single language that captures all levels, and allows control over the way levels can interact. A uniform framework like PTS provides an even greater degree of generality than the Glasgow Haskell Compiler backend [22], which is based on the polymorphic lambda calculus.

2 History, motivation and background

Functional programming plays an important role in the computing science curriculum at Utrecht University. For teaching, the disadvantages of functional languages such as slow speed and large memory consumption are not prohibitive, as they are for industrial usage. Moreover, the advantages of functional languages [16] such as abstraction from evaluation order (lazy evaluation), abstraction from computational patterns (higher-order functions), abstraction from type structure (polymorphism), and high code density, prove to be of great didactical value.

In the last few years we have replaced mathematical syntax by a functional language, in casu Gofer [17], in a number of courses [1] including *Grammars and Parsing*, *Computer Architecture*, and *Implementations of Programming Languages*. The advantages of implementing ideas in a programming language instead of in plain mathematics are that they are type checked and often even executable. In practice, we do not feel constrained by notational restrictions imposed by the use of concrete syntax, although it takes a lot of care to hide unimportant details. On the other hand, it is perhaps too easy to omit essential details when using mathematical notation.

As heavy users of functional languages we felt an ever increasing need to have our own implementation of a Haskell-like language. This would allow us to use the feedback we get from teaching to improve our programming language directly, without being dependent on other implementors who might have different priorities and goals. Understandably Haskell developers are best served by a stable language and thus reluctant to change.

Our home grown compiler should be simple, small, extensible, and written in its own source language. Efficiency is not particularly important. To be competitive with the imperative language implementations our students know and use at home on their PCs (Visual Basic, Visual C++, Visual J++, Delphi), the implementation should come with a fancy graphical programming environment.

A second drive for contriving a Haskell dialect is our belief that in order for functional programming to succeed in the real world, some support for object orientation is a necessary (but not sufficient[1]) condition. As Hughes and Sparud argue [15], Haskell currently lacks the form of incremental reuse that is offered by inheritance in object-oriented languages. Using inheritance you can extend (subclass) a given data type and then redefine only those operations for which the extension is significant while reusing the rest.

There are already several implementations of Haskell-like languages around, most notably the Glasgow [2] and Chalmers [3] Haskell Compilers, Mark Jones's Gofer and Hugs systems [4], Niklas Röjemo's Nearly Haskell Compiler [5], and Clean [6]. So why on earth would anyone want to develop yet another language? The reason is that none of these existing implementations exactly fits our needs. Though one of the goals in the design of GHC was to provide a modular foundation that other researchers can extend and develop, the Glasgow compiler is much too large and complicated to be used for education. Gofer (and Hugs) are small and simple, but they are written in C. The NHC system is written in Haskell, but is optimized for space efficiency. Lastly, the Clean compiler is written in C and optimized for both fast compilation and fast target code. Except for Gofer/Hugs, none of the above implementations has a proper PC/Windows based programming environment.

[1]A sufficient condition would be a killer application in which functional programming is clearly superior. We anticipate that using functional languages, instead of Visual Basic, as glue for COM components could be such a domain.

We recognize that building a complete implementation from scratch is a lot of work, and hard to accomplish with the limited resources we have. Therefore we decided to piggyback on existing infrastructure as much as possible by compiling Mondrian into Haskell. Later we hope to target at Henk [24] as a common intermediate language with GHC. At this moment Mondrian is in its embryonal stage, and it might take a long time before it is delivered. Even so, the ideas ventilated in Mondrian have already prompted the development of a typed intermediate language Henk [24] and helped to keep Haskell on its toes [23, 20, 21].

3 Classes and algebraic data types

Mondrian unifies algebraic types and type classes into a single, simple-minded, object-oriented class mechanism. Dictionaries that are used in the Haskell implementation of type classes [14, 9, 30] are nothing more than algebraic data types with polymorphic fields. Structures with polymorphic components are completely standard and already supported by several existing Haskell implementations [18, 28]. When class dictionaries are first class values, algebraic data types can be modeled by a root class with a subclass for each alternative. Let's see how this all works in some more detail.

As our running example we use a class `Student` that models students that carry a name and an address.

```
class Student where {name, address :: String}
```

We can use the constructor `Student{...}` to construct instances of the `Student` class (values of type `Student`) by defining the fields of interest between the braces as in the following definition of some arbitrary student:

```
aStudent
 = Student
    { address = "4711 NW One Way Street"
    ; name = "John Doe"
    }
```

Instances of `Student` can be updated non-destructively. When John Doe moves, we might want to change his address

```
aStudent{address = "1478 SW Osprey Drive"}
```

An update of an object creates a new copy in which the specified fields are replaced by the indicated values.

Besides construction and update we can also do pattern matching on class instances. Function `showStudent` does a pattern match on the constructor `Student{...}` to destruct a student value. It maps an instance of the class `Student` to a string[2].

[2]In an object-oriented setting one would define this function as a method of the Student class, but here we want to discuss the traditional "pattern matching" view of first class classes

```
showStudent :: Student -> String
showStudent Student{n = name; a = address}
 = n ++ ", " ++ a
```

A field binding is very much like a normal binding found in a let- or where-clause. In a pattern match the equation `n = name` binds the variable `n` to the actual value of the field `name`, in a construction or update the equation `name = "John Doe"` binds the field `name` to the string `"John Doe"`. There should be only one way to bind variables. In Mondrian therefore a field pattern introduces a mutually recursive set of bindings whose scope is the right-hand side of the definition in which they occur. In a field construction the scope is restricted within the braces. Here is an alternative way to write the previous example:

```
showStudent Student{s = name ++ ", " ++ address}
 = s
```

Field bindings are recursive, thus for example `Student{name = name}` binds `name` to $\perp$, as it assign the value of the `name` field of the instance under construction to the `name` field of the instance under construction. This is tedious when you want to update a field with a value that depends on its previous value. But overall regularity is more important than occasional inconvenience.

Mondrian allows Haskell-style punning of field bindings. The following definition of function `showStudent`

```
showStudent :: Student -> String
showStudent Student{name; address}
 = name ++ ", " ++ address
```

is a shorthand for the first definition we gave for function `showStudent`. In general, an object construction or update that uses punning on field `f`

```
foo{p = e; f}
```

is translated into

```
let {f' = f} in foo{ p = e; f = f' }
```

where `f'` is a fresh identifier.

In a pattern match

```
case e of { ... ; foo{ p = e; f } -> b ; ... }
```

a pattern that uses punning is translated into

```
case e of
{ ...
; foo{ p = e; f' = f } -> b where f = f'
; ...
}
```

again, with `f'` a fresh name.

As in a Haskell class declaration, we may define a default value for a method when defining a class. For example we can give instances of `Student` the default name `"John Doe"`.

```
class Student where
  { name :: String; name = "John Doe"
  ; address :: String
  }
```

In this case when the field `name` is omitted in the construction of a student instance, its `name` method is automatically initialized to the default string `"John Doe"`. In general, given a class declaration

```
class C where {f :: A; f = a; g :: B}
```

an object construction `C{p = e}` where field `f` is not bound in pattern `p`, is translated into

```
let {f = a; p = e} in C{f; g}
```

3.1 Subclasses but no subtyping

We can make a subclass `PhD` of students that are working towards a PhD thesis by extending the `Student` class with an extra field `topic` that contains the topic of the student's PhD thesis.

```
class Student => PhD where { topic :: String }
```

An instance of `PhD` has all the methods of `Student` plus the extra method `topic` of type `String`, for example

```
jl
 = PhD{ name = "Jeff Lewis"
      ; address = "OGI"
      ; topic = "ADL"
      }
```

An object of class `PhD` is permitted wherever a `Student` object is required. In particular an instance of `PhD` can match a pattern that expects a `Student`. Thus the call `showStudent jl` evaluates to the string `"Jeff Lewis, OGI"`, even though function `showStudent` was defined by pattern matching on the superclass `Student{...}`. By taking this decision, we nose-dive right into the sticky tarpit of subtyping.

Suppose we define a function `changeAddress` that updates the address of any object that has an address field.

```
changeAddress this = \new -> this{address = new}
```

When we use function `changeAddress` to update the address of a `PhD` student, we expect that the resulting value still is a `PhD` student, i.e., the value of `changeAddress jl "PSU"` is `jl{address = "PSU"}` and has type `PhD`. So the question is what should be the type of function `changeAddress`? On the one hand the argument type should be `Student`, so that it accepts every subclass instance with an `address` field. On the other hand the result type should definitively *not* be `Student` as this means that we lose static knowledge of the fact that updating the `address` of an object does not modify its dynamic type.

The design of a typesystem that deals with subtyping, higher-order functions, and objects is a formidable challenge, and requires heavy-duty type theoretical apparatus [7]. For the sake of simplicity we took an extreme design decision: Mondrian effectively has no notion of subtyping! This implies that a value of a superclass can be used whenever a value of one of its subclasses is required. We recognize that this choice may give rise to runtime errors. However, we only allow runtime errors that can be captured by a dynamic type-check on the constructor of an object. Such errors are of the same nature as division by zero, incomplete case distinction and other runtime errors caused by partial functions in Haskell.

Another reason for taking such an extreme cut at subtyping is that this is dictated by the translation of classes. The translation of Mondrian classes into Haskell proceeds in two stages. In the first stage, the class hierarchy is flattened by taking its transitive closure (sometimes this process is called *dictionary flattening*). The methods of a class `class S => R where ...` are added to any class `class (T,R) => C where ...` that has `R` as one of its superclasses, after which `R` is replaced by `S` in the set of superclasses of `C`, i.e. `class (T,S) => C where ...`. Provided that any two classes that appear in the same superclass set have a common superclass, this process partitions the set of all class declarations into groups of a root class `class R where methodsR` and a number of immediate subclasses `class R => C where methodsC`. Such a group can immediately be translated into a normal data declaration `data R = R methodsR | ... | C methodsR methodsC`. For example, the class hierarchy

```
class P where {a :: A}
class P => Q where {b :: B}
class P => R where {c :: C}
class (Q,R) => S where {d :: D}
```

is flattened into the algebraic data type

```
data P
 = P{a :: A}
 | Q{a :: A, b :: B}
 | R{a :: A, c :: C}
 | S{a :: A, b :: B, c :: C, d :: D}
```

If we have an inheritance tree of depth d, then the corresponding algebraic data type is a factor d bigger. This economy of writing, together with the fact that when algebraic data types and classes are merged there is one concept less to learn, is an important advantage of Mondrian classes over traditional Haskell.

3.2 Recursive classes

Now that the previous section teached us how Mondrian classes can be transformed into ordinary algebraic data types, it is easy to see how to encode algebraic data types using classes: just construct a class hierarchy that maps onto the desired algebraic datatype. To model recursive data types we need classes that have methods that return class values. The standard example of lists with elements of type `a` follows by declaring a base type `List a` and two subclasses `Cons a` of non-empty lists over type `a` and `Nil a` of empty lists over `a`.

```
abstract class List a
final class List a => Nil a
final class List a => Cons a where
 { head :: a
 ; tail :: List a
 }
```

The class `List a`[3] is only needed as a root class for the subclasses `Nil` and `Cons`, but we do not want to build or use objects of class `List a` itself. This intention is captured by declaring `List` as an `abstract` class. Also if we really want to model sum-of-product data types the classes `Nil` and `Cons` cannot be subclassed any further, so we prevent this by declaring them `final`. However, this does *not* prevent us from adding new subclasses to `List a` itself. To cope with this Mondrian uses the normal Haskell data type syntax, i.e.

```
data List a
 = Nil
 | Cons{ head :: a ; tail :: List a }
```

List objects can be built by using the constructor functions `Nil` and `Cons`, for example

```
ones = Cons{ head = 1; tail = ones }
```

Since lists are so common, Mondrian follows Haskell in providing the usual "square bracket delimited elements separated by commas" syntactic sugar for lists.

An interesting consequence of the encoding of Haskell data types in terms of an abstract base class and subclasses for each alternative is that the type of partial functions on data types can become more informative. For example the function `head` has type `Cons a -> List a`, which indicates that it expects a non-empty list and returns a possibly empty list. As in Haskell we can apply `head` to an empty list (as the least common superclass of `Cons a` and `Nil a` is `List a`) with a runtime error as result. Thus, also in Haskell "message not understood" runtime errors are not prevented by the type checker, and Haskell suffers from the very same problem as Mondrian's lack of proper subtyping. In other words, the Haskell type checker does not reject partial functions defined using pattern matching, something that is not normally perceived as a *typing* issue.

3.3 Late binding

Overloading using typeclasses is one of Haskell's most distinctive features. When a member function of a class is used in a definition, this shows up as a constraint in the type of that definition. This definition is then considered as an implicit member function of the class in question as well, so that when it is used in other definitions the "infection" spreads. For example if we start with the (simplified) class `Eq` with member function `(==)`:

```
class Eq a where { (==) :: a -> a -> Bool }
```

and use it in the linear search function `search`

```
search a = foldr ((||) . (== a)) False
```

the inferred type for `search` will be `Eq a => a -> [a] -> Bool`. Any function that subsequently uses `search` also gets a constrained type involving `Eq`.

From an implementation point of view, the type `search :: Eq a => a -> [a] -> Bool` tells us that function `search` is transformed by the compiler into function that passes the dictionary for `Eq a` as an explicit argument:

```
search' :: Eq a -> a -> List a -> Bool
search' Eq{(==)} a
 = foldr ((||) . (== a)) False
```

[3] If a class has no methods, we may omit its `where {}` part.

Haskell style overloading can be regarded as just syntactic sugar for implicit dictionary passing. But in practice, this sugar is genuinely needed. Without it, programming would become a bitter activity.

In reality matters are slightly more complicated than sketched just now. Haskell uses class- and instance- declarations to *reduce* (simplify) contexts inferred by the type checker. Suppose we define the function null as = (as == []), then Haskell infers the type null :: Eq a => [a] -> Bool, by using the instance declaration

```
instance Eq a => Eq [a] where
  { [] == []            = True
  ; (a:as) == (b:bs) = a == b && as == bs
  ; as == bs            = False
  }
```

to reduce the needed context Eq [a] to Eq a. Using the same mechanism, contexts can be eliminated by using primitive instance declarations such as instance Eq Int where .... To see how class declarations are used to simplify contexts in Haskell, we introduce a subclass Ord a of class Eq a:

```
class Eq a => Ord a where
  { (<=) :: a -> a -> Bool
  }
```

Since every instance of Ord a has an (==) method, we can simplify the constraint (Eq a, Ord a) to Ord a. For a more extensive discussion on type classes we refer to a proposal for liberating the Haskell class system [23], which also contains supplementary pointers to relevant work.

When used as variables, field names such as (==) or address of class Student act as overloaded functions of type Eq a => a -> a -> Bool and Student => String respectively. Dictionary translation proceeds as in Gofer [19] or [23, choice 2c or 2d], that is, no instance specific context reduction takes place. Not surprising, given the fact that Mondrian has no explicit instance declarations. In Mondrian instance declarations are just value definitions for (functions from objects to) objects.

We lied when we said that Gofer does not use instance declarations to simplify contexts. In fact, Gofer uses ground instance declarations to eliminate contexts. Otherwise there it would simply be impossible to evaluate expressions that still expect a dictionary argument. In Mondrian, the programmer has to supply the right dictionaries explicitly. To accommodate for this, Mondrian provides field selector functions for every class method. For example, the class declaration Student automatically introduces a toplevel postfix operator (.address) of type Student -> String defined as

```
(.address) :: Student -> String
Student{address}.address = address
```

Future subclassing of Student respects the expected behavior of field selection due to the way pattern matching works; it extracts the address field of any Student instance.

Selector functions are not restricted to member functions; *any* overloaded function f :: P => t can be used as a selector (.f) :: P -> t. Conversely, any (non-member) function that is defined as a selector (.f) :: P -> t can be used as an overloaded function f :: P => t. The only difference with class methods is that the latter are automatically provided.

In Haskell the standard prelude function sortBy takes an explicit comparison operator as an argument, instead of relying on overloading. Why? Because in Haskell it is not possible to have more than one instance declaration for the same instance type. Using Mondrian's selector function mechanism we can use implicit dictionary passing, and at the same time have full control over the dictionary argument.

```
reverse :: Ord a -> Ord a
reverse ord = ord{ (<=) = flip (ord.(<=)) }

sort :: Ord a => [a] -> [a]
sort = ...

revsort :: Ord a => [a] -> [a]
ord.revsort = (reverse ord).sort
```

We believe that by eliminating instance declarations and keeping implicit dictionary passing, Mondrian retains the good aspects of Haskell's type classes while avoiding the bad such as unresolved overloading, overlapping instances etc. In fact, we can completely separate the idea of late binding from that of typeclasses, something that we hope to explore in the future.

3.4 Multiple argument classes

In Gofer subclasses can have more arguments than their base class. For example the class of state monads has an additional state parameter s and an update operation update :: (s -> s) -> m s on that state

```
class Monad m where
  { bind :: m a -> (a -> m b) -> m b
  ; result :: a -> m a
  }

class Monad m => StateMonad m s where
  { update :: (s -> s) -> m s }
```

This provides a challenge to the typesystem. The problem is that after we have used an object of the class StateMonad m s as an object of its superclass Monad m we have not only lost static knowledge of the type s, but also the dynamic knowledge. It is not possible to recover type s by pattern matching on the constructor StateMonad. Another route via which this problem shows up is in the flattening of classes. We cannot translate the above class hierarchy into

```
data Monad m
  = Monad{ bind :: m a -> (a -> m b) -> m b
         ; result :: a -> m a
         }
  | StateMonad{update :: (s -> s) -> m s}
```

since function update is not polymorphic in s. Treating s as an existential type looks more promising. But, to keep things simple we demand that subclasses have the same type arguments as their superclass.

3.5 Differences with Haskell

There are some differences between Mondrian's classes and Haskell's data types with field labels. In Haskell field binders are separated by commas and class signature declarations and default definitions by semicolons. In Mondrian we have used the class instead of the data type syntax.

In Haskell a constructor with labeled fields may be used as an ordinary constructor. Following the 0-1-∞-*rule* [25], Mondrian has only classes with labeled fields. If you do not want to write field bindings, you can use punning instead.

Field bindings in Mondrian are consistent with value definitions and pattern bindings. Variables being bound are always on the left of an =-sign. To construct a `Student` instance with the `name` field bound to the value `"John"` one writes `Student{name = "John"}`. To destruct a `Student` and bind the value of its `name` field to variable `n` one writes `Student{n = name}`. In Haskell one would write in this latter case `Student{name = n}`.

In order to unify all the contexts in which value bindings may occur we made a rigorous decision: there are *no* irrefutable patterns in Mondrian. The only exception are products, which are irrefutable by default. Whenever you write a pattern it will be matched. Thus the function call `joe Student{name = "John"}` will fail, when function `joe` is defined as

```
joe x = let Student{name = "Joe"} = x in True
```

In Haskell `joe Student{name = "John"}` will succeed and return `True`. In Mondrian, the dynamic semantics of a (nonrecursive) let-binding `let p = e in e'` is the same as `(\p -> e') e`. This restores the *Principle of correspondence* [33], which is not valid in Haskell. Lazy matching on products is just enough to make mutually recursive variable bindings work.

To make conforming pattern bindings more convenient, Mondrian has a pattern-aware `where` clause, that skips to the next alternative when one of its pattern binding fails, instead of just giving up as a let does. Thus the function call `joe Student{name = "John"}` will return `False`, when function `joe` is defined as

```
joe x = True where Student{name = "Joe"} = x
joe x = False where Student{name} = x
```

In general we have that a definition `f p = a where q = b; f r = c where s = d` is equivalent to `f x = case x of p -> (case b of q -> a; _ -> f' x); _ -> f' x` with `f' x = case x of r -> (case d of s -> c)`. By this decision, the operational behaviour of where clauses has become more complicated than in Haskell, but we think that this is justified by an increase in expressive power. At the time of writing this paper, Simon Peyton Jones [21] has proposed to extend Haskell style guards from an expression to a list of qualifiers as in a list comprehension. This gives roughly the same expressive power as pattern-aware `where` clauses, but at a considerable lower cost. In this proposed extension to Haskell function `joe` would read:

```
joe x | Student{name = "Joe"} <- x = True
joe x |         Student{name} <- x = False
```

4 Object encoding

Faithful encoding of objects using records and functions is known to be very difficult [7]. We use the standard example of `Point` and `ColorPoint` to illustrate the naive encoding of objects that first comes to mind. Since we do not have subtyping, this solution works. If a method of some class wants to access or modify some of its methods, it takes itself as an explicit argument. This is similar to type bound procedures in Oberon [27], where class methods take the receiver object as an explicit argument. In Mondrian, we write:

```
class Point where
   { pos :: (Int,Int); pos = (0,0)
   ; move :: Point -> (Int,Int) -> Point
   ; move this{(x,y) = pos} (dx,dy)
      = this{pos = (x+dx,y+dy)}
   }
```

If we later extend `Point` to `ColorPoint` by adding a `color` field

```
class Point => ColorPoint where
   { color :: Color; color = White
   }
```

then the `move` method of class `Point` moves an instance of class `ColorPoint` as well since the definition of method `move` assumes only the presence of a `pos` field.

When invoking the method `move` we must pass the point to be moved itself as an additional argument: `p.move p (dx,dy)`. Here we recognize the doubling combinator, since we will use it so often we introduce special syntax for method invocation as well:

```
x#f = x.f x
```

Moving a point `p` can now be written elegantly as `p#move (dx,dy)`.

Besides the two class modifiers `abstract` and `final` that we already encountered, we also have two method modifiers. The modifier `protected` indicates that a field is only visible in its class and all subclasses thereof, and thus it may never appear on the left of an =-sign in either a field match, update or construction *outside* a class declaration. The modifier `static` indicates that a field is fixed statically at compile time, and cannot be updated at run time, and thus may never appear to the left of an =-sign in a field construction or update. A `static` method without a default definition is rather useless. In the class `Point` we might want to restrict access to the position `pos` by declaring it `protected` and prevent the `move` method from being changed by declaring it `static`.

```
class Point where
   { protected pos :: (Int,Int)
   ; pos = (0,0)
   ; static move :: Point -> (Int,Int) -> Point
   ; move this{(x,y) = pos} (dx,dy)
      = this{ pos = (x+dx,y+dy) }
   }
```

The advantage of static methods is that they are shared by all class instances. The disadvantage of declaring a method `static` is that we cannot overwrite it in different instances.

If we want to change it anyway we have to define a new subclass that provides a new default definition for the method.

Besides the notion of `this`, most object-oriented languages also know the notion of `super`. Just as with `this` Mondrian has no hidden mechanism for providing `super`, but it is easy enough: one has access to the default methods of a class `C` by just using `C{}`. Unfortunately, this is not robust, since the choice of `C` is context dependent.

We think that Mondrian provides enough support to express most of the typical object-oriented programming idiom. This falls short in that the encoding is not typesafe and that it is perhaps a bit weird that updating an object is implemented by constructing a whole new object and leaving it up to the garbage collector to eventually remove the original object. This is inevitable because we have no implicit updatable state in a pure functional language.

4.1 Eight-queens in Mondrian

In his book on object-oriented programming [12], Timothy Budd gives object-oriented programs for the well know eight queens problem in several languages. It turns out that the program in Mondrian is more concise than any of the programs given in the book. Don't get this wrong, the example is not supposed to convince you of the fact that the Mondrian solution is more concise than the one in Haskell [11] (which, by the way, is also a perfectly valid Mondrian program):

```
queens number_of_queens
 = qu number_of_queens
   where
     qu 0     = [[]]
     qu (m+1) = [ p++[n]
                | p<-qu m
                , n<-[1..number_of_queens]
                , safe p n
                ]

safe p n
 = all not [ check (i,j) (m,n)
           | (i,j) <- zip [1..] p
           ]
   where m = 1 + length p

check (i,j) (m,n)
 = j==n || (i+j==m+n) || (i-j==m-n)

q = putStr . layn . map show . queens
```

although you might argue about which one is more readable.

A `Queen` carries three instance 'variables'; the `row` and `column` she is in and her left `neighbor`. A queen furthermore knows how to move herself to a safe position (using `getSafe` and `advance`), and whether she can attack a given position to her right (using `canAttack`).

```
class Queen where
   { row, column :: Int
   ; neighbor :: Queen
   ; static getSafe :: Queen -> Queen
   ; static canAttack :: Queen -> Position -> Bool
   ; static advance :: Queen -> Queen
```

A queen starts at row 0.

```
   ; row = 0
```

If a queen can be attacked by her left neighbor at her current position, she advances quickly to the next position

```
   ; getSafe this{ row; column
                 ; neighbor
                 }
      = if neighbor#canAttack (row,column)
        then this#advance
        else this
```

A queen can attack another queen at position `(r,c)` if that queen is on the same row or diagonal, or if her neighbor can attack position `(r,c)`. So actually, the method `canAttack q (r,c)` determines wether queen `q` *and* all the queens to her left can attack position `(r,c)`.

```
  ; canAttack this{ row; column
                  ; neighbor
                  } (r, c)
     = or [ row == r
          , row+(c-column) == r
          , row-(c-column) == r
          , neighbor#canAttack (r,c)
          ]
```

If a queen must advance to the next position but has reached the end of the board, she has been unable to find a safe position in this configuration. So she asks her immediate left neighbor to advance as well and starts again at `row = 0`. Otherwise she just advances one step. In either case, she makes sure the new position is safe.

```
   ; advance this{ row; column; nb = neighbor }
      = (case row + 1 'mod' n of
          { 0 -> this{ row = 0
                     ; neighbor = nb#advance
                     }
          ; r -> this{ row = r }
          }
        )#getSafe
```

In this method we see a disadvantage of mutually recursive field bindings. We are forced to give the old value of `neighbor` a name.

To generate the solutions, we put the queens on their column on the board one by one and make sure they are in a safe position before we place the next queen.

```
board
 = foldr (\c n
           -> Queen{ column = c
                   ; neighbor = n
                   }#getSafe)
         ladyDi
         [n-1,n-2..0]
```

The leftmost queen is special, she cannot move or attack other queens.

```
ladyDi
 = Queen{ getSafe = id
        ; advance = id
        ; canAttack = \_ _ -> False
        }
```

We cycle through all possible solutions by repeatedly advancing the rightmost queen.

```
solutions
 = ( unlines
   . map (#show)
   . scanl (#) board
   ) (repeat advance)
```

The definition of the show function on queens is left to the reader.

5 Various novelties

So much for classes in Mondrian. We now continue to describe the more mundane aspects in which Mondrian differs from Haskell.

5.1 Scoped typevariables

In Haskell, a type expression like `b -> (a,b)` is interpreted as `∀ a, b :: b -> (a,b)` in any context. This cripples the usefulness of type signatures for local definitions. The following definition is rejected by the Haskell typechecker because the given type for g is assumed to be `∀ a, b :: b -> (a,b)`, which is indeed too general.

```
f a = g a where {g :: b -> (a,b); g b = (a,b)}
```

In Mondrian we can restrict the type of g to be monomorphic in a but polymorphic in b, i.e. `g :: ∀ b . b -> (a,b)` by adding an explicit type declaration `f :: a -> (a,a)` for function f as well. This causes the type variable a in the inner declaration `g :: b -> (a,b)` to become bound to the type variable a in the outer declaration `f :: a -> (a,a)`. This convention is consistent with classes with polymorphic fields. In the declaration of `Monad m` above, the field `result :: a -> m a` is polymorphic in a, but monomorphic in m. There might be other ways of indicating the scope of type variables, but we feel that the given alternative fits quite comfortably with the current Haskell approach for providing type signatures.

The usual implementation of Hindley-Milner typechecking already deals with scoping of type-variables internally. Type-variables that are not in scope of an outer quantifier are called *generic type* variables. We only provide a means for the programmer to specify which type variables are to be considered non-generic. Standard ML has had scoped type-variables for years [26, §4.10], and it won't be long before GHC supports some form of scoped type variables [20] too.

5.2 Type abstraction and parametrization

In Haskell the abstraction mechanisms on the level of types fall short when compared to those on the level of expressions. This is a pity since types play such an important role in the programming process. Just as we can use let-expressions on the term level to give a name to program fragments that occur multiple times and use λ-expressions when we do not want to name a function, we should be able to this on the type level too. This need becomes even more pressing when using higher order type variables. For example, consider the following class of generalized trees:

```
class Tree f a where
   { nodes :: f (Tree f a)
   ; value :: a
   }
```

We can then define binary trees by saying `type BinTree a = Tree (\x -> (x,x)) a`, if we want Rose trees, we just write `type RoseTree a = Tree List a`.

There is already a theoretical framework that allows the same language on the type level as on the expression level; it is called $F\omega$. When you consider Haskell as a fragment of the polymorphic lambda-calculus for which type-reconstruction is decidable, you may consider Mondrian as a fragment of $F\omega$ for which type-reconstruction is decidable. For the moment we are conservative and allow only simple functions on the level of types and no classes and pattern matching.

Again we have taken a very pragmatic approach to avoid ending up with a complicated type system. The typechecker tries to normalize type expressions before unifying them. It does not try to do higher-order unification. Because type expressions are well-kinded, the reduction of those type expressions that are legal in Haskell does terminate. Mondrian allows more general recursive type expressions, by allowing recursion on types without being 'protected' by a constructor function. This introduces the risk of nontermination of the type-checker. Now, we can define the second order type constructor `Rec :: (* -> *) -> *` that takes the fixed point of a type constructor as follows

```
type Rec f = f (Rec f)
```

A type expression like `Rec (\x -> x)` does not terminate, but the type expression `Rec (\x -> (Int,x))` does, and denotes the type of infinite `Int` streams.

5.3 Modules

In spirit with keeping the language as small as possible, Mondrian's module system is the smallest subset of Haskell's module system that is powerful enough so that almost every Haskell module can be simplified into a Mondrian-style module.

A module is a collection of type and function declarations that serves as a unit of compilation and delimits a namespace. Mondrian modules cannot be mutually recursive. A module can offer its declarations to other modules by exporting them. A module can use declarations exported by other modules by importing them. When a module imports another module it automatically imports *all* entities exported by that module. All imported names have to be qualified.

In so far there is no difference between Haskell modules, except for the ban on mutually recursive modules. We like the modesty of the Haskell module system, but not its syntax. In Haskell the identifiers to be exported are repeated as 'arguments' in the module heading, while imported modules are written as 'declarations' in the module body. Dually, in Mondrian imported modules are written as arguments in the module heading and exported identifiers are marked `public` at their definition. Identifiers that are not marked `public` or `protected` are visible to any other definition within their defining module, but are invisible outside that module.

```
module RecTypes where
  { public type Rec f = f (Rec f)
  ; in  :: f (Rec f) -> Rec f
  ; out :: Rec f -> f (Rec f)
  ; public in  = id
  ; public out = id
  }
```

A type declaration in a module is always opaque outside that module. Exporting the type only allows you to use it in a type-expression in the importing module. In Gofer terms it is like a restricted type synonym with all functions that are defined within the module in its in list. Thus in a module that imports `RecTypes` the type constructor `Rec` is a constant and the functions `in :: f (Rec f) -> Rec f` and `out :: Rec f -> f (Rec f)` must be used as explicit type coercion functions.

6 Conclusion and Related work

We started creating Mondrian by trying to generalize some of the constructs of Haskell. At every point in the design process, we tried to keep things as simple as possible. As we showed in this paper, we ended up with a small and elegant language, with surprisingly powerful object-oriented-like properties. Some of the thus begotten ideas are already adapted by others. But Mondrian is still in its childhood phase: we think we can still generalize some of the constructs, and make the language even more expressive, simple and clear.

Our work differs from related work on object-oriented extensions for SML [32, 31] in that adding OO features to Haskell was not our goal per se, but rather surfaced as a consequence of the fact that we unified and simplified Haskell's notions of type classes and algebraic data types. Mondrian has no built-in support for classes and objects, but requires that the programmer takes care of the object encoding. Notwithstanding this conceptual difference, we think our work could greatly benefit from carefully studying these alternative approaches.

Acknowledgments

Thanks to Arjan van Yzendoorn, Joost van Dijk, Simon Peyton Jones, John Matthews, Dennis Gruijs, Mark Shields, Thomas Nordin, Daan Leijen, and very special thanks to the ICFP'97 and Haskell workshop'97 referees for comments on earlier drafts. We acknowledge gratefully the support of the Oregon Graduate Institute during our visit, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References

[1] `http://www.cs.ruu.nl.`

[2] `http://www.dcs.gla.ac.uk/fp/software/ghc/.`

[3] `ftp://ftp.cs.chalmers.se/pub/haskell/.`

[4] `ftp://ftp.cs.nott.ac.uk/pub`

[5] `ftp://ftp.cs.chalmers.se/pub`

[6] `http://www.cs.kun.nl/~clean/.`

[7] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[8] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[9] L. Augustsson. Implementing Haskell overloading. In *Proc. FPCA*, 1993.

[10] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[11] R. Bird and P. Wadler. *Introduction to functional programming.* Prentice-Hall, 1988.

[12] T. Budd. *An Introduction to Object-Oriented Programming (2nd edition).* Addison-Wesley, 1997.

[13] J. Peterson (editor). Report on the programming language HASKELL version 1.4. `http://www.haskell.org.`

[14] C.V. Hall, K. Hammond, S.L. Peyton Jones, and P. Wadler. Type Classes in Haskell. In *Proc. ESOP*, volume LNCS 788, 1994.

[15] J. Hughes and J. Sparud. Haskell++: An Object-Oriented Extension of Haskell. In *Proc. Haskell Workshop, La Jolla, California*, YALE Research Report DCS/RR-1075, 1995.

[16] R. J. M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, April 1989.

[17] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *jfp*, 5(1):1–37, jan 1995.

[18] M.P. Jones. From Hindley-Milner Types to First-Class Structures. In *Proc. Haskell Workshop, La Jolla, California*, YALE Research Report DCS/RR-1075, 1995.

[19] M.P.J. Jones. *Qualified Types: theory and practice.* Cambridge University Press, 1994.

[20] Simon Peyton Jones, April 1997. personal communication.

[21] Simon Peyton Jones. A new view of guards, April 1997. `http://www.cse.ogi.edu/~simonpj/guards.html.`

[22] S.L. Peyton Jones, C.V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 1993.

[23] S.L. Peyton Jones, M.P. Jones, and E. Meijer. Type classes: an exploaration of the design space. In *Proc. Haskell Workshop*, 1997.

[24] S.L. Peyton Jones and E. Meijer. Henk: a typed intermediate language. In *Proc. Types in Compilation Workshop*, 1997.

[25] B.J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* Holt, Rinehart and Winston, 1987.

[26] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT press, 1990.

[27] H. Mössenboeck and N.Wirth. The Programming Language Oberon-2. *Structured Programming,* 12(4):179–195, April 1991.

[28] M. Odersky and K. Läufer. Putting Type Annotations to Work. presented at the Newton Institute Workshop on Advances in Type Systems for Computing, August 1995.

[29] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL,* 1997.

[30] J. Peterson and M.P. Jones. Implementing Type Classes. In *Proc. PLDI,* 1993.

[31] D. Rémy and J. Vouillon. Objective ml: A simple object-oriented extension of ml. In *Proc. POPL,* 1997.

[32] John Reppy and Jon Riecke. Simple objects for Standard ML. In *Proc. PLDI,* 1996.

[33] D. Schmidt. *The Structure of Typed Programming Languages.* MIT press, 1994.

Reactive Objects in a Functional Language
An escape from the evil "I"

Johan Nordlander *Magnus Carlsson*

Department of Computing Science
Chalmers University of Technology
S - 412 96 Göteborg, Sweden
Email: {nordland,magnus}@cs.chalmers.se

Abstract

We present an extension to Haskell which supports reactive, concurrent programming with objects, *sans* the problematic blocking input. We give a semantics together with a number of programming examples, and show an implementation based on a preprocessor and a library implementing seven monadic constants.

1 Introduction

With the advent of Haskell 1.3 the monadic I/O model has become well established [PH96]. At the top level, a Haskell program is now a sequence of imperative commands that transforms a state consisting of the real world and/or some program state into a final configuration. In a pure state transformational approach, carrying a monolithic program state around is likely to complicate modular design; however, this problem can to a large extent be circumvented by introducing first-class references in the monadic framework [LJ94]. Taken together, these additions make the resulting Haskell programs — on the top level at least — more and more reminiscent of programs written in traditional imperative languages like Pascal.

Just like I/O in traditional languages, though, straightforward monadic I/O imposes a rather rigid structure on environment interaction. Especially, when a program says "read input", the outside world has nothing to do but to follow order, since otherwise execution is effectively stuck. Examples of this scenario are numerous, the average computer user is probably all too familiar with primitive programs that continuously alternate between telling the user what to do and reading back responses. It goes without saying that putting programs structured this way in the context of an inherently concurrent and nondeterministic environment is bound to be problematic.

In their POPL '96 paper, Peyton Jones, Gordon, and Finne identify the need for *semantically visible concurrency* to alleviate these problems [PGF96]. Their proposed language, Concurrent Haskell, allows multiple execution threads to be initiated, and provides a mechanism for synchronisation and communication between such threads via atomically-mutable state variables. Thus, when one thread blocks for input, another thread can be created that maintains the capability of interacting with the environment in alternative ways. As with the sequential I/O model of Haskell, a central property of Concurrent Haskell is a stratified semantics that limits non-determinism and state-manipulation to the top-level of a program, while leaving the purely functional semantics of expression evaluation unaffected.

Undeniably, the primitives offered by Concurrent Haskell are at a very low level. This choice has been deliberate, though, since the purpose of these primitives is really to act as basic building blocks in the construction of various higher-level abstractions. However, as far as mastering concurrency is concerned, this situation is not very much different from the challenge that faced Algol programmers equipped with pointers and semaphore libraries three decades ago [Dij65]. That is, while important properties like liveness and mutual exclusion certainly *can* be maintained in a Concurrent Haskell system, there is nothing in the language that imposes such a structure on programs. This is a rather paradoxical quality for a purely functional language, which has deliberately abandoned low-level features like assignments and jumps in order to impose good structure on ordinary, non-concurrent computations.

What one ideally would like to see is a language that, in the spirit of Concurrent Haskell, combines the virtues of explicit concurrency and unobstructed functional programming, but which also commits itself to some top-level form that can be found both abstract and flexible, as well as intuitively appealing. In this paper we make an experimental attempt towards this goal, by means of a Haskell extension based on the notion of *reactive objects*. As a slight nod to the object-oriented community, we call our extension O'Haskell.

2 Reactive Objects

Our approach to concurrency in O'Haskell starts out from an identification of a state and a process. This means that there can be no state-container without an associated thread of control in our language, neither can there be any control threads without associated local states. Such a state/process combination is called an *object*, and in the sequel we will use this term interchangeably with our notion of a process.

The code of an object is an expression in the monad `O s ()`, which is a refinement of Haskell's interaction monad

IO () into an *reacting state monad* with a state type s.[1] To improve the readability of expressions in this monad, we provide an extension to the do-syntax of Haskell 1.3 that allows the state of an object to be inspected and assigned in a Pascal-like manner.

The crucial difference between the IO monad and our replacement is now the following: whereas the value return () represents a terminated program/process in Haskell/Concurrent Haskell, we will interpret this value as an *inactive process* that just passively maintains its state. Such a process may indeed become runnable in the future, since the monadic code of an object can be extended with new fragments (using the standard operator >>) by the reception of *messages*. These messages can be of two forms: either an asynchronous action, that lets the sender continue immediately and thus introduces concurrency, or a synchronous request, which allows a value to be passed back to the waiting sender. In both cases, the receiving object reacts by starting a reduction of its updated monadic expression, which will, in the absence of a non-terminating reduction, eventually restore the receiver to an inactive state again. A significant feature of this scheme is that the actions and requests of an object implicitly form a critical region.

Objects are introduced using a template, which defines the initial state of an object together with a *communication interface*. Executing a template expression in the O monad creates a new object that is an instance of the template, which is the O'Haskell equivalent of forking off a process. The following code fragment defines a template for a simple counter object:

```
newCounter =
   template
      val := 0
   with
      ( action
          val := val + 1
      , request
          return val
      )
```

In this case, the communication interface consists of an asynchronous action that increments the counter, and a synchronous request for reading its current value. The silly program below illustrates how a counter object is created and then passed an *increment* message, before its current value is requested. At the end of the sequence the new counter is actually forgotten, and will become garbage.

```
main = do
   (inc,read) <- newCounter
   inc
   one <- read
   return ()
```

Synchronous and asynchronous message sending are the only communication primitives offered by O'Haskell, and neither of these blocks the sender more than temporarily.[2] This means that indefinite blocking for input is confined to the *passive case only*, where *any* defined message may be received. As a consequence, an O'Haskell program cannot impose any order on the events it is set to handle, and the liveness of a system will be upheld by default. These are properties that we consider extremely important for the robustness of a concurrent system. We actually take the absence of blocking input commands as our definition of the term *reactive*, and consider it to be the most significant feature of our concurrency proposal. We will return to this issue several times in the coming sections.

[1]The absence of the letter *I* in the name of our monad is *not* coincidental!

[2]See section 6.1 for a further discussion on the validity of this statement.

3 Semantics

In our presentation of the formal semantics of O'Haskell we will assume the existence of an operational semantics for expression evaluation, with a small step reduction relation $\mapsto$. For the sake of completeness we give such a semantics in appendix A, but it should be kept in mind that the exposition that will follow does not depend on the actual choices made in this definition. In particular, our concurrency extension should be equally applicable to a call-by-value language.

3.1 Syntactic transformations

We begin by transforming away the explicit naming of state variables and the template syntax, as shown in figure 1. The translation is parameterised over a tuple-pattern p, which matches the state introduced by the closest enclosing template expression. If no template is in scope, p equals (). Only the interesting cases are shown, the translation extends trivially to the whole expression syntax. We do however restrict variables bound by let and \ to be distinct from the variables in p.

After translation, our concurrency extension is visible only in the presence of seven primitive constants, of which four are the common state monad operations recast to our O type:

```
return  ::  t -> O s t
>>=     ::  O s t -> (t -> O s t') -> O s t'
set     ::  s -> O s ()
get     ::  O s s
```

As usual we write a >> b as a shorthand for a >>= _ -> b.

The remaining three constants correspond to the action, request, and template constructs, respectively. These constants all involve a second built-in type Ref s, which is the type of primitive *reference* values that uniquely determine a particular object at run-time.

```
new  ::  s' -> (Ref s' -> t) -> O s t
act  ::  Ref s' -> O s' () -> O s ()
req  ::  Ref s' -> O s' t -> O s t
```

References in O'Haskell bear some resemblance to Concurrent Haskell's MVars, although it should be observed that a reference is used to identify a process rather than some passive storage location or synchronisation point. This view is influenced by the Actors programming model, which regards the agents/processes as the sole identity-carrying entities during a computation [Agh86].

References are mostly accessed indirectly, however, via the actions and requests an object exports through its communication interface. The common case where an interface

$$
\begin{array}{llll}
a, b & ::= & \ldots \mid \underline{\mathsf{do}}\ \overline{c} \mid \underline{\mathsf{template}}\ \overline{x := a}\ \underline{\mathsf{with}}\ b \mid \underline{\mathsf{action}}\ \overline{c} \mid \underline{\mathsf{request}}\ \overline{c} & \text{Expressions} \\
c & ::= & a \mid x \leftarrow a \mid x := a & \text{Commands}
\end{array}
$$

$$
\begin{array}{lcll}
[\ \underline{\mathsf{do}}\ c\ ]_p & = & \llbracket\ c\ \rrbracket_p & \\
[\ \underline{\mathsf{do}}\ x \leftarrow a\ ;\ \overline{c}\ ]_p & = & \llbracket\ a\ \rrbracket_p \ \mathsf{>\!\!>\!\!=}\ [\ \backslash x \rightarrow \underline{\mathsf{do}}\ \overline{c}\ ]_p & \\
[\ \underline{\mathsf{do}}\ c\ ;\ \overline{c}\ ]_p & = & \llbracket\ c\ \rrbracket_p \ \mathsf{>\!\!>}\ [\ \underline{\mathsf{do}}\ \overline{c}\ ]_p & \\
 & & & \\
[\ \underline{\mathsf{template}}\ \overline{x := a}\ \underline{\mathsf{with}}\ b\ ]_p & = & [\ \mathsf{new}\ (a_1, \ldots, a_n)\ \backslash\mathsf{self} \rightarrow b\ ]_{(x_1, \ldots, x_n)} & \\
 & & & \\
[\ \underline{\mathsf{action}}\ \overline{c}\ ]_p & = & \mathsf{act\ self}\ [\ \underline{\mathsf{do}}\ \overline{c}\ ]_p & \\
[\ \underline{\mathsf{request}}\ \overline{c}\ ]_p & = & \mathsf{req\ self}\ [\ \underline{\mathsf{do}}\ \overline{c}\ ]_p & \\
 & & & \\
\llbracket\ x := a\ \rrbracket_p & = & \llbracket\ \mathsf{set}\ p[a/x]\ \rrbracket_p & \text{if } x \in \mathrm{FV}(p) \\
\llbracket\ a\ \rrbracket_p & = & \mathsf{get}\ \mathsf{>\!\!>\!\!=}\ \backslash p \rightarrow [\ a\ ]_p & \text{if } \mathrm{FV}(a) \cap \mathrm{FV}(p) \neq \emptyset \\
\llbracket\ a\ \rrbracket_p & = & [\ a\ ]_p & \text{otherwise}
\end{array}
$$

Figure 1: Translation of syntactic sugar

completely hides the existence of its underlying reference value is therefore directly supported by our syntactic sugar, by the implicit binding of a variable self in the body of a template. This corresponds to what John Reynolds calls *procedural abstraction*, where information hiding is achieved by partially applying an access procedure to the structure that needs protection [Rey94].[3]

The counter example discussed in the section 2 looks as follows in its desugared variant:

```
newCounter =
   new 0 \self ->
      ( act self
          (get >>= \val -> set (val+1))
      , req self
          (get >>= \val -> return val)
      )

main =
   newCounter >>= \(inc,read) ->
   inc >>
   read >>= \one ->
   return ()
```

3.2 Dynamic semantics

We now turn to the dynamic aspects of O'Haskell. The stratified semantics approach that is an important part of Concurrent Haskell will be repeated here, although the actual language we define will of course be different. We have tried, though, to use a formulation that as far as possible follows the presentation of Concurrent Haskell, in order to simplify comparison.

First we need to specify that >>=, act, and req are strict in their first argument, and that >>= reduces in the usual monadic manner. We do this by extending appendix A with further evaluation contexts and an evaluation rule.

$$
\begin{array}{lll}
\mathcal{E} & ::= & \ldots \mid \mathcal{E}\ \mathsf{>\!\!>\!\!=}\ a \mid \mathsf{act}\ \mathcal{E} \mid \mathsf{req}\ \mathcal{E} \\
\textsc{Bind} & & \mathsf{return}\ a\ \mathsf{>\!\!>\!\!=}\ b \mapsto b\ a
\end{array}
$$

Next we define a small language of process terms, that will enable us to capture the state of a complete O'Haskell system.

$$
\begin{array}{llll}
P & ::= & a_n^b & \text{Atomic process (object)} \\
 & \mid & P \parallel P' & \text{Parallel composition} \\
 & \mid & \nu n.P & \text{Reference generation}
\end{array}
$$

The atomic process a_n^b corresponds to our notion of an object referenced by n, executing a monadic expression a in the state b. This form is restricted by the requirement that if $b :: s$, then $n :: \mathsf{Ref}\ s$ and $a :: \mathsf{O}\ s\ ()$. Furthermore, we require that no pair of objects are tagged with the same reference n. Our reaction rules introduced below all obey these restrictions.

Following the polyadic π-calculus we also adopt the *chemical solution* metaphor, which uses a structural congruence relation $\equiv$ to abstract away from syntactical differences between equivalent process terms [Mil91, BB90]. Using this metaphor we may safely assume that any pair of objects in a system that are willing to interact can be brought together syntactically (as if they were molecules floating around in a chemical solution). The definition of $\equiv$ from [Mil91] is immediately applicable to our process terms; we include the relevant rules in appendix B.

We are now ready to define how a solution of processes may evolve. This is captured by means of a *reaction relation* $\rightarrow$ between process terms. Thanks to the generality of the chemical framework we only have to specify the axioms of $\rightarrow$; a non-deterministic relation between arbitrary complex process terms is automatically obtained by incorporating the structural reaction rules of the π-calculus (recapitulated in appendix C). In the definition of $\rightarrow$ we sometimes take the liberty of matching against atomic processes using a simplified pattern a_n; this should be interpreted as an assertion

[3]Reynolds's contrasting notion, *type abstraction*, would mean exposing the object reference and its state type in the interface and then encapsulating that knowledge within some scope by means of an existential type.

$$\mathcal{M} \quad ::= \quad [\,] \mid \mathcal{M} \mathbin{\mathsf{>\!>\!=}} a$$

$$\begin{array}{llcll}
\textsc{Eval} & a_n & \rightarrow & b_n & \text{if } a \mapsto b \\
\\
\textsc{Set} & \mathcal{M}[\mathsf{set}\ b]^a_n & \rightarrow & \mathcal{M}[\mathsf{return}\ ()]^b_n & \\
\textsc{Get} & \mathcal{M}[\mathsf{get}]^a_n & \rightarrow & \mathcal{M}[\mathsf{return}\ a]^a_n & \\
\\
\textsc{New} & \mathcal{M}[\mathsf{new}\ a\ b]_m & \rightarrow & \nu n.(\mathcal{M}[\mathsf{return}\ (b\ n)]_m \parallel (\mathsf{return}\ ())^a_n) & \\
 & & & \quad \text{where } n \notin \textsc{fn}(lhs) & \\
\\
\textsc{Ego} & \mathcal{M}[\mathsf{act}\ m\ a]_m & \rightarrow & (\mathcal{M}[\mathsf{return}\ ()] \mathbin{\mathsf{>\!>}} a)_m & \\
\textsc{Act} & \mathcal{M}[\mathsf{act}\ n\ a]_m \parallel b_n & \rightarrow & \mathcal{M}[\mathsf{return}\ ()]_m \parallel (b \mathbin{\mathsf{>\!>}} a)_n & \\
\\
\textsc{Req} & \mathcal{M}[\mathsf{req}\ n\ a]_m \parallel b_n & \rightarrow & \mathcal{M}[\mathsf{\_syn}]_m \parallel (b \mathbin{\mathsf{>\!>}} a \mathbin{\mathsf{>\!>\!=}} \mathsf{\_rep}\ m)_n & \\
\textsc{Reply} & \mathcal{M}[\mathsf{\_syn}]_m \parallel \mathcal{M}'[\mathsf{\_rep}\ m\ a]_n & \rightarrow & \mathcal{M}[\mathsf{return}\ a]_m \parallel \mathcal{M}'[\mathsf{return}\ ()]_n &
\end{array}$$

Figure 2: Semantics of reaction

that the state component of process n is the same on both sides of a rule. Figure 2 shows the axioms of $\rightarrow$.

The first rule, EVAL, connects the semantics of reaction with the semantics of expression evaluation. This rule will in particular enable reduction of monadic expressions that are redexes according to the BIND rule above. However, BIND only allows a return expression on its left hand side, which is necessary to preserve the semantics of $\mapsto$. The remaining constants in the O monad must be dealt with directly by $\rightarrow$. We therefore define the notion of a *reaction context* $\mathcal{M}$ to single out the head of a sequence of >>= applications in an atomic process. Using $\mathcal{M}$, the SET and GET rules implement the standard semantics of a state monad.

Process creation is defined by NEW. Note that new processes are born in an inactive state. Axiom EGO captures the case where the sender and the receiver of an asynchronous message are identical, while the general case is handled by ACT. Since arriving code fragments are appended to the receiver irrespective of its current activities, our semantics actually specifies a *message buffer* for each object.

The final and most complex case is synchronous communication, which is defined in terms of two internal constants, _syn and _rep, that are not accessible to the programmer. REQ differs from ACT by the attachment of _rep to the message sent, and by putting the sender in a blocking state. Rule REPLY will subsequently be applicable to resolve this situation, provided that the receiver never loops indefinitely and never sends a req back to the waiting process. The impact of these preconditions, and their relevance to our claim that O'Haskell supports liveness by default is discussed in section 6.1.

Direct programmer access to _syn and _rep would be problematic for at least the following reasons: (1) both constants are hard to type in a way that is sound in general, and (2) unrestricted use of _syn would immediately destroy the reactive character of our language. Note also that req is not redundant; it cannot be faithfully simulated by asynchronous messages in two directions, since the original sender has no means of filtering out a reply before handling other messages that might be pending. What req actually offers is a very restricted, but convenient form of input, whose termination *does not depend on any external events that have yet to occur.* This stands in contrast to the *read* primitive of most traditional languages, which returns only at the will of a potential user.

We exemplify our semantic rules by showing how the desugared counter example of section 3.1 reduces. The program environment, which is supposed to initiate execution, is modelled as a process m with an empty state () and an initial monadic expression main. To reduce clutter, we only write out ν-binders where they are introduced, and skip trivial applications of rules EVAL and BIND. For the same reason, some eager evaluation is also implicitly performed.

$$\begin{array}{lr}
\nu m.\mathsf{main}_m & \\
\rightarrow \textsc{Eval} & (1) \\
(\mathsf{newCounter} \mathbin{\mathsf{>\!>\!=}} (\mathsf{inc,read}) \rightarrow \ldots)_m & \\
\rightarrow \textsc{Eval} & (2) \\
(\mathsf{new}\ 0\ \backslash\mathsf{self} \rightarrow (\mathsf{act\ self} \ldots, \mathsf{req\ self} \ldots) \mathbin{\mathsf{>\!>\!=}} \ldots)_m & \\
\rightarrow \textsc{New} & (3) \\
\nu n.(\mathsf{return}\ ((\backslash\mathsf{self} \rightarrow \ldots)\ n) \mathbin{\mathsf{>\!>\!=}} \ldots)_m \parallel (\mathsf{return}\ ())^0_n & \\
\rightarrow \textsc{Eval} & (4) \\
(\mathsf{act}\ n\ (\mathsf{get} \mathbin{\mathsf{>\!>\!=}} \ldots) \mathbin{\mathsf{>\!>}} \mathsf{req} \ldots)_m \parallel (\mathsf{return}\ ())^0_n & \\
\rightarrow \textsc{Act} & (5) \\
(\mathsf{req}\ n \ldots)_m \parallel (\mathsf{get} \mathbin{\mathsf{>\!>\!=}} \backslash\mathsf{val} \rightarrow \mathsf{set}\ ((\mathsf{val}+1)))^0_n & \\
\rightarrow \textsc{Get} & (6) \\
(\mathsf{req}\ n \ldots)_m \parallel (\mathsf{set}\ (0+1))^0_n & \\
\rightarrow \textsc{Set} & (7) \\
(\mathsf{req}\ n\ (\mathsf{get} \mathbin{\mathsf{>\!>\!=}} \ldots) \mathbin{\mathsf{>\!>\!=}} \ldots)_m \parallel (\mathsf{return}\ ())^1_n & \\
\rightarrow \textsc{Req} & (8) \\
(\mathsf{\_syn} \mathbin{\mathsf{>\!>\!=}} \ldots)_m \parallel (\mathsf{get} \mathbin{\mathsf{>\!>\!=}} \ldots \mathbin{\mathsf{>\!>\!=}} \mathsf{\_rep}\ m)^1_n & \\
\rightarrow \textsc{Get} & (9) \\
(\mathsf{\_syn} \mathbin{\mathsf{>\!>\!=}} \ldots)_m \parallel (\mathsf{return}\ 1 \mathbin{\mathsf{>\!>\!=}} \mathsf{\_rep}\ m)^1_n & \\
\rightarrow \textsc{Eval} & (10) \\
(\mathsf{\_syn} \mathbin{\mathsf{>\!>\!=}} \ldots)_m \parallel (\mathsf{\_rep}\ m\ 1)^1_n & \\
\rightarrow \textsc{Reply} & (11) \\
(\mathsf{return}\ 1 \mathbin{\mathsf{>\!>\!=}} \backslash\mathsf{one} \rightarrow \mathsf{return}\ ())_m \parallel (\mathsf{return}\ ())^1_n & \\
\rightarrow \textsc{Eval} & (12) \\
(\mathsf{return}\ ())_m \parallel (\mathsf{return}\ ())^1_n &
\end{array}$$

To illustrate the non-determinism inherent in the reaction relation we give an alternative reduction sequence for steps 7-8 above, which also shows buffering of a message to an active object.

$$\begin{array}{ll}
\quad\vdots & (6)\\
(\mathsf{req}\ n\ (\mathsf{get} \mathbin{>\!\!>=} \ldots)\ldots)_m \parallel (\mathsf{set}\ (0{+}1))^0_n & \\
\to \textsc{Req} & (7)\\
(\mathsf{\_syn} \mathbin{>\!\!>=} \ldots)_m \parallel (\mathsf{set}\ (0{+}1) \mathbin{>\!\!>} \mathsf{get} \mathbin{>\!\!>=} \ldots)^0_n & \\
\to \textsc{Set} & (8)\\
(\mathsf{\_syn} \mathbin{>\!\!>=} \ldots)_m \parallel (\mathsf{get} \mathbin{>\!\!>=} \ldots)^1_n & \\
\quad\vdots &
\end{array}$$

We have not developed any theory around our semantics so far, although this would certainly be an interesting research project. The similarities between our formulation and the presentation of Concurrent Haskell are so strong, however, that we expect the main results about the latter language to directly carry over to our case. This includes the important property that the deterministic, purely functional semantics of expression evaluation ($\mapsto$) is not affected by its inclusion into a non-deterministic, imperative context ($\to$).

4 Typing issues

Like monadic operators in general, actions, requests, and templates are first-class values in O'Haskell. This means, intuitively, that it should be possible to send a communication interface along from one object to another, to obtain dynamically evolving communication patterns. Unfortunately, the typings given to our primitive constants given in the previous section severely limit this possibility. The problem has to do with instantiating the state component of the O monad. Recall that for an action value this type stands for the internal state of the *sender*, so the problems we encounter are not an indication of insufficient object encapsulation. Still, the state component is necessary in general, since it ensures that all code fragments of an object read and assign to the same type of state. What is problematic is that all interesting action values in O'Haskell will be lambda-bound, and thus subject to monomorphic instantiation only. The net result is that communication interfaces can only be shared between objects that have the same internal state type!

There might be several ways out of this dilemma, including the first-class structures proposed by Mark Jones [Jon96], or the use of explicit, polymorphic coercion functions that get the static typings right but behave as the identity function at runtime. We have found, though, that the object-oriented concept of *subtyping* solves the problem very neatly, and we will present our program examples in the next section using the alternative typings that this solution gives rise to.

Our approach to subtyping in polymorphic languages is covered in detail in [Nor97]; for the purposes of this paper it suffices to know that subtyping relations between type constants must be explicitly declared by the programmer, via *polymorphic subtype axioms*. In this spirit, our concurrency primitives can be given more precise types if the following types and subtype axioms are provided as built-in:

```
Template t  <  O s t
    Action  <  O s ()
 Request t  <  O s t
```

The alternative typings for new, act, and req now become

```
new  ::  s → (Ref s → t) → Template t
act  ::  Ref s → O s () → Action
req  ::  Ref s → O s t → Request t
```

We argue informally that this refinement is sound, since none of the constants above reads or writes the local state of its executing object. Hence, the narrow types we have introduced may be safely promoted to the state monad O s, for any state s. What the subtyping axioms allow us to do is to choose this s anew *at each occurrence* of a particular constant.

Struct-like types still have their place in O'Haskell, though, because of their natural correspondence to the idea of a communication interface. They also fit quite nicely into our subtyping machinery, although we will not exploit this aspect here. Appendix D contains an extension of core Haskell with struct values that we will rely on in the next section. Taking both structures and subtypes into account, here is (finally) our preferred formulation of the by now well-known counter program:

```
struct Counter =
   inc    :: Action
   read   :: Request Int

newCounter :: Template Counter
newCounter =
   template
      val := 0
   with struct
      inc = action
         val := val + 1
      read = request
         return val

main = do
   c ← newCounter
   c.inc
   one ← c.read
   return ()
```

5 Examples

Classical examples used as a test-bed for new concurrency proposals are mainly concerned with *coordination problems*, e.g. how shared, mutable data can be protected from concurrent access, and *communication problems*, e.g. how buffers, channels, and the like can be built from available primitives. However, these problems often have less relevance to the class of message-based languages where O'Haskell belongs. So, let us from the outset emphasise that O'Haskell directly supports the following basic needs:

- **Critical sections.** A shared data structure is equivalent to an object in our model, and the actions and requests an object provides are mutually exclusive. Thus, the code fragments that constitute these services can be considered as automatically protected critical sections.
- **Message buffering.** Our semantics for message sending already provides unbounded buffering of messages.

This means that there is no need to implement separate queuing mechanisms when the ordering of messages sent must be preserved. We believe that the vast majority of buffer applications used in existing concurrent systems fits into this many-to-one communication pattern.

- **A signal system.** The use of signals to synchronise various events in a system is greatly simplified in our model, since actions have first-class status, and objects are only temporarily in a state where they cannot respond to any input.

Other, more specific communication and synchronisation requirements may of course occur in practice, but our experience with the language so far suggests that the programming style enforced by O'Haskell is surprisingly flexible, and that problems which initially seem to call for both blocking input commands and shared state variables often benefit from a reactive reformulation. Subsections 5.2 and 5.3 describe how some classical synchronisation facilities can be encoded in O'Haskell. We will, however, begin this section at the other end of the spectrum, by sketching a framework for a high-level application: an event-driven, interactive program with a graphical user interface.

5.1 Event-driven, interactive programs

Graphical user interfaces are standard on today's personal computers, and with them has come a style of programming that can be characterised as *event-driven*. In traditional languages, this style consists of structuring a program around an *event-loop*, which repeatedly does a blocking system-call to get the next event-structure, performs case-analysis on this structure, and then executes one of its branches depending on the actual event received. In the normal case, these branches never perform any blocking operations themselves; all input to the program is concentrated to the top of the event-loop.

Evidently, O'Haskell natively supports this programming style. From the view of the operating system, an application is nothing more than a process that responds to a certain set of events/messages, and the application, in turn, just sees the operating system as a process with another, specific communication interface. A (greatly simplified) specification of these interfaces might look like this:

```
struct GUIApp =
    redraw :: Action
    click  :: Point → Action
    drag   :: Point → Action
    key    :: Char → Action

struct Window =
    size   :: Request Point
    resize :: Point → Action
    clear  :: Action
    line   :: Point → Point → Action

someGUIApp :: Window → Template GUIApp
```

Here, someGUIApp can be a template for any kind of interactive program; the only thing an operating system (which is supposed to instantiate the template) needs to know is that the resulting process supports the GUIApp interface, and the template implementor, in turn, must only assume that there will be some Window interface available at runtime, where drawing commands can be sent. We like to call this division of responsibilities *communication by contract*.

So far the normal case. But what about exceptions to this scheme, for example when a graphical application needs to present a dialog box, or a popup menu, which has to consume all input until it closes? Is this not a situation where a blocking input abstraction would be helpful?

A reactive approach to this problem needs to make a clear distinction between opening a popup menu, say, and reacting on its completion. The former activity is naturally modelled as sending a message, while the latter part is equivalent to the *arrival* of a message. Since actions are first-class values, the code that creates a popup menu object could easily be parameterised with respect to the actions that handle selection. So, if we assume that the service of creating and opening a popup menu is available through some WindMgr interface, and that the click-handling code of our graphical application asks for this service, a reactive solution could follow this outline:

```
struct WindMgr =
    ...
    popup :: Point → [(String,Action)] → Action

...
    click pt = action
        windmgr.popup pt
            [ ("Foo",action A)
            , ("Bar",action B) ]
```

The important thing to notice here is that code fragments A and B above are only lexically within the scope of click — there is no "subroutine" relationship implied, and the execution of click will be over as soon as the popup message is sent. In due course of time, one of the actions at A or B will be executed, provided that the user chooses to select anything at all from the menu. A nice consequence of this reactive scheme is that our graphical application will still be able to respond to any message directed to it while the popup menu is open. This is really as it should be: there is no reason why, for example, redrawing a window should be postponed just because there is a special form of graphical input device open in front of it.

5.2 Semaphore encoding

A semaphore is a synchronisation device that in its simplest form is just a "token" that can be claimed and released, and where the claiming operation may block if the token is not available [Dij65]. According to our definition, a semaphore is clearly not a reactive abstraction, but as we have mentioned, O'Haskell provides both synchronisation and communication by more abstract means.

However, there might be situations where the implementation of synchronisation mechanisms is actually a part of the programming problem, for example in the simulation of a railway system. Hence it might be necessary to encode semaphores reactively, and we need to show how this can be done.

The key step towards a reactive implementation is to lift out the client code that is supposed to run after a successful claim, and put it into a separate action. Then the responsibility for triggering this action can be put on the semaphore, quite similar to the *continuation passing style* sometimes used in functional languages to encode stateful computations.

Here is the semaphore implementation:

```
struct Semaphore =
    claim   :: Action → Action
    release :: Action

semaphore :: Template Semaphore
semaphore =
    template
        active := False
        wakeup := []
    with struct
        claim grant = action
            if not active then
                active := True
                grant
            else
                wakeup := wakeup ++ [grant]
        release = action
            case wakeup of
                []      → active := False
                w:ws    → wakeup := ws; w
```

One can imagine many variations on this theme, e.g. letting claim be a boolean request that returns True in the successful case instead of triggering grant. This would allow a client to take special action if the claim is granted immediately.

5.3 Queue encoding

An alternative to the many-to-one buffering mechanism inherent in O'Haskell would be a many-to-many communication scheme, where data is communicated via a distinguished *queue* process. In order to implement such a queue in O'Haskell, however, we need to reconsider the *remove* operation, since it is supposed to block if no data is available. The reactive way of removing an item must be a two-phase operation: first a consumer *announces* its readiness to receive some data, then, perhaps later on when data has arrived, the queue process sends a message with the data to the consumer process. The code looks as follows:

```
struct Queue a =
    insert   :: a → Action
    announce :: (a → Action) → Action

queue :: Template (Queue a)
queue =
    template
        packets := []
        servers := []
    with struct
        insert p = action
            case servers of
                []      → packets := packets ++ [p]
                s:ss    → servers := ss; s p
        announce s = action
            case packets of
                []      → servers := servers ++ [s]
                p:ps    → packets := ps; s p

producer q =
    template
        ...
    with let produce = action
                x ← ... produce an x
                q.insert x
                produce
    in produce

consumer q =
    template
        ...
    with let consume x = action
                ... x ... consume x
                q.announce consume
    in q.announce consume

main = do
    q  ← queue
    p1 ← producer q
    p2 ← producer q
    c1 ← consumer q
    c2 ← consumer q
    p1; p2; c1; c2
```

Note that this "polarity switch" of the remove operation does not really clutter up the code; it is still as symmetrical as it would be in a language with blocking read operations. Turning the queue into a bounded buffer would mainly just require switching the polarity of the insert operation as well. The real benefit of this reactive encoding is that all processes involved can easily be extended to handle additional messages, without affecting the basic communication pattern.

5.4 Interrupt Service Routines

As an example of how the concept of a *hardware interrupt* fits into the reactive style, we give an implementation of a timer process, that allows its clients to "sleep" for a specified number of ticks. Sleeping should be interpreted reactively, however, meaning that a process passively awaits any message, one of which will signal that a certain amount of time has passed. This action must be supplied as a parameter to the timer each time a timing task is started. We do not specify the communication interface for the timer process as a struct value, since one of its actions, tick, is only meant to be installed in some interrupt vector table, and the other, start, should preferably be made available through some general, operating-system-like interface that we do not wish to consider any further.

```
newTimer =
    template
        time := 0
        pend := []
    with let
        start t sig = action
            pend := insert (time+t,sig) pend
    check_pending = do
        case pend of
            (t,sig):pend' | time >= t ->
                pend := pend'
                sig
                check_pending
            _ -> done

    tick = action
        time := time + 1
        check_pending

in (start, tick)
```

A notable feature of this example is the use of a recursive procedure in the interrupt service routine tick. Such a call is completely local to a process, and cannot be interspersed with other messages. Compare this with the recursive message sending that is performed by the producer processes of section 5.3.

It is interesting to see that safe communication between an interrupt service routine and an ordinary process (a rather tricky task in most programming languages) can be handled with the same mutual exclusion machinery that is used in ordinary interprocess communication. The reason behind this is that, in effect, *all* messages are modelled as interrupts in O'Haskell, and it does not matter whether some of them are actually generated by hardware.

5.5 A telnet client

We conclude this section with a slightly larger example: a rudimentary implementation of a Telnet client. The resulting code has a very appealing structure, centered around the intuitive fact that the state of a Telnet process is its current connection. We have not implemented any Telnet-specific handshaking, though, since that would just mean repeating the pattern used in the handling of open/close acknowledgements.

```
struct Connection =
    send       :: Packet -> Action
    close      :: Action
    peer       :: Request Host

struct Client =
    connected :: Connection -> Action
    deliver    :: Packet -> Action
    closed     :: Action

struct Tcp =
    open       :: Host -> Port -> Client -> Action
    listen     :: Port -> Client -> Action

struct Telnet =
    connect    :: Host -> Action
    keypress   :: Char -> Action
    disconnect :: Action
```

```
telnet :: Tcp -> Screen -> Template Telnet
telnet tcp screen =
    template
        you_server := Nothing
    with let me_client = struct
                connected c = action
                    you_server := Just c
                    screen.puts "[Connected]\n"
                deliver pkt = action
                    screen.putc (mkchar pkt)
                closed = action
                    you_server := Nothing
                    screen.puts "[Disconnected]\n"
    in struct
        connect host = action
            tcp.open host telnet_port me_client
        keypress ch = action
            case you_server of
                Just c     -> c.send (mkpkt ch)
                Nothing    -> screen.beep
        disconnect = action
            case you_server of
                Just c     -> c.close
                Nothing    -> done
```

The type definitions used in this example clearly demonstrate the principle of *communicating by contract*. Notice how the use of a connection is effectively prohibited until it is established (by splitting the interface to Tcp into two struct types). Note also how the Telnet process exhibits two different interfaces at the same time, depending on which level in the protocol hierarchy it is seen from. Finally, one interesting consequence of the reactive implementation is actually visible in the body of keypress, where the user is notified by a beep if characters are entered before a reliable connection has been established. This would not have been so straightforward in a language that had modelled tcp.open as a blocking input operation.

6 Discussion

6.1 Liveness

The decision to abolish blocking input commands is a radical one, but as we have seen, the program structure that emerges as a result has some interesting merits. As we have mentioned, it keeps programmers from imposing any order on the events that a program is set up to handle, and it makes the important liveness property hold by default.

The latter property needs some clarification. What we really would like to state is that every active object in an O'Haskell system is guaranteed to react to any message in a finite amount of time, but clearly there are some preconditions that must hold for such a statement to be true.

Firstly, an object that is stuck in a cycle of objects blocked on synchronous requests to each other will never react to any more messages. However, this is a detectable exception on the same level as division by zero, and one which is highly unlikely to occur in practice due to the relative sparseness of requests that we anticipate in real programs.

Secondly, an object which calls a non-terminating recursive procedure will obviously not be able to proceed. But this is an instance of the problem of guaranteeing termination in general, and we see no reason to treat the O type specially in this respect. It is worth noting here that the liveness of an O'Haskell system does not depend on any *non-terminating* properties that must be proved (c.f. [HPS96]).

With these observations in mind, we feel justified in claiming that an O'Haskell program upholds the liveness property *by default*, that is, liveness holds trivially for all processes unless the programmer constructively destroys it by writing inherently erroneous code. It is our belief that O'Haskell in this sense is less sensitive to programming mistakes, than a language where the liveness property crucially depends upon active cooperation from the programmer.

6.2 Preserving message ordering

Our choice to specify that messages should be queued is also worth some comments. Common practice in concurrent languages is to leave this issue unspecified, in order to facilitate distributed implementations across an unreliable network [PGF96, Agh86]. We base our decision on the following arguments:

- Many simple programs would be unduly complicated if message ordering was not preserved (c.f. the counter example in section 2).
- Processes on an unreliable network can be conveniently accessed via a library of local *proxy* objects. These proxies can present a reliable or an unreliable connection, at the free choice of the implementor. Implementing a proxy is greatly simplified, however, if communication with the local clients is order preserving.
- The only operation which cannot be faithfully simulated by a proxy is the synchronous request. Thus, a network object will need to have a more limited interface than a corresponding local one. This is a necessary restriction if we consider liveness to be important, and blocking input to be harmful. On the other hand, if the network in question is considered so reliable that a blocking input operation really would have done no harm, then the network can equally well be made a part of the language implementation, since it guarantees the language semantics.

7 Implementation

Since O'Haskell is defined as an extension to ordinary Haskell, it has turned out to be straightforward to build a compiler for O'Haskell by performing just the syntactic transformations in figure 1, and providing the seven primitive constants of the O monad in a library module. We actually have two implementations of this module at the moment; one being written in pure Haskell using a datatype with constructors for each primitive. This has allowed us to program the reaction semantics in figure 2 by pattern matching, but has resulted in a rather inefficient implementation. In addition, not all semantic rules can be typed in pure Haskell, which we have circumvented using explicit type casts.

It turns out that a more pragmatic implementation, that also uses concurrent evaluation, can actually be built from the primitives provided by Concurrent Haskell. These primitives are available in Lennart Augustsson's eminent 2nd generation Haskell compiler (HBCC) [Aug96], which also provides "real", first class structures à la Mark Jones, to our delight. This implementation is short and is included below, as it might help the Concurrent Haskellate reader in understanding certain aspects of O'Haskell's reactive semantics.

To make a complete comparison between the two languages, we also provide a translation from Concurrent Haskell into O'Haskell, which can be found in appendix E.

7.1 O'Haskell in Concurrent Haskell

The type of objects is built on top of the IO monad, and supplies commands with a mutable variable that holds the local state.

```
type O s t      = IOVar s -> IO t
```

The type O is recognised as the standard reader monad.[4]

```
returnO a       = \_ -> return a
m `bindO` f     = \v -> do a <- m v
                           f a v
```

The state-related operations boils down to accessing and manipulating the mutable variable supplied to the commands.

```
get             = \v -> readIOVar v
set s           = \v -> writeIOVar v s
```

The type of object references encapsulates a message queue (called *channel* in [PGF96]) of commands to be executed by the object.

```
type Ref s      = Chan (O s ())
```

Sending an asynchronous action to an object implies writing the command to this object's channel.

```
act r c         = \_ -> putChan r c
```

The synchronous request is a little more involved, here we create a temporary MVar to mediate the answer.

```
req r c         = \_ -> do ans <- newMVar
                           putChan r
                             (\v -> do a <- c v
                                       putMVar ans a)
                           takeMVar ans
```

Each object has an associated server thread which forever reads commands from the object channel and executes them.

```
objproc         :: IOVar s -> Chan (O s ()) -> IO ()
objproc v r     = do c <- getChan r
                     c v
                     objproc v r
```

The last primitive to define is new, which creates a fresh mutable variable for the state, and a new command queue. Finally, new forks off a server thread for the new object.

```
new s iface     = \_ -> do v <- newIOVar s
                           r <- newChan
                           forkIO (objproc v r)
                           return (iface r)
```

[4]To avoid any confusion regarding overloading, we let O'Haskell's basic monad operations be denoted by returnO and bindO in this subsection.

8 Related work

Much work has been done in extending functional languages with concurrency features. Within the lazy functional community, *stream-based* approaches have a fairly long tradition [Sto86, Tur87, Hen82, HC95]. A common characteristic of these solutions is that communication is directed towards a particular *process*, and that all input streams for a process are merged into one before reception. This means that stream-based processes do not constrain the order in which different messages are received, a feature quite similar to the reactive property of our proposal. The drawback of the stream-based school is that it requires a rather heavy use of disjoint sum types in the merging of messages, and that the coupling between output and input is very loose, thus ruling out synchronous constructions like our requests.

The connection between a stream processor and an O'Haskell object can be illustrated by considering the type SP (Either a b) (Either c d) (taken from the *Fudgets* library for concurrent programming in Haskell [HC95]). This type stands for a stream processor which reacts to messages of type a or b by outputting messages of type c or d. Such a stream processor is naturally modelled in O'Haskell as a template parameterised over a pair of output actions, presenting an interface with the input actions:

$$\begin{array}{l}(c \rightarrow \text{Action}, d \rightarrow \text{Action}) \rightarrow \\ \text{Template}\ (a \rightarrow \text{Action}, b \rightarrow \text{Action})\end{array}$$

A different approach is to separate the concept of a process and a communication destination. This is the common view in most process calculi, and it has been adapted in many functional languages as well [PGF96, Sch95, Rep92, Car86, Hol83]. An immediate benefit of these systems is that most message typing problems disappear, even for very complex communication patterns. However, since a *channel* (as we might collectively call the passive communication points) must be addressed explicitly in both the send and receive operations, the ability to simultaneously wait for any kind of message is lost. One remedy is to introduce the *choice* operator known from process calculi [Mil91], but its complexity generally makes it hard to implement efficiently, and great care must also be taken to avoid loss of abstraction when it is used [Rep92]. Choice-free encodings of object-like structures using multiple threads and locks are described in [PT94], while Concurrent Haskell seems to assume tagging by means of a datatype in its encoding of iterated choice [PGF96]. What O'Haskell offers is direct support for reactive reception of synchronous and asynchronous messages of multiple types, without the need for multiplexing by tagging, or coordination by additional processes.

Our unified view of objects as processes stems from the Actor model [Agh86]. Like objects in this model, an O'Haskell object can basically do three things in reaction to a message: (1) send messages, (2) create objects, and (3) update its local state. In the Actor model, these activities all occur in parallel, and reaction is thus atomic. We allow a *sequence* of operations in each action body, partly because this blends better with the monadic framework, and partly because we consider sequential imperative programming to be quite natural and something we wish to support. Furthermore, while state change in the pure Actor model is equivalent to specifying a replacement behaviour, we rely exclusively on state variables, which has the desired effect of making the state of an object decoupled from the set of messages it is willing to receive.

Erlang and UFO are two functional languages that are also influenced by the Actor model [AVWW96, Sar93]. Erlang endorses a sequential view of processes like we do, but relies on tail calls to keep a process alive, and utilises an untyped, Prolog-like pattern-matching mechanism to specify a current set of accepted messages. UFO is typed and has encapsulated state variables, but the object/process correspondence is vague and sequencing comes from data dependence only. Message acceptance can furthermore be restricted by manipulating predefined pseudo-variables. Neither of these languages can be called reactive in our meaning of the word.

An interesting characterisation of concurrency proposals for functional languages is of course whether they are able to preserve a pure semantics of expression evaluation. Of the works cited above, [HC95, Tur87, Sto86, PGF96, Sch95, Hol83] belong to this category, while [Hen82, Rep92, Car86, AVWW96, Sar93] do not. As we have mentioned before, retaining an unobstructed evaluation semantics has been one of the primary goals in the design of O'Haskell.

There is a host of concurrent object-oriented languages that might relate to O'Haskell in the sense that they support both concurrency and some typical object-oriented features (see e.g. [AYW93]). We are not yet ready to comment on these in any detail; however, it seems to us that many concurrent object-oriented designs are mostly concerned with maximising parallelism, in order to achieve good performance on massive multiprocessors [Pap92]. This might be the reason why it is hard to find any well-developed notion of functions in these proposals. Like the designers of Concurrent Haskell, we take the standpoint that performance-enhancing parallelism should be *semantically transparent*, a task for which pure functional languages are ideally suited [Ham94].

Objects in O'Haskell bear some resemblance to the *monitor* concept developed by Hoare [Hoa74]. Monitors were introduced to automatise the typical pattern where a semaphore is always released by the same process that has claimed it, as a means of protecting a critical section. The other distinct use of a semaphore, as identified by Dijkstra in [Dij68], feeds input to a particular process by means of a counting semaphore which is exclusively claimed by the receiver. This latter requirement is handled in a monitor by explicitly managed *condition variables*, something which significantly complicates the monitor semantics. We might say that our objects generalise the ordinary use of a monitor to cater for the input-feeding needs as well.

Reactive is a term coined by Pnueli and further used by Berry and Benveniste to describe systems that maintain an interaction with their environments [Pnu86, BB91]. The word is also sometimes seen as a synonym for the *synchronous* approach to *deterministic* reactive programming advocated by the latter authors. Although execution of an O'Haskell program is neither deterministic nor synchronised by a global clock, we think that our somewhat narrow definition of the term is still appropriate. The reason for this is that, in common with the synchronous school, an O'Haskell system can always be considered to react "immediately" just by utilising a sufficiently fast processor.

The work reported in this paper is an extension and reformulation of previous work by the first author [Nor95].

9 Conclusion and further work

In this paper we have defined an extension to Haskell that supports semantically visible concurrency by means of *reactive objects*. The core of our approach is a deviation from the common view of a process as a long thread of control interspersed with active resting points, towards the more object-oriented notion of a process as a collection of unordered, terminating code fragments connected by a common mutable state. Our communication primitives notably lack any blocking input facilities — a central feature of our reactive model is that all input to a process is confined to the passive, resting state that every object returns to when it is not executing.

We have developed a prototype implementation, which has enabled us to run several quite interesting example programs. Our experience with the language and the programming style it supports is encouraging, still the obvious path for further work starts with the development of more realistic applications that will tell us whether this reactive style, that looks very promising in the small, scales up.

Another branch of work concerns the language implementation. We would like to integrate the subtyping approach in [Nor97] with the preprocessor, and do a direct implementation of the library primitives in terms of the underlying runtime machinery.

Even though we believe that most O'Haskell programs by default uphold the reactive property, it might be interesting to investigate possibilities to formally guarantee this (and other properties), for example by using a more informative type system.

Acknowledgements

Thomas Hallgren and Björn von Sydow.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Aug96] Lennart Augustsson. The Haskell Compiler HBCC. Personal communication, augustss@cs.chalmers.se, 1996.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 1996.

[AYW93] G. Agha, A. Yonezawa, and P. Wegner, editors. *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *ACM Principles of Programming Languages*, pages 81–94, San Francisco, CA, January 1990.

[BB91] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. Technical Report 1445, INRIA-Rennes, 1991.

[Car86] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, LNCS 242. Springer Verlag, 1986.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, New York, 1965. Academic Press.

[Dij68] E.W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, 1968.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction (Invited Paper). In *PASCO '94*, Linz, Austria, September 1994.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In *Advanced Functional Programming*, LNCS 925, pages 137–182. Springer Verlag, 1995.

[Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications*, pages 177–189. Cambridge University Press, 1982.

[Hoa74] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hol83] Sören Holmström. PFL: A Parallel Functional Language and Its Implementation. Technical Report PMG-7, Programming Methodology Group, Chalmers University of Technology, 1983.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[Jon96] M.P. Jones. Using Parameterized Signatures to Express Modular Structure. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[LJ94] J. Launchbury and S.L. Peyton Jones. Lazy Functional State Threads. In *ACM Programming Languages Design and Implementation*, Orlando, FL, 1994. ACM Press.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Lab for Foundations of Computer Science, Edingburgh, October 1991.

[Nor95] Johan Nordlander. Lazy Computations in an Object-oriented Language for Reactive Programming. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages*, San Fransisco, CA, January 1995. Available as Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana-Champaign.

[Nor97] Johan Nordlander. Pragmatic Subtyping in Polymorphic Languages. Forthcoming, 1997.

[Pap92] M. Papathomas. A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages. In *ECOOP '91 Workshop on Object-Based Computing*, LNCS 612. Springer Verlag, 1992.

[PGF96] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295–308, St Petersburg, FL, January 1996. ACM Press.

[PH96] J. Peterson and K. Hammond, editors. *The Haskell 1.3 Report.* YALEU/DCS/RR-1106. Yale University Research Report, 1996.

[Pnu86] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In *Current Trends in Concurrency*, LNCS 224. Springer Verlag, 1986.

[PT94] B. Pierce and D.N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming*, Sendai, Japan, November 1994.

[Rep92] J. Reppy. Concurrent ML: Design, Application and Semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, LNCS 693. Springer Verlag, 1992.

[Rey94] J. Reynolds. User Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, Cambridge, MA, 1994. MIT Press.

[Sar93] J. Sargeant. Uniting Functional and Object-Oriented Programming (invited paper). In *Object Technologies for Advanced Software*, LNCS 742, Kanazawa, Japan, November 1993.

[Sch95] E. Scholz. Four Concurrency Primitives for Haskell. In *Proc. Haskell Workshop*, pages 1–12, La Jolla, CA, 1995. Available as Yale University Research Report YALEU/DCS/RR-1075.

[Sto86] W. Stoye. Message-based Functional Operating Systems. *Science of Computer Programming*, 6:291–311, 1986.

[Tur87] D. Turner. Functional Programming and Communicating Processes. In *Conference on Parallel Languages and Architectures*, LNCS 259, 1987.

A An operational semantics for core Haskell

$$
\begin{array}{lll}
a, b & ::= & x \mid \backslash x \to a \mid a\ b \mid \underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ b \mid \\
 & & k \mid \underline{\mathsf{case}}\ a\ \underline{\mathsf{of}}\ \overline{k \to b}
\end{array}
$$

$$
\mathcal{E} \quad ::= \quad [\,] \mid \mathcal{E}\ b \mid \underline{\mathsf{case}}\ \mathcal{E}\ \underline{\mathsf{of}}\ \overline{k \to b}
$$

$$
\begin{array}{lrcll}
\textsc{Apply} & (\backslash x \to a)\ b & \mapsto & a[b/x] & \\
\textsc{Case} & \underline{\mathsf{case}}\ k_j\ \overline{a}\ \underline{\mathsf{of}}\ \overline{k_i \to b_i} & \mapsto & b_j\ \overline{a} & \\
\textsc{Let} & \underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ b & \mapsto & b[(\underline{\mathsf{let}}\ x = a\ \underline{\mathsf{in}}\ a)/x] & \\
\textsc{Cont} & \mathcal{E}[a] & \mapsto & \mathcal{E}[b] & \text{if } a \mapsto b
\end{array}
$$

B Structural congruence

$$
\begin{array}{rcll}
P \parallel Q & \equiv & Q \parallel P & \\
P \parallel (P' \parallel Q) & \equiv & (P \parallel P') \parallel Q & \\
P & \equiv & Q & \text{if } P =_\alpha Q \\
\nu n.\nu m.P & \equiv & \nu m.\nu n.P & \\
\nu n.(P \parallel Q) & \equiv & P \parallel \nu n.Q & \text{if } n \notin \textsc{fn}(P)
\end{array}
$$

C Structural reaction

$$
\textsc{Par} \quad \frac{P \to P'}{P \parallel Q \to P' \parallel Q}
$$

$$
\textsc{Res} \quad \frac{P \to P'}{\nu n.P \to \nu n.P'}
$$

$$
\textsc{Equiv} \quad \frac{P \equiv Q \qquad Q \to Q' \qquad Q' \equiv P'}{P \to P'}
$$

D Extending core Haskell with structures

$$
\begin{array}{lll}
a, b & ::= & \ldots \mid \underline{\mathsf{struct}}\ \overline{l = a} \mid a.l \\
\mathcal{E} & ::= & \ldots \mid \mathcal{E}.l
\end{array}
$$

$$
\textsc{Select} \quad (\underline{\mathsf{struct}}\ \overline{l_i = a_i}).l_j \quad \mapsto \quad a_j
$$

E Concurrent Haskell in O'Haskell

To complement the encoding of O'Haskell given in section 7, we also provide a translation in the other direction: Concurrent Haskell's primitives implemented in our language. We consider the following IO operations:

```
returnIO    :: a -> IO a
bindIO      :: IO a -> (a -> IO b) -> IO b
forkIO      :: IO () -> IO ()
newMVar     :: IO (MVar a)
putMVar     :: MVar a -> a -> IO ()
takeMVar    :: MVar a -> IO a
```

The IO monad in implemented as a composition of a continuation monad and a reader monad, reflecting the fact that *any* expression of type IO *t* might involve indefinite blocking, so we must be able to produce action values for the currently executing process on the fly.

```
type IO a      = Ref () -> (a -> O () ()) -> O () ()

returnIO a     = \_ c -> c a

m `bindIO` f   = \s c -> m s (\a -> f a s c)
```

Forking off a process is interpreted as the creation of a stateless object exporting a single action-valued interface, which is immediately triggered. Note the use of the implicitly bound variable self.

```
forkIO p       = \_ c -> do o <- template
                             with action
                               p self return
                        o
                        c ()
```

An MVar becomes a special kind of object, with the following interface:

```
struct MVar a =
    put        :: a → Request ()
    take       :: (a → Action) → Action
```

The intention is that put updates the state of its MVar with a new item, while take announces the readiness of some process to consume an eventual item stored inside the MVar (c.f. section 5.3). A Request, rather than an Action, is necessary in the type of put in order to mimic the error-handling semantics of Concurrent Haskell.

Assuming there is a template mvartempl for creating MVar objects, the implementation of newMVar and putMVar is straightforward:

```
newMVar        = \_ c → do v ← mvartempl
                           c v

putMVar v a    = \_ c → do v.put a
                           c ()
```

Since executing takeMVar marks the end of the currently executing process fragment, the translation does not call the given continuation, but wraps it up in an action value using the Ref identity of the currently executing process. This action is sent to the MVar in question, and the calling process enters a resting state.

```
takeMVar v     = \s c → v.take (\a → act s (c a))
```

Finally we give the code for mvartempl. It is a variant of the queue encoding in section 5.3 that limits the number of stored items to at most one. Types for the state variables are included as a reading aid, but we have to put these inside comments since we do not (yet) support scoped type variables.

```
mvartempl :: Template (MVar a)
mvartempl =
    template
        val := Nothing      -- Maybe a
        takers := []        -- [a → Action]
    with struct
        put a = request
            case takers of
                []   → case val of
                           Nothing   → val := Just a
                           Just _    → error "putMVar"
                t:ts→ t a
                           takers := ts
        take t = action
            case val of
                Nothing    → takers := takers ++ [t]
                Just a     → t a
```

Judging from the sheer amount of code required, the translation of Concurrent Haskell into O'Haskell looks slightly more complex than the converse encoding. It should be borne in mind, though, that the interpretation in section 7 also involves an implementation of the abstract data type Chan, which roughly corresponds to the encoding of MVars above. We therefore conclude that neither language can be coined more 'expressive' or 'basic' than the other.

Debugging Reactive Systems in Haskell

Amr Sabry
Department of Computer Science
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Jan Sparud
Department of Computer Science
University of York
Heslington, York Y01 5DD, UK
sparud@cs.york.ac.uk

Abstract

We support the use of Haskell in two classes of applications.

First, we note that although Haskell is well-suited for stream-oriented applications, like reactive systems, most implementations lack tools for locating subtle errors like deadlocks. To support the use of Haskell in such applications, we have designed and implemented a debugger with an intuitive graphical interface. The debugger simplifies the task of locating deadlocks by providing users with a mostly declarative view of programs as systems of recursive equations over streams.

Second, we investigate the use of Haskell in the implementation of the debugger. As expected, this effort requires a variant of Haskell, currently used in the intermediate phases of Haskell compilers, that combines call-by-need semantics with computational effects. To support robust programming in this system-oriented variant, we formalize the semantics of handleable exceptions in a call-by-need language.

1 Streams and Deadlocks

As Kahn showed in 1974 [23], any deterministic network of processes can be modeled using recursive equations over streams. Such networks occur in telephones, missiles, and many other embedded *reactive* programs. The stream model corresponds in spirit to the electrical engineer's concept of a signal-processing system [1] and thus provides a natural formalism for the design, specification, and verification of these networks [11]. Additionally, stream-based specifications are directly executable in a variety of high-level languages. Examples of languages that elegantly support computations over streams include modern lazy functional languages such as Haskell [19], hardware-description languages such as MHDL [7] (an extension of Haskell), data-flow programming languages such as Lucid [4, 5], and synchronous programming languages such as Esterel [9, 10], Lustre [12], and Signal [8].

These languages have been remarkably successful in supporting the rapid development of reliable systems. First, streams (lazy lists) are the fundamental and most widely supported data structure in most of these languages. Second synchronization is dealt with automatically: programmers are only required to specify sets of recursive functions over streams without having to worry about many tedious and error-prone details. Indeed Caspi and Pouzet [13] report that some of the programs running on the Airbus A320 have been obtained in this way. Also Broy *et al.* [11] use this methodology for the development of various distributed systems.

Despite its appeal, programming with streams is quite subtle. A common mistake consists of writing a system of recursive equations over streams whose fixed point is semantically undefined [32]. Operationally this error may manifest itself in terms of an obscure error message from the compiler or an equally obscure error message from the runtime system. For example, when suspecting a deadlock, our liveness analysis [22] returns a large (and incomprehensible) list of unsatisfied constraints. As another example, typical Haskell implementations, *e.g.*, the 'hbc' compiler from Chalmers University, detect and report deadlocks with the message "`Runtime error: Black hole detected in eval zap`," which gives about as much information as the infamous C error message "`Segmentation fault`."

In fact, one can usually locate the sources of segmentation faults using a C debugger but this is not even the case for black holes: users get little debugging support from typical Haskell implementations. Furthermore, conventional debugging techniques for imperative languages such as setting breakpoints, single-stepping, tracing, and watching variables are not particularly suited for locating deadlocks occurring in lazy functional languages [15, 16, 29, 33] or synchronous languages [14]. Indeed our own personal experience in specifying reactive systems in Haskell quickly convinced us of the need for a special-purpose debugging technology specifically tailored towards computations over streams.

This paper reports on the design and implementation of such a debugger in the context of the Haskell compiler 'hbc.' The debugger has an intuitive graphical user interface, and can be used to quickly pinpoint the sources of many deadlocks. Experience with the implementation indicates that the system works remarkably well for small but realistic programs. The debugger nicely complements our formal liveness analysis [22] providing users with a useful set of tools for developing reactive programs.

Like many system-level Haskell programs, the implementation of the debugger relies on expressions with side-effects. Since Haskell bans such expressions, programmers that need this functionality either write low-level routines in C or in an *ad hoc* variant of Haskell that integrates computational effects with the call-by-need semantics. Recently Launchbury and Peyton Jones [24] convincingly argued for the second approach:

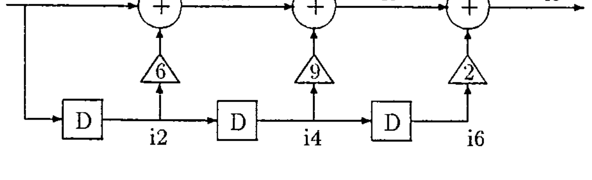
> "Should we outlaw `interleaveST` on the grounds that it is insufficiently well behaved? Not necessarily. Outlawing `interleaveST` would simply drive its functionality underground rather than prevent it happening. For example, we want to have lazy file reading. If it cannot be implemented in Haskell then it will have to be implemented "underground" as a primitive operation written in C or machine code. The same goes for unique-supply trees and lazy arrays.
>
> *Implementing such operations in C does not make them more hygienic or easy to reason about.* On the contrary, it is much easier to understand, modify and construct variants of them if they are implemented in a Haskell library module than if they are embedded in the runtime system"[24, p.45].

We therefore study a variant of Haskell with call-by-need semantics and handleable exceptions. This variant of Haskell is reminiscent of the informal Haskell dialect used by some programmers, at their own risk, to implement generation of new names [6], memoization [20], non-deterministic choice [21] and other paradigms that are *not* expressible in a modular way in a purely functional language. It is also reminiscent of the intermediate language of Haskell compilers like the Glasgow Haskell compiler, that implement monadic state by updates on a global shared store.

The next section motivates the use of streams in programming reactive systems. Section 3 discusses conventional debugging techniques and explains why they are unsuitable for our purpose. Section 4 presents a realistic example that uses Haskell to write the controller for a small industrial production cell. The example deadlocks due to a subtle bug that we unintentionally introduced in our first attempt at writing the code. We illustrate the user's interface of the debugger and show how a simple debugging session fixes the problem. Section 5 provides the details of the implementation of the debugger, and reveals that the implementation requires handleable exceptions. Section 6 presents the semantics of such exceptions in a call-by-need language. Finally Section 7 puts our work in perspective by explaining its weaknesses and charting ideas for future development.

2 Programming Reactive Systems

Reactive systems are programs that must *continuously* react to external stimuli from their surrounding environment [17]. For example, a digital filter is usually specified as a weighted sum of the n last samples of the input signal. The following diagram depicts such a filter:



The block D stands for a delay element. The triangles represent elements that scale every value in the stream by the corresponding number.

It is relatively straightforward to transliterate the filter to working Haskell code. The first step consists in implementing the intended semantics of each block:

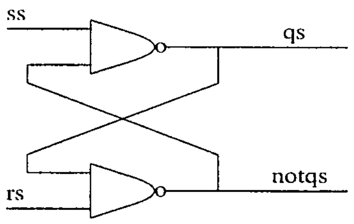
```
sum (x : xs) (y : ys) = (x + y) : sum xs ys
delay x = tail x
weight w = map (w*)
```

Having specified the blocks, it remains to write a set of mutually recursive equations over streams that reflects the interconnections of the network:

```
filter i0 = i5
 where i1 = sum i0 (weight 6 i2)
       i2 = delay i0
       i3 = sum i1 (weight 9 i4)
       i4 = delay i2
       i5 = sum i3 (weight 2 i6)
       i6 = delay i4
```

Note that the generation of this last piece of code can be easily automated given the diagram: it only depends on the interconnections between the components, not on their semantics.

The above development process is simple, natural, and elegant. But one must not get carried away; there are some limitations! Consider the following diagram specifying a basic flip-flop circuit with nand gates and the program produced by naïvely following our development process:



```
nand (1 : xs) (1 : ys) = 0 : nand xs ys
nand (_ : xs) (_ : ys) = 1 : nand xs ys

ff rs ss = (qs, notqs)
  where qs = nand ss notqs
        notqs = nand rs qs
```

Although the physical flip-flop device works, our program doesn't. And in fact pure Haskell appears ill-suited for such asynchronous circuits.

However, despite its limitations the stream model is suitable for many applications and is at the core of more expressive formalisms that accommodate asynchronous events, real-time, non-determinism, etc.

3 Related Work

To understand the contribution of our work, it is necessary to review the available debugging tools for Haskell in general and stream computations in particular. But first it is worth noting that most implementations of synchronous languages include debuggers and simulation environments in which users can experiment with the code. The widespread use of such tools reinforces our claim that they are essential in a programming environment that supports computations over streams.

The lazy (demand-driven) evaluation of functional languages schedules computations depending on the demands for their results. This evaluation mechanism is hard to predict and to relate to the source code of the program, and hence is not well suited for conventional debuggers that operate at a low level of abstraction that is closer to the machine code than to the source program. This led researchers to propose *declarative debugging* as a viable strategy for debugging lazy functional programs [27, 28, 29].

One promising declarative debugging technique is *algorithmic debugging* [30, 33, 34]. Algorithmic debugging is a two phase process. First, the program is run and an execution dependence tree is built. Then, this tree is interactively traversed. Each node in the tree corresponds to a function call and its result. While traversing the tree (in a top-down manner), the system asks, for each node, if the result of the corresponding function call is correct or not. Based on the answers the programmer gives, the system chooses which function call to show next. A bug has been found when the result of a function call is erroneous and all children nodes are correct. The method requires that the functions have no side effects, which makes it suitable for pure functional languages.

The ability to view the value of any expression in a program execution, and the expressions on which that value depends, is generally useful for debugging many lazy functional programs. Unfortunately, it is not of much help when debugging stream computations. To understand a computation that deadlocks while manipulating several streams, it is crucial to:

- understand the sequence of *events* that led to the deadlock, and
- have a view of *all* the relevant streams at the point in which the deadlock occurred.

Algorithmic debugging abstracts away from time and does not faithfully report the evaluation steps in the order in which they actually happened. Furthermore, users of an algorithmic debugging tool never have a global view of the computation in which they can keep track of several streams at once. These properties make algorithmic debugging unsuitable for locating deadlocks in stream computations.

4 Controlling An Elevating Table

We present a simplified version of the actual program fragment that spurred the idea of the new debugger.

4.1 The Program

Figure 1 shows a small fragment of a case-study production cell [26]. This fragment is concerned with the implementation of the controller of a simple elevating table and testing it with a simulated hardware environment. The table is initially in a low position. When the controller detects an input blank, it issues an Up command to the motor interface. This abstract command is implemented by the interface using a low-level protocol that reads sensors and emits control signals (actuators) to a motor. The block Env simulates the actual environment. Assuming the environment operates correctly, the controller eventually receives an acknowledgment, delivers the blank to the next device (a robot arm), and issues a Down command to lower the table to its initial position. The Down command is implemented by the interface using a protocol similar to the one implementing Up; when the Down command is acknowledged the controller is ready to accept more input blanks.

We will give the precise implementation of the protocols implementing the commands Up and Down. First we formally describe the sets of messages that are required by the specification of the table [26]:

```
data MotorCmd  = Up | Down
data Sensor = Low Bool | High Bool
type SensorVector = (Sensor,Sensor)
data Actuator = MotorUp | MotorDown
              | MotorStop | GetStatus
```

The specification of the system requires the existence of two sensors that identify the current location of the table. For example, when the table is in the low position, the sensors are (Low True, High False). The controller can only access the current values of the sensors by issuing a GetStatus control signal; changes to the sensors between GetStatus signals are undetected by the controller. Finally, the motor that moves the table can be controlled using three self-explanatory control signals: MotorUp, MotorDown, and MotorStop.

The above description yields the following code for an idealized environment:

```
env acs = loop acs (Low True, High False) where
 loop (ac:acs) currentSensors =
   case ac of
    GetStatus  -> currentSensors :
                  loop acs currentSensors
    MotorUp    -> loop acs (Low False, High True)
    MotorDown  -> loop acs (Low True, High False)
    MotorStop  -> loop acs currentSensors
```

The environment code assumes that the table is initially in the low position and repeatedly looks at the current control signal in the actuators stream. If the signal is GetStatus, then the current values of the sensors are returned. Otherwise the appropriate action is performed, possibly modifying the current values of the sensors.

It remains to perform the calculation of the actuators stream in the module Interface:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
          MotorUp : GetStatus : goU mcs ss

        goD mcs ((Low True, High _) : ss) =
          MotorStop : emit mcs ss
        goD mcs ((Low _, High _) : ss) =
          MotorDown : GetStatus : goD mcs ss
```

First the code issues a GetStatus signal to get the initial values of the sensors. Then we start a loop that examines the current abstract command and the current values of the sensors and emits the appropriate actuators. In a realistic environment, the actuators would not be immediately acknowledged and the controller might have to loop several

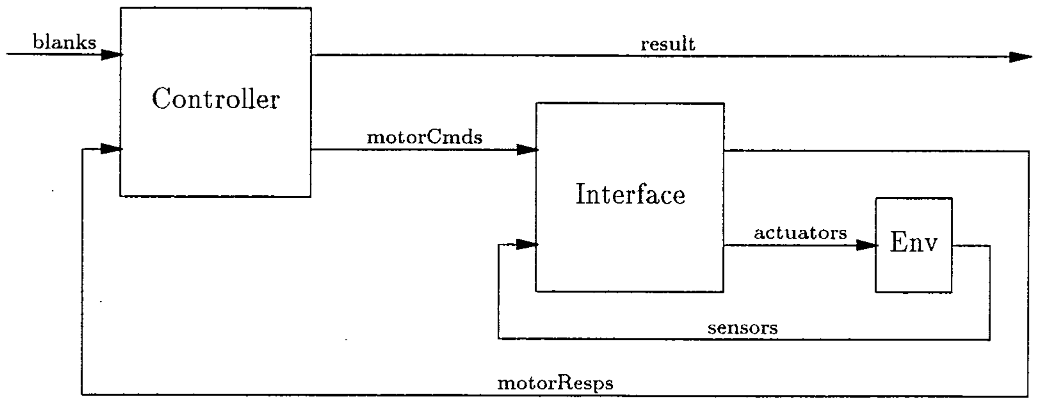


Figure 1. Elevating Table

times until the sensors confirm that the actuator signal has been acknowledged.

The main program reflects the interconnections in Figure 1. To test the program fragment independently, let's assume that the stream of motorCmds consists of an infinite sequence of alternating Up and Down commands:

```
motorCmds = Up : Down : motorCmds
actuators = interface motorCmds sensors
sensors = env actuators
```

Note how the definitions of sensors and actuators are mutually recursive.

The current code is overly simplified because it ignores the remainder of the production cell, and even ignores some aspects of the elevating table like the fact that it should rotate left and right while moving up and down. Nevertheless, the code captures the essence of our initial implementation of the elevating table including the fact that it contains a subtle error causing the system to deadlock! Because of the simplifications, the bug may be obvious at this time, but it certainly wasn't obvious in the complete program (around 600 lines of Haskell).

4.2 Finding the Deadlock

Our debugger operates at a high-level of abstraction that reflects the source program itself. Instead of getting lost in details about lazy evaluation, our debugging technique maintains the illusion of a mostly declarative computation expressed as a fixed point of recursive equations over streams.

We start by illustrating the user's point of view by going through a possible debugging session for the elevating table code. To find the source of the deadlock the user must identify the relevant streams of interest. This information is used by the debugger to keep track of, and display the sequences of values that belong to these streams. In our case, we only have three streams and we state that we are interested in all of them.

When running the program under the control of the debugger, the window in Figure 2 appears (initially empty). The user then can interact with the computation by clicking on the buttons Next. Each click returns the next value on the corresponding stream as well as all values on the other streams on which it depends. In our example, the first click on the stream actuators returned the value GetStatus without demanding any other values on the other streams. A second click on the same stream returned the values MotorUp and GetStatus. These two values can only be produced if a command Up is issued on the stream motorCmds and the sensors indicate that the table is in the low position. After the next click the computation has proceeded to the point where the table is in the high position and the motor is ordered to stop. When attempting to process the Down command, the next click displays the symbol $\perp$ in the columns for the streams of sensors and actuators indicating a cyclic dependency between the next two values, *i.e.*, a deadlock.

Even before examining the code for bugs, we can immediately note one important fact: each GetStatus signal on the stream of actuators produces the current values of the sensors on the next click. The first two GetStatus signals have been processed and acknowledged and no values can appear on the stream of sensors unless a new GetStatus signal is issued. It is also clear that the deadlock occurs at a point in the neighborhood of the code that issues the MotorStop signal and the error manifests itself immediately after consuming the command Down. Looking back at the relevant piece of code we find:

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

(*)     goU mcs ((Low _, High True) : ss) =
          MotorStop : emit mcs ss
        goU mcs ((Low _, High _) : ss) =
          MotorUp : GetStatus : goU mcs ss

        goD mcs ((Low True, High _) : ss) =
          MotorStop : emit mcs ss
        goD mcs ((Low _, High _) : ss) =
          MotorDown : GetStatus : goD mcs ss
```

In the line marked (*) we see that after issuing the signal MotorStop, we proceed to the emit loop which consumes a Down command, and then attempts to access the stream of sensors without any intervening GetStatus signals. Thus one way to fix the code would be to simply add a GetStatus signal after stopping the motor (in both lines that stop the motor):

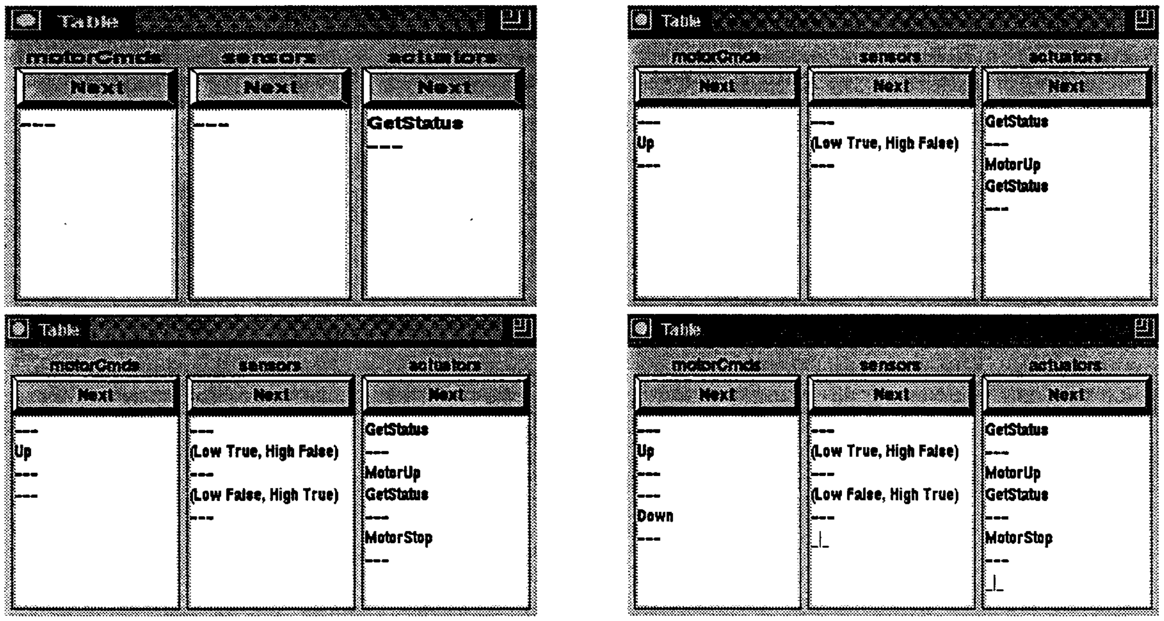


Figure 2. A Sample Debugging Session

```
actuators = GetStatus : emit motorCmds sensors
  where emit (Up:mcs) ss = goU mcs ss
        emit (Down:mcs) ss = goD mcs ss

        goU mcs ((Low _,High True):ss) =
          MotorStop:GetStatus:emit mcs ss
        goU mcs ((Low _,High _):ss) =
          MotorUp:GetStatus:goU mcs ss

        goD mcs ((Low True,High _):ss) =
          MotorStop:GetStatus:emit mcs ss
        goD mcs ((Low _,High _):ss) =
          MotorDown:GetStatus:goD mcs ss
```

And indeed the code now works just fine. A proof that the new code does not produce deadlocks is outside the scope of this paper but could be achieved either by compiling the code to a finite state machine or subjecting it to our liveness analysis [22].

5 Implementing the Debugger

Having argued that the debugger would be useful to include in a programming environment for Haskell, we turn our attention to the details of our current implementation in the context of the Chalmers Haskell compiler 'hbc.'

5.1 Background

The fundamental philosophy behind lazy evaluation [18] is that expressions are not evaluated unless their values are demanded. The initial demand usually starts from the top level routine that seeks to print the answer of the program. During debugging, the initial demand starts when the user clicks on one of the streams. In both cases the demands are propagated until they reach an expression that is already evaluated. Such an expression is said to be in *weak head normal form* (whnf). Examples of expressions in whnf include integers, λ-expressions, and pairs. Generally speaking an expression that consists of a constructor applied to arbitrary (possibly unevaluated) argument expressions is in whnf. For example, the expression `(1+2 : f xs)` is in whnf, since it consists of a top level constructor `(:)`, an unevaluated head `1+2`, and an unevaluated tail `f xs`.

Lazy evaluation is viable because once an expression has been evaluated, there is no need to re-evaluate it again. Since the language is purely functional, all subsequent evaluations of an expression are guaranteed to return the same result as the first, and hence are useless. This property immediately implies that it is impossible to write a Haskell expression that evaluates to `True` the first time, and to `False` the second time.

A deadlock occurs in a lazy functional program when the evaluation of an expression needs its own value to proceed. An obvious way of detecting deadlocks is thus to mark an expression as being under evaluation when its evaluation starts. If the value of such a marked expression is needed, we have a deadlock. Most compilers for lazy functional languages mark expressions in this way and can indeed detect deadlocks at run-time albeit with an obscure error message about black holes.

5.2 Global View

The debugger consists of a number of stream agents and a top level coordinator. A stream agent is responsible for evaluating and displaying the elements of one stream. The graphical representation of each stream agent is as shown in Figure 2.

The top level coordinator and the stream agents interact as follows. When the button Next of a stream agent is clicked, the agent tries to evaluate the next element of the stream. It then sends a notification message to the coordinator. Upon receipt of such a notification, the coordinator sends a broadcast message to all stream agents, to which each agent responds by checking for, and displaying, the evaluated elements on its stream.

5.3 Peeking at Streams

To perform their jobs, the stream agents need to peek at streams to see how many elements are already evaluated to whnf. Unfortunately, in a lazy functional language it is not possible to look at a stream to see how many elements have been produced. In fact, it is by looking at them that they will be produced, since there is demand for them! So when inspecting a stream, the agent would want to see the stream as a read-only data structure.

For this reason we have implemented a new primitive function peek that takes an argument and returns a value of type EvalStatus:

```
data EvalStatus = Evaluating | Closure | Evaluated
peek :: a -> EvalStatus
```

Clearly the above function is not pure and hence is not expressible in Haskell. Indeed, it may return different answers when called with the same argument twice. This function is not available to programmers.

Using peek we define a function safeInitialList that takes a stream as argument, and returns the initial list of already evaluated elements in the stream, a flag saying whether the stream has become deadlocked or not, and the stream itself (except for the initial evaluated segment):

```
safeInitialList :: [a] -> ([a], Bool, [a])
safeInitialList xs =
 case peek xs of
   Evaluating -> ([], True, xs)
   Closure    -> ([], False, xs)
   Evaluated  ->
    case xs of
     y:ys -> let (zs, b, ws) = safeInitialList ys
             in (y:zs, b, ws)
     []   -> ([], False, [])
```

5.4 Recovering from Fatal Errors

A fatal error is one which normally terminates a program with an error message, *e.g.*, a division by zero or a deadlock. Since our debugger is part of the same program as the application, we certainly don't want such errors to abort the debugger. To be able to recover from such faults and get some clues on which part of the source code is responsible for the fault, we introduce a pair of functions catch and throw:

```
throw :: String -> a
catch :: a -> (String -> a) -> a
```

that provide a simple form of exception handling. The semantics of these extensions is presented in the next section.

Intuitively, they are used as follows. The compiler is modified so that all fatal errors call throw with an appropriate error message. These errors are caught (handled) by the debugger. For example, the evaluation of result below produces 42:

```
result = catch (f []) handler
f (x:xs) = x + f xs
handler errMsg = 42
```

Otherwise, when not debugging, there will be no active catch and the program would terminate with the usual error message: "No match in f".

6 Call-by-Need & Exceptions

Consider a small functional *call-by-need* language extended with exceptions.

Definition 6.1 (Syntax) *The set of terms includes:*

$$\begin{array}{rcl} M, N & ::= & V \mid M + N \mid x \mid MN \\ & & \mid \texttt{throw}\ M \mid \texttt{catch}\ M\ N \\ V & ::= & n \mid \lambda x.M \end{array}$$

In other words, the language extends the pure λ-calculus with a representative set of constants including numbers and addition as well as two expressions for throwing and handling exceptions. The intended semantics of throw and catch is the usual one [35] adapted to a call-by-need language. Given an expression M, the evaluation of (throw M) aborts the computation and returns M. The evaluation of (catch M N) where N is a handler proceeds as follows. First evaluate M to whnf; if this evaluation terminates normally then its result is the value of the entire catch expression. Otherwise if the evaluation of M throws an exception, the handler is called with the thrown expression as an argument.

Definition 6.2 (Syntax of Types) *The set of types is inductively defined as follows:*

$$\tau \ ::= \ \texttt{Int} \mid \tau - \tau'$$

The typing rules for our language are in Figure 3. We let Γ scope over type environments (partial mappings from variables to types). A type judgment $\Gamma \vdash e : \tau$ means that under the assumptions in the type environment Γ, expression e has type τ.

The typing rules for the functional core are standard. The typing rules for throw and catch are similar to the ones found in SML [37] but, for simplicity, we have restricted all thrown expressions to be of type Int.

If typechecking guarantees anything, it is that some obviously faulty terms are not typable.

Definition 6.3 (Faulty Terms) *An expression is* faulty *if it is contains one of the following subterms:*

- $n\ M$, *or*
- $(\lambda x.M) + V$, *or*
- $V + (\lambda x.M)$.

It is easy to confirm that faulty terms are not typable.

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x:\tau} \qquad \frac{\Gamma \cup \{x:\tau\} \vdash M:\tau'}{\Gamma \vdash \lambda x.M : \tau \rightarrow \tau'} \qquad \frac{\Gamma \vdash M : \tau' \rightarrow \tau \qquad \Gamma \vdash N:\tau'}{\Gamma \vdash M\ N : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathtt{Int}} \qquad \frac{\Gamma \vdash M : \mathtt{Int} \qquad \Gamma \vdash N : \mathtt{Int}}{\Gamma \vdash M + N : \mathtt{Int}}$$

$$\frac{\Gamma \vdash M : \mathtt{Int}}{\Gamma \vdash \mathtt{throw}\ M : \tau} \qquad \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \mathtt{Int} \rightarrow \tau}{\Gamma \vdash \mathtt{catch}\ M\ N : \tau}$$

Figure 3. Typing Rules

6.1 Evaluation Contexts

Because of the *lazy* semantics of the language, the definition of the axioms is intertwined with the definition of evaluation contexts [2, 3]. Intuitively, "needed" variables within subterms correspond to those variables that occur in evaluation context positions. Therefore, we define evaluation contexts first before defining the semantics of the language.

Definition 6.4 (Evaluation Contexts E**)** *The set is inductively defined as:*

$$\begin{array}{rcl} E & ::= & [\,]\mid EM \mid (\lambda x.E)\ M \mid (\lambda x.E[x])\ E \\ & & \mid E + M \mid n + E \mid \mathtt{catch}\ E\ M \end{array}$$

The first four clauses are identical to the ones for the pure call-by-need calculus. They have the following intuitive explanation. The context EM reflects the fact that the function position of an application is needed. The next two contexts make the call-by-need nature of the language evident: arguments to functions are not evaluated when the function is called. Indeed, we first attempt to evaluate the body of the function without touching the argument. Only if the body of the function "needs" the value of the parameter, do we start processing the argument.

The contexts $E+M$ and $n+E$ state that both arguments to an addition are needed: first the left one, then the right one. In the presence of exceptions and handlers, this order is clearly significant. Finally the context catch E M reflect our informal understanding of the semantics of a catch expression which first attempts to evaluate the first argument. The second argument (the handler) is only needed if the evaluation of the first argument throws an exception.

6.2 Axioms

We are almost in a position to define the semantics of our language. We begin by identifying a syntactic category of terms that are *answers* and how to observe them.

Definition 6.5 (Answers) *An answer A is one of the following terms:*

$$A \ ::= \ V \mid \mathtt{throw}\ M \mid (\lambda x.A)\ M$$

When observed, answers yield the following output to users:

$$\begin{array}{rcl} obs(n) & = & n \\ obs(\lambda x.M) & = & \text{"Procedure"} \\ obs(\mathtt{throw}\ M) & = & \text{"Uncaught Exception"} \\ obs((\lambda x.A)\ M) & = & obs(A) \end{array}$$

Our definition of answers extends the usual definition for the pure call-by-need calculus by adding an additional kind of answers for uncaught exceptions.

We can now specify the semantics of the language via the set of axioms in Figure 4. The first three axioms are essentially the axioms of the pure call-by-need calculus. The next three axioms specify the semantics of exceptions. A throw expression aborts any context other than a catch. The value of a catch expression is either the value of the first argument or the result of applying the handler to the value thrown by the first argument.

The following proposition guarantees that the definitions of evaluation contexts, faulty expressions, and axioms are consistent and complete.

Proposition 6.1 *Every closed term is either an answer, faulty, or can be uniquely decomposed into an evaluation context and a redex.*

6.3 Evaluator

The above axioms can be used in any context to transform programs (perhaps for optimization purposes). They can also be used to specify an evaluator by orienting them from left to right and always choosing the redex that occurs in the hole of the evaluation context.

Definition 6.6 *The call-by-need evaluator is defined as follows: eval(M) = B if $M \longmapsto^* A$ and obs(A) = B, where $M \longmapsto N$ if and only if:*

- $M = E[R]$,
- *$R = R'$ is an axiom of Figure 4 (R must be the left hand side), and*
- $N = E[R']$.

6.4 Connection to Call-by-Name

Clearly the semantics of a call-by-need with exceptions is *not* the same as the semantics of the call-by-name variant. This is in contrast to the pure functional subset in which call-by-name reasoning principles like β are sound. However this is no worse than the current intermediate languages of Haskell compilers, like the Glasgow Haskell compiler, that use a call-by-need language with assignments to implement monadic state efficiently. Indeed β is also unsound in the intermediate language of the compiler for precisely the same reason [31, 25].

Abbreviations for one-step contexts:

$$\begin{array}{rcl} G & ::= & H \mid \texttt{catch}\,[\,]\,M \\ H & ::= & ([\,]\,M) \mid [\,] + M \mid n + [\,] \mid (\lambda x.E[x])\,[\,] \end{array}$$

Axioms:

$$\begin{array}{rcll} (\lambda x.E[x])\,V & = & (\lambda x.E[V])\,V & \\ G[(\lambda x.A)\,M] & = & (\lambda x.G[A])\,M & \\ n_1 + n_2 & = & n & \text{where } n = n_1 + n_2 \\ \texttt{catch}\,V\,M & = & V & \\ H[\texttt{throw}\,M] & = & \texttt{throw}\,M & \\ \texttt{catch}\,(\texttt{throw}\,M)\,N & = & NM & \end{array}$$

Figure 4. Axioms

7 Current Limitations and Future Work

We have presented a novel debugger for reasoning about the recursive computations over streams usually found in reactive systems. The debugger appears to work remarkably well for small programs, but there is a need for further work to handle larger programs and to formalize the theoretical foundations of the current implementation or to investigate different implementation strategies. We expand on these points in the remainder of the section.

There are three main limitations of the current system. First, the number of streams that can be simultaneously debugged is limited by the size of the screen. In practice we have not found this to be a problem. In large examples, it is relatively easy to use the debugger to narrow down the deadlock to a handful of streams.

Second, our implementation relies on impure primitives that, although confined to the internals of the compiler, could pollute the source language and break some obvious program optimizations. For example, the following two definitions of ones are equivalent in pure Haskell but not equivalent during debugging sessions:

```
ones = map id (1 : ones)
ones = 1 : ones
```

To see the difference consider the call `(safeInitialList ones)`. Using the first definition of ones, the result is a triple of the form `([], False, ones)`. Using the second definition of ones, the call (and hence the debugger itself) diverges! To avoid such situations, a fully evaluated stream cannot be used during debugging sessions. Indeed in the elevating table example, the definition that we used for `motorCmds` is actually:

```
motorCmds = map id (Up : Down : motorCmds)
```

This trick of inserting `map id` around trivial stream definitions works fine with the current version of the 'hbc' compiler but is unsatisfactory. A smart compiler that is unaware of the debugger can certainly optimize `(map id e)` to `e` since they are equivalent in pure Haskell. A better solution would be to integrate the debugger and the compiler in a programming environment which would require them to agree on a common representation for streams such as `motorCmds` above.

As another example, addition is commutative in pure Haskell, *i.e.*, $M + N = N + M$. This equivalence no longer holds in the presence of `catch` and `throw`. The evaluation of `result` below produces 5 or 6 depending on the order of evaluation of the arguments to `+`:

```
result = catch (f ()) handler
f () = throw "L" + throw "R"
handler errMsg = if errMsg=="L" then 5 else 6
```

General programmers do not have access to `catch` and `throw` and this situation does not seem to occur in the current debugger. Nevertheless a proof of safety would be interesting. This situation is reminiscent of the use of assignments in the implementation of monadic state without affecting the purity of the source language [24, 25].

Third, the debugger can only cope with streams defined at the top level of the program. Thus if we have a large program in which one function uses locally defined streams, we cannot debug the program until these streams are globally defined at the top level. Such a change may require a major restructuring of the program, and may even be impossible in certain circumstances. This problem is a general Haskell problem: it is impossible to perform any computational effect (*e.g.*, printing) as a side-effect of evaluating an expression. Instead computational effects must be propagated to the top level either explicitly or by using a monad [36]. This is the major limitation of the debugger; we are actively seeking a solution to this problem.

Finally, we have started an investigation of the semantics of computational effects in call-by-needs languages since we believe that this will lead to more robust programs, and will formalize some of the *ad hoc*, informal, and potentially unsafe programming practices in system-level Haskell programs.

References

[1] Abelson, H., and Sussman, G. J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. To appear in the *Journal of Functional Programming*, 1996.

[3] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In *ACM Symposium on Principles of Programming Languages* (1995), pp. 233–246.

[4] Ashcroft, E. A., and Wadge, W. W. Lucid, a nonprocedural language with iteration. *Communications of the ACM 20*, 7 (1977), 519–526.

[5] Ashcroft, E. A., and Wadge, W. W. *Lucid, the Data-Flow Programming Language.* Academic Press, 1985.

[6] Augustsson, L., Rittri, M., and Synek, D. Splitting infinite sets of unique names by hidden state changes. Tech. Rep. 67, Chalmers University of Technology, 1992.

[7] Barton, D. L., and Dunlop, D. D. An introduction to MHDL. In *Proceedings of IEEE 1993 MTT-S Symposium* (San Diego, CA, May 1993), pp. 1487–1490. (Paper UU-1).

[8] Benveniste, A., LeGuernic, P., and Jacquemot, C. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming 16* (1991), 103–149.

[9] Berry, G., and Cosserat, I. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency* (1985), S. Brookes and G. Winskel, Eds., vol. 197 of *Lecture Notes in Computer Science*, Springer.

[10] Berry, G., and Gonthier, G. The Esterel synchronous programming language, design, semantics, implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[11] Broy, M., et al. The design of distributed systems—an introduction to Focus. Tech. Rep. SFB-Bericht Nr. 342/2-2/92 A, Technische Universität München, 1992.

[12] Caspi, P., Halbwachs, N., Pilaud, D., and Plaice, J. Lustre: A declarative language for programming synchronous systems. In *ACM Symposium on Principles of Programming Languages* (1987).

[13] Caspi, P., and Pouzet, M. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming* (1996), pp. 226–238.

[14] Edwards, S. An Esterel compiler for a synchronous/reactive development system. `ftp://ic.eecs.berkeley.edu/pub/Esterel/esterelcomp.ps.gz`.

[15] Hall, C. V., Hammond, K., and O'Donnell, J. T. An algorithmic and semantic approach to debugging. In *Proceedings of the Glasgow Workshop on Functional Programming* (1990).

[16] Hall, C. V., and O'Donnell, J. T. Debugging in applicative languages. *Lisp and Symbolic Computation* (1988).

[17] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[18] Henderson, P., and Morris, Jr., J. H. A lazy evaluator. In *ACM Symposium on Principles of Programming Languages* (1976), pp. 95–103.

[19] Hudak, P., Peyton Jones, S. L., and Wadler, P. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices 27*, 5 (1992).

[20] Hughes, J. Lazy memo-functions. Tech. Rep. 21, Chalmers University of Technology, 1985.

[21] Hughes, J., and Moran, A. Making choices lazily. In *Conference on Functional Programming and Computer Architecture* (1995). To appear.

[22] Hughes, J., Pareto, L., and Sabry, A. Proving the correctness of reactive systems using sized types. In *ACM Symposium on Principles of Programming Languages* (1996).

[23] Kahn, G. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress* (1974), J. L. Rosenfeld, Ed., North-Holland, pp. 471–475.

[24] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8* (1995), 193–341.

[25] Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In *ACM SIGPLAN International Conference on Functional Programming* (1997).

[26] Lewerentz, C., and Lindner, T. *Formal Development of Reactive Systems: Case Study Production Cell.* Lecture Notes in Computer Science 891. Springer-Verlag, 1995.

[27] Naish, L., and Barbour, T. Towards a portable lazy functional declarative debugger. Technical Report 95/27, Department of Computer Science, University of Melbourne, Australia, 1995.

[28] Nilsson, H. A declarative approach to debugging for lazy functional languages. Licentiate Thesis No. 450, Department of Computer and Information Science, Linköping University, Sept. 1994.

[29] Nilsson, H., and Fritzson, P. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming 4*, 3 (July 1994), 337–370.

[30] Nilsson, H., and Sparud, J. The evaluation dependence tree as a basis for lazy functional debugging. *Journal of Automated Software Engineering 4*, 2 (April 1997), 152–205.

[31] Sabry, A. What is a purely functional language? Unpublished manuscript, April 1997.

[32] Sijtsma, B. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems 11*, 4 (1989), 633–649.

[33] Sparud, J. A transformational approach to debugging lazy functional programs. Licentiate Thesis, Department of Computing Science, Chalmers University of Technology, Feb. 1996.

[34] Sparud, J., and Nilsson, H. The architecture of a debugger for lazy functional languages. In *Proceedings of AADEBUG '95, 2nd International Workshop on Automated and Algorithmic Debugging* (Saint-Malo, France, May 1995), M. Ducassé, Ed., IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, Cedex, France.

[35] Steele, G. L. *Common Lisp—The Language.* Digital Press, 1984.

[36] Wadler, P. Comprehending monads. In *ACM Conference on Lisp and Functional Programming* (1990), pp. 61–78.

[37] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 91-160, Rice University, April 1991. Final version in *Information and Computation* **115** (1), 1994, 38–94.

Bulk types with class

Simon Peyton Jones
University of Glasgow
Email: simonpj@dcs.gla.ac.uk
WWW: http://www.dcs.gla.ac.uk/~simonpj

May 1, 1997

Abstract

Bulk types — such as lists, bags, sets, finite maps, and priority queues — are ubiquitous in programming. Yet many languages don't support them well, even though they have received a great deal of attention, especially from the database community. Haskell is currently among the culprits.

This paper has two aims: to identify some of the technical difficulties, and to attempt to address them using Haskell's constructor classes.

This paper appears in the proceedings of the 1997 Haskell Workshop, Amsterdam, 7 June 1997. A slightly earlier version appears in the (electronic) proceedings of the 1996 Glasgow Functional Programming Workshop:

`http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Proceedings96.html`

1 Introduction

Functional programs use a lot of lists, but often a list is actually used to represent:

> a stack, a queue, a deque, a bag, a set, a finite map (by way of an association list of (key,value) pairs), or a priority queue.

Using lists for all of these so-called *bulk types* is bad programming style for two reasons:

1. The type of the object does not specify its invariant (e.g. in a set there are no duplicates) and its expected operations (e.g. lookup in a finite map). The lack of these invariants makes the program harder to understand, harder to prove properties about, and harder to maintain.
2. Operations on lists may be less efficient, or perhaps even in a different complexity class, than operations on a suitably optimised abstract data type. For example, list append

(++) takes time linear in the size of its first argument, whereas it is easy to implement an ordered sequence ADT with constant-time concatenation[1].

Everyone knows this, but everyone still uses lists! Why? Because lists are well supported by the language: they admit pattern matching, there is built-in syntax (list comprehensions), and there is a rich library of functions that operate over lists. Even experienced functional programmers knowingly write an $O(n^2)$ algorithm where an $O(n)$ algorithm would do, because it is just so convenient to use lists and append them rather than to design and implement and use an abstract data type.

Why, then, aren't there well-engineered libraries to support sets, bags, finite maps, and so on? Many decent attempts have been made, notably C++'s standard template library (STL) - see Section 5 - but all have technical difficulties. This paper identifies some of these difficulties and attacks them using Haskell's type classes.

2 The problem with bulk types

The central difficulty with bulk types is their degree of polymorphism. First, there are many different sorts of collections — lists, sets, queues, and so on. Second, one such sort may have many different possible representations — lists, trees, hash tables, and so on. Lastly, each such representation may have many different element types — integers, booleans, characters, pairs, and so on.

A language that supports *polymorphism* allows the programmer to write a single algorithm that can be used in many "essentially similar" situations. For example, suppose we want to construct the list (or set, or bag) of leaves of a tree, where the tree is defined by the following data type:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

Here is a possible algorithm that works for a tree with `Int` leaves, constructing a set of `Int`s:

```
leavesSetInt :: Tree Int     -> SetInt
leavesSetInt (Leaf a)        = singletonSetInt a
leavesSetInt (Branch t1 t2) = leaves t1 `unionSetInt` leaves t2
```

This code assumes the existence of the following set construction functions:

```
singletonSetInt :: Int -> SetInt
unionSetInt     :: SetInt -> SetInt -> SetInt
```

There are two ways in which this program can be made more polymorphic:

Element polymorphism Firstly, it is obvious that code of precisely the same form would be required for a tree of booleans. We would like to be able to generalise `leaves` like this:

[1]At least, it is easy if one is prepared to give up O(1) head and tail functions. It is possible, albeit somewhat more complex, to support append, head and tail all in constant (amortized) time (Okasaki [1995]).

```
leavesSet :: Tree a        -> Set a
leavesSet (Leaf a)         = singletonSet a
leavesSet (Branch t1 t2) = leaves t1 `unionSet` leaves t2
```

To make this work we would need to have these set operations:

```
singletonSet :: a -> Set a
unionSet     :: Set a -> Set a -> Set a
```

Bulk-type polymorphism Suppose that we have a second data type, `OrdSet`, that uses a different representation from that of `Set` — perhaps `Set` represents the set as a list with no duplicates, while `OrdSet` uses a balanced tree, for example. The function to gather the leaves of a tree into an `OrdSet` will be of just the same form as that for `Set`. The same is true if we want to collect the leaves into a bag, or a priority queue, or a list. Ideally, then, we would like to make `leaves` more polymorphic still, something like this:

```
leaves :: Tree a        -> c a  -- where c is a bulk type
leaves (Leaf a)         = singleton a
leaves (Branch t1 t2) = leaves t1 `union` leaves t2
```

where the bulk-type constructors are now something like:

```
singleton :: a -> c a          -- where c is a bulk type
union     :: c a -> c a -> c a -- where c is a bulk type
```

The trouble is that neither of these two generalisations is straightforward. We discuss each in turn.

2.1 Element polymorphism

Consider the goal of making `leaves` polymorphic in the elements of the `Set`. The tidiest kind of polymorphism, *parametric polymorphism*, works when the very same source code will work regardless of the argument type. It is supported by many modern programming languages, including C++, ML, and Haskell[2]. If we were collecting the leaves of a tree into a list, then we could use parametric polymorphism very easily:

```
leavesList :: Tree a -> [a]
leavesList (Leaf x) = singletonList x
leavesList (Branch t1 t2) = leavesList t1 `unionList` leavesList t2
```

Here, `unionList` has type `[a] -> [a] -> [a]`; it is just list append, commonly written `++`.

The trouble arises with sets, because *we cannot make a* `union` *operation that works on sets whose elements of arbitrary type.* To remove duplicates we must at least have equality on the set elements! Furthermore, equality may not be enough:

[2]In the case of ML and Haskell, this polymorphism extends to the executable code too; that is, the same executable code works regardless of the argument type. In the case of C++, using templates one can have a single source-code function, but the compiler must instantiate it separately for each type at which it is used.

- If the element type admits only equality, then determining whether an element is a member of the set must take linear time.
- If the element type supports a total order then a tree (balanced or otherwise) may be more appropriate, and set membership can be determined in logarithmic time.
- If the element type admits a hash function, then the set might be represented by a hash table, or — in a purely-functional language where *persistent data structures*[3] are the rule — by a tree indexed on the hash key.
- If the element type has a one-to-one function mapping elements to integers, then radix-based tree representations become possible.

One way out of this dilemma, taken by Java for example, is to decide that every data type supports equality, together with ordering and/or a hash function. This is simple but crude — what about equality of functions, for example? A cleaner solution, adopted by ML for equality, and generalised in Haskell by type classes, is to use a type system that allows type variables to be qualified by the operations they support. Thus, in Haskell we can give the following type for union on a set data type that required only equality:

```
unionSet :: (Eq a) => Set a -> Set a -> Set a
```

This type specifies that the element type, a, must lie in the class Eq[4]. The class Eq is defined like this:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The declaration says that types that are instances of Eq must provide operations (==) and (/=) with the given types. For each data type that we want to be in Eq we must give an instance declaration that defines (==) and (/=) at that type. For example:

```
instance Eq Int where
  x == y = x `eqInt` y
  x /= y = not (x `eqInt` y)

instance (Eq a, Eq b) => Eq (a,b) where
  (a1,b1) == (a2,b2) = (a1==a2) && (b1==b2)
```

Given this type for unionSet, the type of leavesSet is now inferred to be:

```
leavesSet :: (Eq a) => Tree a -> EqSet a
```

[3] A data structure is "persistent" if, following an update, the old version of the data structure is still available (Okasaki [1996]).

[4] Strictly speaking, the semantics of union does not require the elimination of duplicates — that could be postponed until the set is observed by a membership test or by enumerating its elements. However, nothing fundamental is changed by such an implementation decision so in this paper we will stick with the naive view that union requires equality.

If our Set type required ordering as well as equality, we would simply replace (Eq a) by (Ord a) in the above types.

2.1.1 Other approaches

ML has equality types built in, but not ordered types, so the Haskell solution is not available in ML. (Restricting to equality only would be unreasonable, because sets based only on equality are hopelessly inefficient.) The solution adopted by some ML libraries is to make Set into a functor:

```
functor Set( ORD:ORD_SIG ) : SET
```

That is, Set is a functor taking an ordering as its argument, and producing a set structure (i.e. module) as its result. One can thereby construct efficient set-manipulation functions for particular element types:

```
IntSet  = Set IntOrd
CharSet = Set CharOrd
```

but now the leaves function has to mention either IntSet.union or CharSet.union — leaves cannot be polymorphic in the element type. To solve this, leaves must be defined in a functor that takes the Set structure as argument, and so on.

2.2 Bulk-type polymorphism

Next, we consider how to generalise leaves to work over arbitrary bulk types. To begin with we will consider only types — such as lists, queues, and stacks — that are truly parametric in their element types.

2.2.1 Using type classes

We start off with one union operation for each collection type, each of which has quite different code to the others:

```
unionList :: [a] -> [a] -> [a]
unionQueue :: Queue a -> Queue a -> Queue a
...etc...
```

In order to generalise leaves, we earlier informally suggested the type:

```
leaves :: Tree a -> c a  -- where c is a bulk type
```

We are suggesting here that leaves is polymorphic in c, the bulk type constructor. The polymorphism is not parametric, however, because each union operation uses different code; leaves should call a different union operation for each type. This is exactly what type classes are for! Perhaps we could write:

```
  leaves :: (Bulk c) => Tree a -> c a
```

where Bulk is the class of bulk types, defined thus:

```
  class Bulk c where
    empty     :: c a
    singleton :: a -> c a
    union     :: c a -> c a -> c a
```

Now we can give an instance declaration for each Bulk type:

```
  instance Bulk [] where         -- [] is the List type constructor
    empty       = []
    singleton x = [x]
    union       = (++)

  instance Bulk Queue where
    empty     = emptyQueue
    singleton = singletonQueue
    union     = unionQueue
```

All of this is legal Haskell, but notice that c is a variable that ranges over *type constructors* rather than *types*. This sort of higher-kind quantification is a fairly straightforward but powerful extension of the Hindley-Milner type system (Jones [1995]). It can be used in ordinary data type declarations but, as we shall see, it is particularly useful in Haskell's system of classes, which are thereby generalised from type classes to constructor classes.

Alas, things go wrong when we try to deal with non-parametric element types. We cannot give an instance declaration:

```
  instance Bulk Set where
    empty     = emptySet
    singleton = singletonSet
    union     = unionSet
```

because unionSet has the wrong type! It requires that the element type be in Eq, whereas the overloaded union operator does not.

2.2.2 Other approaches

A possible alternative approach is to use *ad hoc* polymorphism. The symbol union would stand for a whole family of union operations, each with a different type. The choice of which to use would be made statically by the compiler, based on local type information. ML uses this sort of overloading for numeric operators, and so does C++, Ada, and other languages. The small disadvantage of *ad hoc* overloading is that one may need to write type signatures to specify which type to use; "small" because writing type signatures is a Good Idea anyway.

The big disadvantage is that one cannot write generic operations over collections. For example, we could write leaves thus:

```
leaves (Leaf a)       = singleton a
leaves (Branch t1 t2) = leaves t1 `union` leaves t2
```

but the compiler would have to resolve the `union` to `unionList`, or `unionQueue` or `unionSet`, or whatever. This resolution might be done implicitly, or by requiring the programmer to add a type signature; but however it is done `leaves` will only work on collections of one type. An exact copy of the code, with a different type signature, would deal with one more type, and so on. Every time you add a new collection type you would need to add a new copy of `leaves`. (Or perhaps the compiler could automatically make them all for you, in which case the issue is one of code size.)

2.3 Adaptive representations

There is a third issue which adds yet more spice to the challenge of implementing bulk types: that of choosing an appropriate representation. The appropriate representation of a collection depends on:

1. The size of the collection.
2. The relative frequency of the operations supported by the bulk type.
3. The operations that are available on the underlying element type.

Of course, we can simply dump the problem in the programmer's lap, by providing a large variety of different set data types, and leaving the choice to the programmer. (This is precisely what STL does.) A more attractive alternative is to make the bulk type choose its own representation.

Items (1) and (2) have been fairly well studied. Clever algorithms have been developed that adapt the representation of a data type based on its size and usage (Brodal & Okasaki [1996]; Chuang & Hwang [1996]; Okasaki [1996]). It is less obvious how to tackle item (3). How can we build an implementation of `Set` that chose its representation based on what operations are available on the elements? We return to this question in Section 3.3.

2.4 Summary

In this section we have reviewed various approaches to manipulating bulk types in polymorphic fashion. The bottom line is that "nothing quite works". Bulk types seem quite innocent, but the combination of polymorphism in both element and bulk types, and the non-parametric nature of both, conspire to defeat even the most sophisticated type systems.

3 First design: the `XOps` route

In this section we turn to our first solution, based on the most promising of the approaches reviewed, namely constructor classes. The solution we present has the merit of being implementable in standard Haskell (1.3), but it has some shortcomings that we will address in our second solution (Section 4).

Like C++'s STL, we identify two main groups of bulk types:

1. *Sequences*, where the order of insertion is significant (e.g. one can extract the most recently inserted element), but where no operations need be performed on the elements themselves.

2. *Collections*, where the order of insertion is unimportant, but where the elements must admit at least equality and preferably some other operations[5].

3.1 Sequences

A *sequence* contains a linear sequence of zero or more elements. The order of insertion and removal of elements is significant, and elements can be added or removed at either end. Examples of sequences are: *lists*, *catenable lists*[6], *stacks*, *queues*, *deques*. They all support the same set of operations, but they differ in the complexity bounds for these operations.

Figures 1 and 2 defines a module `Sequence` whose main declaration is a type class, also called `Sequence`, that defines the set of operations on sequences. The names of the operations are chosen to be compatible with Haskell's current nomenclature for lists. `front` and `back` return both the first (respectively, last) element of the sequence, together with the remaining sequence; they return `Null` if the sequence is empty. The `SeqView` type is used as the return type for both of these functions: you can think of `front` and `back` as providing a head-and-tail-like "view" of each end of the sequence.

The `fold` functions, along with `length`, `filter`, `partition`, `reverse`, are straightforward generalisations of their list counterparts. They can all readily be defined in terms of either `front` or `back`. Indeed, each of them has a default method in the class declaration, indicating that an instance of `Sequence` may (but is not compelled to) provide a method for these operations. The reason for this decision is that for at least some instances of `Sequence` (snoc-lists, say) the default definition of some functions (`foldr`, in this case) is likely to be outrageously inefficient. Making these functions into class methods gives the implementor the option (though not the obligation) of providing more efficient definitions.

The standard classes `MonadPlus` and `Functor` are superclasses of `Sequence`: that is, any type in `Sequence` must also be in `MonadPlus` and `Functor`. Both of the latter are defined by the Haskell 1.3 prelude. Figure 3 gives their definitions, except that we have added `cons` and `snoc` to the class `MonadPlus`. They can both be implemented in terms of `++`, as their default methods show, but for many types they can be more efficiently implemented directly.

All the operations of the standard classes `Monad`, `MonadZero`, `MonadPlus`, and `Functor` make sense for sequences: `++` appends two sequences; `map` applies a function to each element of a

[5]STL refers to these as "associative containers".
[6]Catenable lists support constant-time append.

```
module Sequence where

data SeqView s a = Null | Cons a (s a)

empty :: Sequence s => s a
empty = zero

singleton :: Sequence s => a -> s a
singleton x = return x

fromList :: Sequence s => [a] -> s a
fromList xs = foldr ((++) . return) zero xs

toList   :: Sequence s => s a -> [a]
toList s = foldr (:) [] s

class (Functor s, MonadPlus s) => Sequence s where
  null  :: s a -> Bool
  front :: s a -> SeqView s a
  back  :: s a -> SeqView s a

  (!!) :: s a -> Int -> a
  update :: s a -> Int -> a -> s a

  foldr  :: (a -> b -> b) -> b -> s a -> b
  foldr1 :: (a -> a -> a) -> s a -> a
  foldl  :: (b -> a -> b) -> b -> s a -> b
  foldl1 :: (a -> a -> a) -> s a -> a

  length :: s a -> Int

  elem :: (Eq a) => a -> s a -> Bool

  filter :: (a -> Bool) -> s a -> s a
  partition :: (a -> Bool) -> s a -> (s a, s a)

  reverse :: s a -> s a
```

Figure 1: The sequence class

```
        -- Default methods
  foldr k z xs = case front xs of
                   Null -> z
                   Cons x xs -> x `k` foldr k z xs
  foldr1 k xs = case back xs of
                  Cons x xs -> foldr k x xs
  foldl k z xs = case front xs of
                   Null -> z
                   Cons x xs -> foldl k (z `k` x) xs
  foldl1 k xs = case front xs of
                  Cons x xs -> foldl k x xs

  length xs = foldr (\_ n -> n+1) 0 xs
  filter p xs = foldr f zero xs
            where f x ys | p x = x `cons` ys
                         | otherwise = ys
  partition p xs = (filter p xs, filter (not.p) xs)
  reverse xs = foldl (flip.cons) zero xs
  elem x xs = foldr ((||) . (==) x) False xs
```

Figure 2: The sequence class, continued

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

class (Monad m) => MonadZero m where
  zero :: m a

class (MonadZero m) => MonadPlus m where
  (++) :: m a -> m a -> m a

  cons :: a -> s a -> s a
  snoc :: s a -> a -> s a

  cons x xs = (return x) ++ xs          -- Not yet in 1.3
  snoc xs x = xs ++ (return x)          -- Not yet in 1.3

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

Figure 3: Monad and functor classes

sequence; `zero` is the empty sequence; `return` forms a singleton sequence; and `>>=` takes a function that maps each element of a sequence to a new sequence, and concatenates the results.

Because sequences lie in the class `Monad` we can use the `do` notation to describe sequence-valued expressions. For example:

```
do { x<-xs; y<-ys; return (x,y) }
```

will deliver the sequence composed of all `(x,y)` pairs, where `x` is drawn from the sequence `xs` and `y` is drawn from `ys`. The same applies to Haskell's comprehension notation, which also defines an expression over any monad. For example, this comprehension defines the same sequence as that above.

```
[(x,y) | x<-xs, y<-ys]
```

A great deal of work has been done on the connection between bulk types and comprehension syntax, especially in the context of database queries and their optimisation (Buneman et al. [1994]; Trinder [1991]; Trinder & Wadler [1989]; Wadler [1992]).

Lastly, the `Sequence` module contains a couple of ordinary declarations that give more collection-oriented names to the monadic functions `zero` and `return`, namely `empty` and `singleton` respectively.

3.2 Collections

We observed earlier that the problem with collections is that the `union` operation may impose different constraints on the element type, depending on which collection we are dealing with. Our solution is very simple, namely to give them all the same type. First we define a new class `XOps`, the class of element operations, thus:

```
class XOps a where
  xEq     :: a -> a -> Bool
  xCmp    :: Maybe (a -> a -> Ordering)  -- Three-way comparison
  xHash   :: Maybe (a -> Int)            -- Hash function; could be many-one
  xToInt  :: Maybe (a -> Int)            -- Injection; guaranteed one-one
```

(`Ordering` is a standard Haskell data type with three constructors, `LT`, `EQ` and `GT`.) The point about `XOps` is that it tells not only how to (say) compare two elements, but *also whether such a comparison is available*. The equality operation, however, is mandatory, so it is not wrapped in a `Maybe` type[7]. For example, for a particular type `T`, we might have an instance declaration:

```
instance XOps T where
  xEq    = (==)
  xCmp   = Just cmpT
  xHash  = Nothing
```

[7]We could make Eq a superclass of `XOps` instead of having xEq, but it is sometimes convenient to define a non-standard equality for collection operations — see Section 3.6 — and it is confusing to have non-standard instances of Eq.

```
    xToInt = Nothing
```

to say that T had a comparison operation, cmpT, but no xHash or xToInt operation.

Next, we define the class Collection, of collections, like this:

```
  class Collection c where
    empty :: c a
    insert :: XOps a => a -> c a -> c a
    ...and much more...
```

We will add many further operations shortly. We use empty for both collections and sequences, relying on the use of qualified names (such as C.empty) to distinguish them when necessary.

With these definitions, it is now possible to give a fully-respectable type to leaves:

```
  leaves :: (Collection c, XOps a) => Tree a -> c a
```

Notice that, unlike the Sequence class, *we cannot make* MonadPlus *and* Functor *into superclasses of* Collection. Why not? Because for collections we cannot give sufficiently polymorphic definitions for ++ and >>=. To perform these operations we will need the constraint XOps t on the element type t — but that would not fit the signature of the classes MonadPlus and Functor.

3.3 Instances of Collection

Next, suppose we have a datatype, OrdSet that implements sets using trees, making use of an ordering operation on the set's elements. We can make OrdSet, the type of ordered sets (whose implementation depends on an element ordering), an instance of Collection thus:

```
  instance Collection OrdSet where
    empty = Empty
    insert x t = case xCmp of
                   Just cmp -> insertTree cmp x t
                   Nothing  -> error "OrdSet.insert"
```

(Here we are assuming the existence of a suitable data type of Trees, with operations insertTree to insert an element.) An obvious sadness is that if we try to build an OrdSet of things that only admit equality then we will only get a runtime error, not a compile-time type error. Whilst this is undoubtedly sad, we will see shortly how to design set datatypes that cannot fail in this way. Furthermore, it is worth remembering that most programs contain quite a few functions with incomplete patterns. To take a simple example:

```
  head :: [a] -> a
  head (x:xs) = x
  head [] = error "head"
```

Of course, head is only called when (we think that) we know the argument is a non-empty list. It would be nice if the type system proved this, and one could imagine more sophisticated type systems that could (Freeman & Pfenning [1991]), but Haskell and ML are certainly not rendered unusable by the possibility of such runtime errors. The error in `insert` is arguably in this class.

However, it would really be best to avoid even the possibility of run-time failure, and we can do this by building a `Set` data type that *chooses its representation based on the available operations on elements.* Here is a sketch of one possible implementation:

```
data Set a = Empty
           | List [a]        -- No duplicates
           | Tree (Tree a)

instance Collection Set where
  empty = Empty

  insert x Empty
      = case xCmp of
          Just cmp -> Tree (Branch x Empty Empty)
          Nothing  -> List [x]

  insert x (List xs) = List (insertList xEq x xs)

  insert x (Tree t)
      = case xCmp of
           Just cmp -> Tree (insertTree cmp x t)
```

The point of the game is that `insert` dynamically selects which representation to use in the `Empty` case depending on whether or not there is a comparison operation. Notice, crucially, that *the extraction of* `cmp` *in the final equation for* `insert` *cannot fail, because one of the arguments is already a* `Tree`, *and it could only have become so by virtue of the* `Empty` *equation deciding that there was a comparison operation.*

Not only have we eliminated runtime errors, but we have also delegated to the abstract data type the choice of representation. This is a rather attractive property. When computing with sets, most programmers do not want to have to look up the operations that are available for the element type, and choose which set implementation to use depending on the answer. Being able to use a single type, `Set`, and having the implementation choose the representation automatically is a big advantage. Of course, we are still free to fix a particular representation by using a simpler, more specific set implementation (such as `OrdSet`).

3.4 Efficiency

The generic `Set` implementation sketched above is just a start. A real implementation would be rather cleverer.

- Very small sets should probably be represented by lists even if ordering is available. This is easily programmed.
- A good compiler should be able to create specialised instances of `insert` at widely-used types. For example, if it sees that `insert` is often used at the type

  ```
  insert :: String -> Set String -> Set String
  ```

  then it can create a specialised version of `insert`, in which `c` is fixed to `Set` and `a` is fixed to `String`, and hence the comparison operations ought to be turned into inline code.

There are two other efficiency concerns about `Set` that turn out to be relatively unimportant:

- The implementation of `insert` has to choose which equation to use based on which constructor it finds in its second argument. However, in most implementations the major cost is doing pattern-matching (and hence forcing evaluation) at all; it is very little more expensive to choose between equations based on the constructor found.
- One might worry that every call to `insert` has to pattern-match on `xCmp` to extract the comparison operation, which carries an efficiency cost. This can be done once and for all when a tree is first built:

  ```
  data Tree a = Tree (a->a->Ordering)    -- Comparison
                     Int                 -- Size
                     (TreeR a)
  data TreeR a = Empty | Branch a (TreeR a) (TreeR a)
  ```

  On the whole, though, this is probably a bad thing to do. If the implementation fetches the ordering function from the tree, it is less likely that the compiler will be able to prove that for some given type, `Int` say, the ordering function is bound to be `cmpInt`. So it may be less easy for the compiler to generate improved code when the types are known.

3.5 Taking collections apart

The operations on collections we have suggested so far (`empty`, `insert`, `union`) only deal with *constructing* collections. What about taking collections apart? The obvious thing to do is to augment class `Collection` with a homomorphism over the constructors of the collection. Since our "constructors" (so far) are `empty` and `insert` the obvious homomorphism to add to `Collection` is:

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

```
module Collection where

class Collection c where
  empty  :: c a
  null   :: c a -> Bool
  size    :: c a -> Int

  singleton :: (XOps a) => a -> c a
  fromList  :: (XOps a) => [a] -> c a
  toList    :: c a -> [a]

  fold    :: (a->b->b) -> b -> c a -> b
  fold1   :: (a->b->b) -> c a -> b

  filter    :: (XOps a) => (a -> Bool) -> c a -> c a
  partition :: (XOps a) => (a -> Bool) -> c a -> (c a,c a)
  elem      :: (XOps a) => a -> c a -> Bool

  flatMap :: (XOps b) => c a -> (a -> c b) -> c b

  insert     :: (XOps a) => a -> c a -> c a
  insertWith :: (XOps a) => (a->a->a) -> a -> c a -> c a
  insertK    :: (XOps k) => k -> a -> c (Pr k v) -> c (Pr k v)

  union     :: (XOps a) => c a -> c a -> c a
  unionWith :: (XOps a) => (a->a->a) -> c a -> c a -> c a

  delete     :: (XOps a) => a -> c a -> c a
  deleteK    :: (XOps k) => k -> c (Pr k v) -> c (Pr k v)

  lookup      :: (XOps k) => k -> c (Pr k v) -> Maybe v

  intersect, without :: (XOps a) => c a -> c a -> c a
```

Figure 4: The collection class

```
      -- Default methods (part of class declaration)
  size c = fold (\_ n -> n+1) 0 c
  null c = size c == 0

  singleton x = insert x emptyC

  toList c    = fold (:) [] c
  fromList xs = insertList xs emptyC

  filter p c = fold f empty c
             where
               f x r | p x       = x `insert` r
                     | otherwise = r
  partition p c = (filter p c, filter (not.p) c)

  elem x c = fold (\y r -> if (x==y) then True else r) False c

  flatMap c f = fold (union.f) empty c

  insert           = insertWith (\x y -> y)
  insertWith f x c = unionWith f c (singleton x)
  insertK k x c    = insert  (k:>x) c

  union             = unionWith (\x y -> y)
  unionWith f c1 c2 = fold (insertWith f) c1 c2

  delete x c      = filter (/= x) c
  deleteK k c     = delete (k :> error "Collection.deleteK") c

  without   c1 c2 = filter (\x -> not (elem x c2)) c1
  intersect c1 c2 = filter (\x -> elem x c2) c1


-- Standard functions defined using class operations
insertList,deleteList :: (XOps a) => [a] -> c a -> c a
insertList xs c  = foldr insert c xs
deleteList xs c  = foldr delete c xs

unionList, intersectList :: XOps a) => [c a] -> c a
unionList cs      = foldr union empty cs
intersectList cs  = foldr1 intersect cs
```

Figure 5: The collection class continued

The question is, of course, what meaning we should give to a call such as `(fold (-) 0 s)` where `s` is a set. Since `(-)` is not commutative such a call is nonsense. There is no way out of this. All we can do is specify in any particular instance of `Collection` what property `fold` assumes of its arguments. For example, for sets and bags `fold`'s first argument should be left-commutative (i.e. $f\ x\ (f\ y\ a) = f\ y\ (f\ x\ a)$), but there may be instances of `Collection` for which this property need not hold (ones which guarantee to apply `fold` to their elements in sorted order, for example). For arguments that do not satisfy the required properties, `fold` delivers a result based on an unspecified ordering of the elements of the collection.

`fold` is a compositional form of what in STL is called an *iterator*. It lays out the collection in some order, ready to be operated on by some consuming function.

This `fold` is a catamorphism if we regard a collection as built by the constructors (`empty`, `insert`). An equally valid alternative set of constructors is (`empty`, `singleton`, `union`), leading to a different catamorphism:

```
  fold' :: (b->b->b) -> (a->b) -> b -> c a -> b
```

These two algebras have been explored by Buneman et al. [1995], who use the terminology `sr_add` for `fold`, and `sr_comb` for `fold'`. We have chosen to use `fold` because it is easier to use than `fold'` — only two arguments need be provided.

As we have seen, `fold` is a bit too powerful because in order to be well defined we have to assume undecidable properties of its argument. Buneman et al. [1995] also discusses ways to avoid this by instead using a function they call `ext`, but which we called `flatMap` in Figure 4. The advantage of `ext/flatMap` is that it requires no particular properties of its argument; yet using it one can define a bunch of useful functions.

3.6 Finite maps

Finite maps (in various guises) are ubiquitous in functional programs. In mathematics, a function (or map) is defined by a set of ordered (argument,result) pairs. The natural thing to do is therefore to represent a finite map by a set of ordered pairs, thus:

```
type FM k v = Set (Pr k v)
data Pr k v = k :> v deriving( Show )

instance (Eq k) => Eq (Pr k v) where


instance (XOps k) => XOps (Pr k v) where
  (k1:>v1) `xEq` (k2:>v2) = k1 `xEq` k2
  xCmp = case xCmp of
           Just cmp -> Just (\(k1:>v1) (k2:>v2) -> k1 `cmp` k2)
           Nothing  -> Nothing
  ...similarly the other operations...
```

Here, (k :> v) is a key-value pair, read "k maps to v". Comparison of a key-value pair is done solely on the basis of the key. It is crucial that we use a new data type for key-value pairs, rather than using the built-in pair constructor, because the latter has equality and ordering instances that look at both components of the pair, not just the first.

A Set of key-value pairs, with comparison done on this basis, is a finite map. All that is needed to complete the picture is to add some crucial functions to the Collection class:

```
class Collection c where
  ...as before...
  insertWith  :: (XOps a) => (a->a->a) -> a -> c a -> c a
  unionWith   :: (XOps a) => (a->a->a) -> c a -> c a -> c a
  lookup      :: (XOps k) => k -> c (Pr k v) -> Maybe v
```

The "With" variants have a function that combines values that compare as equal when doing insertion or union. This is very important when those values are equal because they have equal keys, but we might wish (for example) to add the second component of the pairs.

A disadvantage of this approach is that every instance of Collection must, in principle, provide an implementation of lookup. While doing so is always possible — indeed one could write a default declaration for lookup using fold — it is not desirable because for many instances of Collection a lookup might be wildly inefficient and inappropriate.

3.7 The complete class

Figures 4 and 5 give the complete definition of the collections module. There are several points to note:

- The type of elem is a bit more specific than the default method requires. Again, this is to allow an implementor to make a more efficient elem that exploits the ordering on elements.
- If the representation of a non-empty collection always included the necessary comparison operations (see item (1) in Section 3.4), it would be possible to give many operations a rather simpler type, by omitting the (XOps a) context. Doing so would place more constraints on the implementor, so we have refrained from doing so.
- toList is a pretty dodgy looking operation because (:) is not left-commutative. Nevertheless, lists are so ubiquitous (albeit perhaps less so once these libraries are in place!) that it may be more convenient to use toList followed by a list operation rather than a single more respectable fold. The final result may (indeed should) still be independent of the order in which the fold chose to lay out the collection.

4 Second design: multi-parameter constructor classes

Our first design was written in standard Haskell, but it has three fundamental deficiencies:

- It defers to run time some checks that one might intuitively expect to be statically checked.
- It separates sequences and collections entirely, whereas one might have expected that they would share common operations.
- It does not separate (say) lists, from FIFOs, from deques. These are all in class Sequence and provide the same operations, but one might prefer the type system to express the idea that FIFOs have more operations than lists, and deques than FIFOs.

Our second design, which we give in much less detail than the first, overcomes both these objections, but at the expense of stepping outside standard Haskell by using *multi-parameter* constructor classes. In the view of the author, the clean way that multi-parameter constructor classes turn out to accommodate bulk types is a very persuasive reason for extending Haskell to embrace them, just as monads provide the key motivation for adding constructor classes.

4.1 The key idea

The key idea is very simple. Suppose we (re-)define the class of collections like this:

```
class Collection c a where
  size   :: c a -> Int
  empty  :: c a
  cons   :: a -> c a -> c a
  union  :: c a -> c a -> c a
  fold   :: (a->b->b) -> b -> c a -> b
  filter    :: (a->Bool) -> c a -> c a
  partition :: (a->Bool) -> c a -> (c a, c a)
```

Notice that Collection has two parameters: c, the type constructor of he collection, and a, the element type. Notice too that insert has no XOps constraint. The type of cons, for example, is now:

```
cons :: Collection c a => a -> c a -> c a
```

The interesting part comes when we define instances of Collection:

```
instance Collection [] a where
  empty  = []
  insert = (:)
  ...and so on...

instance Ord a => Collection OrdSeq a where
  empty  = emptyTree
  insert = insertTree
  ...and so on...
```

The exciting thing is that now *we can provide instance-specific constraints on the element type.* In the first instance declaration, for lists, no constraints are placed on a, so insert can be used on lists without placing any constraints on the element type. In contrast, the second instance declaration specifies that the element type a must be in class Ord, just what is needed to allow the use of insertTree (here assumed to have type Ord a => a -> Tree a -> Tree a) to define insert.

This simple extension solves at a stroke both of the deficiencies of our first design:

- Things that "should" be checked statically are checked statically. In particular, an attempt to use an OrdSet with an element type that has no ordering will provoke an error at compile time rather than at run time.
- The same class embraces both collections with constraints on the elements, and collections with none (termed sequences of the first design). There is no need for both a filter on sequences and a separate filter on collections; plain filter will work on both.

It remains possible to have adaptive representations for collections, using the same XOps class as before, thus:

```
instance XOps a => Collection Set a where
  ...as before...
```

This instance declaration makes clear that the type Set of adaptive sets requires its element type to be in class XOps. The implementation can now be given exactly as before.

Notice that MonadZero and Functor are not superclasses of Collection, as they were of Sequence, because not all instances of Collection could be instances of Monad since the latter requires operations polymorphic in the elements. We can still make *particular* bulk types (the polymorphic ones) instances of Monad, of course, by giving a suitable instance declaration, so we are not giving up the possibility of using monad comprehensions to create and filter collections.

4.2 Using the class hierarchy

It seems obvious that sequences should have all the operations that unordered collections have, and some more besides. Now that the operations in Collection apply to sequences as well as unordered collections, we can use the class hierarchy to express precisely the inheritance we want:

```
class (Collection s a) => Sequence s a where
  snoc  :: s a -> a -> s a
  first :: s a -> SeqView s a
  last  :: s a -> SeqView s a
  foldl :: (b->a->b) -> b -> s a -> b
  reverse :: s a -> s a
```

There are now quite a few design decisions to make. For example:

- Does one want one class that supports front but not back, another that supports back but not front, and a third that combines these capabilities? Or is it best to have one class (such as the `Sequence` just defined above) that has both. After all, one can get the last element of a list — it's just rather inefficient to do so.

- Should `cons` (implying "add an element to the front" for sequences, and just "add an element" for unordered collections) be in `Collection`, and `snoc` ("add an element to the back") in `Sequence`, or should both be in `Sequence`, with some other subclass of `Collection` having a neutral `insert` for unordered collections?
- Similar questions arise for `fold` and its directional cousins `foldr` and `foldl`.
- Should every collection support `union` when for some it may be a constant time operation while for others it is an O(N) operation?

The answers to these questions are not obvious, but the the collection classes of Smalltalk and C++ provide a good deal of guidance. For example, Smalltalk's collection-class hierarchy looks like this:

```
Collection
  Bag
  Set
    Dictionary
  Sequencable collection
    Interval
    LinkedList
    OrderedCollection
      SortedCollection
    ArrayedCollection
      Array
```

4.3 Finite maps

Finite maps can be still handled exactly as described in Section 3.6, but multi-parameter type classes opens up another intriguing possibility:

```
class Collection (c k) a => FM c k a where
  extend :: k -> a -> c k a -> c k a
  lookup :: c k a -> c -> k -> a
```

This declares the three-parameter type class `FM`, parameterised over `c`, the type constructor of the map, `k`, the key type, and `a`, the value type. It requires that the partial application of `c` to `k` is a collection type constructor. Now all the collection operations work on finite maps, but the latter add two new operations, `extend` and `(!!)` (i.e. lookup).

One advantage of this approach is that it makes it possible to include Haskell's standard arrays in `FM` — which is nice, because arrays are plainly finite maps:

```
instance Ix k => FM Array k a where
  lookup = (!!)
```

Of course, Haskell arrays don't support `extend` or any of the operations in `Collection`, so one might change the hierarchy to look like this:

```
class Indexable c k a where
  lookup :: c k a -> c -> k -> a

class (Collection (c k) a, Indexable c k a) => FM c k a where
  extend :: k -> a -> c k a -> c k a
```

Again, there are many possible design choices.

4.4 Summary

Multi-parameter constructor classes seem to be just what is needed to make a clean job of bulk types. What we have done here is only to sketch the basic idea. A considerable amount of design work remains to flesh it out into a concrete design, even assuming the existence of multi-parameter constructor classes.

5 Related work

There is a large literature on collection types, also known as *bulk types*. Tannen [1994] gives a useful bibliography, from a database perspective. Buneman et al. [1995] explores the algebra and expressiveness of algebras based on (`empty`, `insert`) and (`empty`, `singleton`, `union`).

C++ has a well-developed library called the Standard Template Library (STL) which is specifically aimed at collection types (Stepanov & Lee [1994]). There are major differences from the work described here. Rather than a collection being a value which can be combined with other similar values, it is regarded as a container into which new values can be placed. There is no equivalent of `fold`; instead *iterators* are provided, which specify a location within a container. It does handle polymorphism, however, using C++ templates; when a collection is declared one specifies both the element type and the comparison operation to use.

Parametric type classes (Chen, Hudak & Odersky [1992]) have similar power to multi-parameter constructor classes. Indeed Chen's thesis uses bulk types as the main motivating example for parametric type classes (Chen [1994]).

6 Summary

Designing suitable signatures for bulk types is surprisingly tricky. The number of different kinds of collection, and the number of possible implementations of each kind of collection, makes it rather unattractive to use distinct names for the operations of each. Furthermore, if we do so we cannot write polymorpic algorithms; that is, algorithms that work regardless of which kind of collection is involved.

The first design proposed here exploits type classes to obtain a substantial amount of polymorphism. Algorithms can be polymorphic over the elements of the collection, the implementation of the collection, and the nature of the collection.

Apart from the use of type classes, the two key design decisions are these:

- At first sight it seems attractive to unify all bulk types into a single class. We propose instead to use two classes, one for sequences and one for collections. Sequences are parametric in their element type, and are sensitive to insertion order, while the reverse holds for collections.
- We solve the typing problems of collections with the `XOps` class, thereby requiring a small amount of run-time type-checking (at least when the types are not statically known). Whilst it is not perfect, this can be turned to our advantage by allowing the programmer to design data types that choose their representation based on the operations available for the element type.

The second design uses multi-parameter type classes to unify sequences and unordered collections into a single class hierarchy. It seems to be a noticeably cleaner solution, but requires a signficant extension to Haskell.

An attractive property of both designs is the possibility of writing adaptable implementations, that automatically choose their representation based on the operations available on the underlying data type.

Acknowledgements

I would like to thank Peter Buneman, Laszlo Nemeth, and David Watt, for their helpful comments on earlier drafts of this paper. Mark Jones and Peter Thiemann both pointed out to me the beautiful fit between multi-parameter constructor classes and bulk types that is sketched in Section 4. They both also gave me particularly useful feedback on other aspects of the paper.

I gratefully acknowledge the support of the Oregon Graduate Institute during my sabbatical, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References

GS Brodal & C Okasaki [Dec 1996], "Optimal purely-functional priority queues," *Journal of Functional Programming* 6.

P Buneman, L Libkin, D Suciu, V Tannen & L Wong [March 1994], "Comprehension syntax," in *SIGMOD Record* #23 #1, 87–96.

P Buneman, S Naqvi, V Tannen & L Wong [Sept 1995], "Principles of programming with complex objects and collection types," *Theoretical Computer Science* 149.

K Chen [1994], "A parametric extension of Haskell's type classes," PhD thesis, Department of Computer Science, Yale University.

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

T-R Chuang & WL Hwang [May 1996], "A probabilistic approach to the problem of automatic selection of data representations," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, 190–200.

T Freeman & F Pfenning [June 1991], "Refinement types for ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'91), Toronto*, ACM, 268–277.

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

C Okasaki [May 1996], "The role of lazy evaluation in amortized data structures," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, 62–72.

C Okasaki [Oct 1995], "Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking," in *IEEE Symposium on Foundations of Computer Science*, 646–654.

A Stepanov & M Lee [Dec 1994], "The Standard Template Library," Hewlett-Packard Laboratories, Palo Alto.

V Tannen [1994], "Tutorial: languages for collection types," University of Pennsylvania.

PW Trinder [Aug 1991], "Comprehensions: a query notation for DBPLs," in *Proc 3rd International Workshop on Database Programming Languages, Nahplion, Greece*, Morgan Kaufman, 49–62.

PW Trinder & PL Wadler [1989], "List comprehensions and the relational calculus," in *Functional Programming, Glasgow 1988*, Workshops in Computing, Springer Verlag, 115–123.

PL Wadler [1992], "Comprehending monads," *Mathematical Structures in Computer Science* 2, 461–493.

Disposable Memo Functions*

Byron Cook John Launchbury
byron@cse.ogi.edu jl@cse.ogi.edu

Department of Computer Science and Engineering
Oregon Graduate Institute

Abstract

We formalize the meaning of lazy memo-functions in Haskell with an extension to the lazy λ-calculus, Haskell's computational model. The semantics enable reasoning about memoization's effect on space and time complexity. Based on the semantics, we present a prototype implementation that requires no changes to the garbage-collector: memo-tables are simply reclaimed when no references to them remain.

1 Introduction

A *memo-function* remembers the arguments to which it has been applied, together with the result. If applied to a repeated argument, memo-functions return the cached answer rather than recomputing it from scratch. Memoization improves the time complexity of algorithms with repeated computations — but can consume vast amounts of memory. Some implementations of memoization use heuristic cache replacement policies (such as LRU or FIFO) to manage memory. Although implementations try to minimize space consumption, in their essence, memo-functions trade better runtime performance for worse space behavior.

Programmers are accustomed to these sorts of trade-offs: "Should I cache this value or is it too big to keep around?" Many programmers develop principles on which to base such decisions. But when memo-functions use heuristic purging, the old reasoning principles may no longer apply. Values might be removed from the memo-table if they have a certain type, or size, or are not used frequently enough. To make matters worse, the cache replacement policies make assumptions about evaluation strategy. However, there is one form of purging known to work well in Haskell: garbage-collection.

Can memoization be integrated into Haskell such that memo-tables are managed like other values in the heap? We show that it can. What's more, we give a formal semantics for memoization that is compatible with Haskell's underlying computational model, and where the garbage-collection rule can reclaim obsolete memo-functions and the space their tables consumed.

The key idea in this paper is to adapt Hughes' research on lazy memoization[7] and provide a polymorphic function:

```
memo :: Eval a => (a -> b) -> (a -> b)
```

*The authors are supported by a contract with Air Force Material Command (F19628-93-C-0069).

When applied to a function `memo` returns an equivalent memoized function. When all references to this new function have been dropped, the garbage-collector is able to reclaim the function and its table. It is in this sense that memo functions are disposable.

To provide a concrete example of using `memo`, consider applying the factorial function to each element in a list.

```
map fact [17,8,17,17,17,8]
```

Clearly, recomputing `(fact 17)` four times is inefficient. This can be remedied by mapping a memoized version of `fact` down the list:

```
map (memo fact) [17,8,17,17,17,8]
```

Once the expression has been reduced, no references to the memoized `fact` will remain. The garbage collector can then reclaim it and its memo-table. Of course, in this trivial example we could have explicitly cached the repeated computations with a `let` but, as we will demonstrate later, this technique doesn't scale to larger programs.

We have developed a prototype implementation of the semantics, appropriately named Huggies, by extending the Hugs Haskell interpreter. Huggies demonstrates that, in theory, there need be no direct connection between the garbage-collector and `memo`, and our implementation required no changes to the Hugs garbage-collector (we discuss compacting collectors later). Using heap profiles, we will show that Huggies can garbage-collect disposable memo-functions.

2 Applications of Memoization

In this section we demonstrate the utility of memo-functions by addressing two problems in areas of recent research. Both examples involve specification languages embedded in Haskell. Although specifications are concisely expressed in pure functional languages, the resulting programs often contain computational redundancies. The programs typically must be modified to enhance their computational content. However, the very changes that improve efficiency also obscure structure and degrade maintainability.

2.1 Parsers

If interpreted directly with recursive descent, presentations of grammars often contain repeated computations. Typically an efficient parser is based on a transformed grammar. In some cases, rather than redesigning the grammar, a memoized parser can be about as fast. As a simple example, consider the grammar for arithmetic expressions:

$$\begin{array}{lcl} \mathit{term} & ::= & \mathit{factor}\ +\mathit{factor} \\ & \mid & \mathit{factor}\ -\mathit{factor} \\ & \mid & \mathit{factor} \\ \mathit{factor} & ::= & \mathit{expr}\ \ *\ \mathit{expr} \\ & \mid & \mathit{expr}\ \ /\ \mathit{expr} \\ & \mid & \mathit{expr} \\ \mathit{expr} & ::= & \mathit{number} \\ & \mid & (\mathit{term}) \end{array}$$

Using Hutton and Meijer's parser combinators[8] the corresponding naive parser is:

```
naiveTerm =   binary factor '*' factor
          +++ binary factor '/' factor
          +++ factor
     where
     factor =   binary expr '+' expr
            +++ binary expr '-' expr
            +++ term
     expr = num +++ bracketed naiveTerm
```

Unfortunately, naiveTerm is slow. When parsing the expression "(((1-2)-2)-3)", naiveTerm first traverses "((1-2)-2)" before discovering that the next character is not a '*'. It throws away the parse and begins anew only to repeat the process several more times. Of course, the parser and grammar could be restructured to explicitly cache intermediate values, but a simpler transformation would be to memoize factor and expr:

```
term () = trm
     where
     trm =   binary factor '*' factor
         +++ binary factor '/' factor
         +++ factor
     factor = memoParser (
             binary expr '+' expr
         +++ binary expr '-' expr
         +++ term )
     expr = memoParser (num +++ bracketed trm)

memoParser (Parser f) = Parser (memo f)
```

The memoized parser, while perhaps a bit slower than the restructured parser, has reasonable time complexity. More importantly, the program remains maintainable because the transformation has preserved the original structure.

Notice that term is not itself a memo-function. When applied to (), it returns a parser which builds local memo-functions. While parsing, the local memo-tables are bounded by the size of the input string. Once the string is parsed, (term ()) becomes garbage, along with any memo-functions.

It is tempting to define the parser at the top-level:

```
term' = term ()
```

However, term' will build its local memo-functions only once. In Huggies, term' will persist throughout the lifetime of the program. The more term' is used, the more heap it will consume.

2.2 Animation Combinators

Reactive Behavior Modeling in Haskell[3] (RBMH) is a language in development at Microsoft for describing multimedia interactive animation. RBMH is a suite of combinators and simple behaviors from which programmers can build complex behaviors. The abstraction of combinators and behaviors can lead to programs with repeated computations — especially since behaviors are functions on time:

```
data Behavior a = B(Time -> (a, Behavior a))
```

In RBMH, references to behaviors are easily duplicated, resulting in redundant sampling. For example, (+) is overloaded on behaviors such that, when b and c are behaviors, the expression (b + c) reduces to:

```
B(\t -> let (x,b') = at t b
            (y,c') = at t c
        in (x + y, b' + c'))
```

where at returns the value of b at time t. However, b + b does not reduce to:

```
B(\t -> let (x,b') = at t b
        in (x + x, b' + b'))
```

as you might hope. Instead, (at t b) is computed twice:

```
B(\t -> let (x,b') = at t b
            (y,c') = at t b
        in (x + y, b' + c'))
```

The overloaded (+) could remove this redundancy by memoizing (at t):

```
B(\t -> let at' = memo (at t)
            (x,b') = at' b
            (y,c') = at' c
        in (x + y, b' + c'))
```

Is at' disposable? Yes; once x, y, b', and c' are computed no references to at' will remain.

3 Semantics

In this section, we provide an operational semantics for memoization in Haskell by extending the lazy λ-calculus. The power of the operational semantics is that it gives neither a trivial denotational meaning, nor a meaning based on stack pointers, program counters, and jumps. The semantics captures the essence of memoization at the right level of abstraction; it is detailed enough to be useful, and simple enough that it does not obscure meaning.

3.1 Lazy Semantics

The syntax of the source language is defined as:

$$\begin{array}{rcll} e & \in & \mathit{Expns} & ::= x \mid v \mid c \mid e\ a \mid \\ & & & \quad \texttt{let}\ \{x_1 = e_1; \ldots\}\ \text{in}\ e \\ a & \in & \mathit{Atoms} & ::= x \mid n \\ v & \in & \mathit{Values} & ::= n \mid \lambda x.e \\ c & \in & \mathit{Constants} & ::= + \mid - \mid \ldots \\ n, m, p & \in & \mathit{Numbers} & ::= 1 \mid 2 \mid \ldots \\ \Gamma, \Delta & \in & \mathit{Heaps} & ::= \Gamma, x \mapsto e \mid \varnothing \\ f, g, x, y, z, t & \in & \mathit{Variables} & \\ \rho & \in & \mathit{Envs} & ::= \{\ldots (x, y) \ldots\} \mid \varnothing \end{array}$$

3.2 Memo Semantics

Terms in the semantics are formed of pairs $\langle\, e \mid \Gamma \,\rangle$, where e is an expression and Γ is a heap. The semantic rules of the lazy λ-calculus are given in Figures 1 and 2 as a relation $\Longrightarrow$ and $\longrightarrow$ between terms. The semantics contain the following meta-operations:

$e[y/x]$: substitution of y for occurrences of x in e.

$\hat{v}$: α-renaming with fresh variables

$$\langle\ \texttt{memo}\ f \mid \Gamma\ \rangle \longrightarrow \langle\ \lambda x.\ \mathsf{access}\ t\ x \mid \Gamma,\ t \mapsto (f, \varnothing)\ \rangle \qquad (\mathit{memo})$$

$$\frac{\langle\ e \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ \mathsf{access}\ t\ x \mid \Gamma,\ x \mapsto e\ \rangle \longrightarrow \langle\ \mathsf{access}\ t\ x \mid \Delta,\ x \mapsto e'\ \rangle} \qquad (\mathit{access}^e)$$

if e is not a value

$$\langle\ \mathsf{access}\ t\ x \mid \Gamma,\ x \mapsto n\ \rangle \longrightarrow \langle\ \mathsf{access}\ t\ n \mid \Gamma,\ x \mapsto n\ \rangle \qquad (\mathit{access}^n)$$

$$\langle\ \mathsf{access}\ t\ n \mid \Gamma,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ \rho(n) \mid \Gamma,\ t \mapsto (f, \rho)\ \rangle \qquad (\mathit{access}^{\in n})$$

if $n \in \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ n \mid \Gamma,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ z \mid \Gamma,\ t \mapsto (f, \rho \cup \{(n, z)\}),\ z \mapsto f\ n\ \rangle \qquad (\mathit{access}^{\notin n})$$

if $n \notin \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ x \mid \Gamma,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ \rho(x) \mid \Gamma,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \qquad (\mathit{access}^{\in \lambda})$$

if $x \in \mathit{dom}\ \rho$

$$\langle\ \mathsf{access}\ t\ x \mid \Gamma,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho)\ \rangle \longrightarrow \langle\ z \mid \Gamma,\ x \mapsto \lambda w.e,\ t \mapsto (f, \rho \cup \{(x, z)\}),\ z \mapsto f\ x\ \rangle \qquad (\mathit{access}^{\notin \lambda})$$

if $x \notin \mathit{dom}\ \rho$

Figure 3: Semantics of Memo

$$\langle\ (\lambda x.e)\ y \mid \Gamma\ \rangle \longrightarrow \langle\ e[y/x] \mid \Gamma\ \rangle \qquad (\mathit{app}^\beta)$$

$$\frac{\langle\ e \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ e\ y \mid \Gamma\ \rangle \longrightarrow \langle\ e'\ y \mid \Delta\ \rangle} \qquad (\mathit{app}^e)$$

$$\langle\ y \mid \Gamma,\ y \mapsto v\ \rangle \longrightarrow \langle\ \hat{v} \mid \Gamma,\ y \mapsto v\ \rangle \qquad (\mathit{var}^v)$$

where $\hat{v}$ is v with all bound variables renamed to fresh names

$$\frac{\langle\ e \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ y \mid \Gamma,\ y \mapsto e\ \rangle \longrightarrow \langle\ y \mid \Delta,\ y \mapsto e'\ \rangle} \qquad (\mathit{var}^e)$$

$$\langle\ \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e' \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Gamma,\ x \mapsto e\ \rangle \qquad (\mathit{let})$$

$$\langle\ n\ \texttt{+}\ m \mid \Gamma\ \rangle \longrightarrow \langle\ p \mid \Gamma\ \rangle \qquad (\mathit{plus}^n)$$

where $p = m + n$

$$\frac{\langle\ e \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ n\ \texttt{+}\ e \mid \Gamma\ \rangle \longrightarrow \langle\ n\ \texttt{+}\ e' \mid \Delta\ \rangle} \qquad (\mathit{plus}^l)$$

$$\frac{\langle\ e_1 \mid \Gamma\ \rangle \longrightarrow \langle\ e_1' \mid \Delta\ \rangle}{\langle\ e_1\ \texttt{+}\ e_2 \mid \Gamma\ \rangle \longrightarrow \langle\ e_1'\ \texttt{+}\ e_2 \mid \Delta\ \rangle} \qquad (\mathit{plus}^r)$$

Figure 1: Lazy λ-calculus Semantics

The lazy λ-calculus models Haskell's evaluation strategy by sharing computations and halting reduction when outside λs are encountered. The sharing is achieved through two constraints on reduction:

$$\langle\ e \mid \Gamma,\ x \mapsto e'\ \rangle \Longrightarrow \langle\ e \mid \Gamma\ \rangle \qquad (\mathit{gc})$$

if x is not reachable from e

$$\frac{\langle\ e \mid \Gamma\ \rangle \longrightarrow \langle\ e' \mid \Delta\ \rangle}{\langle\ e \mid \Gamma\ \rangle \Longrightarrow \langle\ e' \mid \Delta\ \rangle} \qquad (\mathit{reduce})$$

Figure 2: Lazy λ-calculus with Garbage-collection

- Functions can only be applied to atomic values like integers and variables. This prevents arbitrary expressions being substituted into function bodies. For example, the reduction $(\lambda x.x + x)(3 + 3) \longrightarrow (3 + 3) + (3 + 3)$ is not allowed because it would duplicate the computation $3 + 3$.
- Before replacing a variable with its heap value, the value must be in weak head normal form. For example, the expression $3 + 3$ must be reduced in the heap first before substituting it for x: $\langle\ x + x \mid x \mapsto 3 + 3\ \rangle \longrightarrow \langle\ x + x \mid x \mapsto 6\ \rangle \longrightarrow \langle\ 6 + 6 \mid x \mapsto 6\ \rangle$

We maintain the invariant that all binding sites bind distinct names. Whenever a term is suplicated (in a (var^a) rule) we α-rename with fresh names.

To define memoization we extend the λ-calculus with the constant **memo** and language construct access:

$$\begin{array}{lll} c \in \mathit{Constants} & ::= \ldots & \mid\ \texttt{memo} \\ e \in \mathit{Expns} & ::= \ldots & \mid\ \mathsf{access}\ t\ x \end{array}$$

Initial terms should not contain access expressions — access should only be introduced by reduction.

Figure 3 extends the semantics with rules for **memo** and access. The semantics use the additional meta-operations:

$\mathbf{x} \in \mathbf{X}$: set membership. The sets contain mixtures of variable names and numbers (i.e. atomic values).

$\rho(x)$: the corresponding value for x in the memo-table ρ

$\rho \cup \{(x, y)\}$: extension of the memo-table ρ with the tuple (x, y); x may not appear in the domain of ρ

The basic point is that if a memo-function is applied to an argument already in the table, then the already-computed value is returned. If the function is applied to a new value then it is placed into the table along with a reference to the answer.

When `memo` is applied to a variable y (and notice that `memo` can only be applied to variables) the computation is replaced by ($\lambda x.$access t x), where t is a reference to the tuple $(y, \varnothing)$ in the heap. When an `access` expression is reduced, the memo-table is potentially updated.

The behavior of memo-functions depends critically on the sharing of the lazy λ-calculus. Performance is lost if reductions can duplicate expressions containing applications of `memo`. For example, the expression $(\lambda f.f\ 5 + f\ 5)$(`memo` g) should evaluate to (`access` t 5 + `access` t 5) and not ((`memo` g) t 5 + (`memo` g)t 5).

We are now able to answer this paper's thesis. We have defined memo-tables as local variables inside of memo-functions. Therefore, when a memo-function becomes garbage the corresponding memo-table should be garbage too. Let $E[\bullet]$ be an expression containing $\bullet$. If $\langle\ E[$`let` f = `memo` g `in` $e'] \mid \Gamma\ \rangle \Longrightarrow \langle e \mid \Delta,\ f \mapsto \lambda x.$access t x, $t \mapsto (f, \rho)\ \rangle$ and ($\lambda x.$access t x) is not reachable from e, then $\langle\ e \mid \Delta,\ f \mapsto \lambda x.$access t x, $t \mapsto (f, \rho)\ \rangle \Longrightarrow \langle\ e \mid \Delta\ \rangle$. Because *memo* is the only rule introducing references to t, and the access rules do not duplicate references to t, then all references to t are in the form `access` t x. Therefore, if `access` t x is not reachable from e, then t is not reachable from e: and both f and t can be removed from the heap by an application of (gc).

In this paper we do not explore the properties of the extended semantics. A proof showing that `memo` f is denotationally equivalent to a strict f would be a very strong result — and one we would like to pursue.

It might also be fruitful to abstract the semantics of `memo` over the computational model. By specifying the requirements of the underlying model, rather than relying on the particular properties of the lazy λ-calculus, the meaning of memoization could be applicable in many settings. For example, can the semantics be adapted to SML, or a strictness neutral language like Henk?

3.3 Example Reduction

Figure 4 provides a top-level reduction sequence for the term $\langle$`let` g = `memo` f `in` g 5 + g 5 $\mid \Gamma\ \rangle$ The arrows are annotated with the rules of the sub-reduction used to establish each reduction. The (gc) step is important. After the expression has been reduced to $\langle\ 16 + z \mid \ldots \rangle$ any extra heap space is thrown away. We assume $f\ 5\ =\ 16$ for concreteness.

4 Implementation

Huggies provides a proof-of-concept, and is far from optimal. Huggies clearly implements our semantics, and is a starting point from which future refinements can be based.

The function `memo` is written in Haskell as the composition of several non-standard primitives. The primitives represent the pieces of the semantics that are impure.

4.1 Memoizing in Standard Haskell

Before extending the language with `memo`, it is useful to see how we might memoize functions in standard Haskell. Using the example in Section 1 we can build a new function, `fastMap`, that uses a memo-function while traversing the list and assumes that the list contains integers between 0 and 96:

```
fastMap :: (Int -> b) -> [Int] -> [b]
fastMap f list = runST (
    do g <- memoST f
       mapM g list
    )

memoST :: (a -> b) -> ST s (a -> ST s b)
memoST f =
  do t <- newArray (0,97) Nothing
     return (\x -> accessST t x)
  where
  accessST t x =
    do let w = x `mod` 97
       a <- readArray t w
       case a of
         Nothing -> do let z = f x
                       writeArray t w (Just z)
                       return z
         Just y -> return y
```

The example can now be re-written as:

```
fastMap fact [17,8,17,17,17,8]
```

In this expression, (`fact 17`) is calculated only once. What's more, the expression is purely functional.

4.2 Non-standard Primitives

4.2.1 Memo-tables

Fast memoization depends critically on an efficient implementation of environment lookup ($\in$ in the semantics). However, Haskell's purity stands in the way of fast polymorphic environments written within the language. Therefore, we have added the environment type `Env` to Huggies:

```
newEnv     :: (a -> a -> ST s Bool)
                  -> ST Mem (Env s a b)
readEnv    :: Env s a b -> a -> ST Mem (Maybe b)
writeEnv   :: Env s a b -> a -> b -> ST Mem ()
accessEnv  :: Env s a b -> a -> b -> ST Mem b
```

The underlying implementation is a hash-table where we hash on based the value of integers, booleans, characters and floating point numbers, and the location of other values. Because environments depend on the state of global memory, `ST`'s first parameter is instantiated to the constant `Mem`.

The function `newEnv`, when passed a stateful equality function, returns a new environment. Subsequent reads and writes to the environment use the parameterized equality function to determine where in the environment the read or write should be performed.

The function `accessEnv` is an efficient composition of `readEnv` and `writeEnv`: but it can be thought of as:

```
accessEnv e x y =
  do r <- readEnv e x
     case r of
       Nothing -> do writeEnv e x y
                     return y
       Just y' -> return y'
```

Notice that our implementation of `Env` and the garbage-collector are tightly coupled — environments would have to be rebuilt after each garbage-collection if Hugs compacted memory. But the interaction between `Env` and the garbage-collector is not visable from the defintion of `memo`.

$$
\begin{array}{ll}
 & \langle\ \texttt{let}\ g = \texttt{memo}\ f\ \texttt{in}\ g\ 5 + g\ 5 \mid \Gamma\ \rangle \\
\overset{let}{\Longrightarrow} & \langle\ g\ 5 + g\ 5 \mid \Gamma,\ g \mapsto \texttt{memo}\ f \rangle \\
\overset{plus^{l}}{\Longrightarrow}{,app^{e},var^{a},memo} & \langle\ g\ 5 + g\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\overset{plus^{l}}{\Longrightarrow}{,app^{e},var^{v}} & \langle\ (\lambda x.\texttt{access}\ t\ x)\ 5 + g\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\overset{plus^{l}}{\Longrightarrow}{,app^{\beta}} & \langle\ \texttt{access}\ t\ 5 + g\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \varnothing)\ \rangle \\
\overset{plus^{l}}{\Longrightarrow}{,access^{\notin n}} & \langle\ z\ + g\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto f\ 5\ \rangle \\
\overset{plus^{l}}{\Longrightarrow}{,var^{v}} & \langle\ 16\ + g\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\overset{plus^{r}}{\Longrightarrow}{,app^{e},var^{v}} & \langle\ 16\ +\ (\lambda x.\texttt{access}\ t\ x)\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\overset{plus^{r}}{\Longrightarrow}{,app^{\beta}} & \langle\ 16\ +\ \texttt{access}\ t\ 5 \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\overset{plus^{r}}{\Longrightarrow}{.access^{\in n}} & \langle\ 16\ +\ z \mid \Gamma,\ g \mapsto \lambda x.\texttt{access}\ t\ x,\ t \mapsto (f, \{(5, z)\}),\ z \mapsto 16\ \rangle \\
\overset{gc}{\Longrightarrow} & \langle\ 16\ +\ z \mid \Gamma,\ z \mapsto 16\ \rangle \\
\overset{plus^{r}}{\Longrightarrow}{,var^{v}} & \langle\ 16\ +\ 16 \mid \Gamma,\ z \mapsto 16\ \rangle \\
\overset{plus^{n}}{\Longrightarrow} & \langle\ 32 \mid \Gamma,\ z \mapsto 16\ \rangle
\end{array}
$$

Figure 4: Example Reduction

4.2.2 Equality

As described in Section 1, lazy memo-functions use a different notion of equality depending on the type of the arguments. The primitive function eql implements this behavior:

```
eql :: (Eval a,Eval b) => a -> b -> ST Mem Bool
```

Notice that eql accepts arguments of different types. We are in a precarious position with the type system. By using eql rather than (==) we have avoided a subtle error that could cause programs to crash. To see why this is true, imagine memoizing the identity function with traditional memoization:

```
memoEq :: Eq a => (a -> b) -> (a -> b)

let f = memoEq id
in (f 5, f "hello")
```

If (f 5) is reduced first then the subsequent reduction of (f "hello") might cause access to compare 5 and "hello". Fortunately, eql simply returns False, but (==) is not so forgiving.

4.2.3 Unsafe State Monad Escape Operator

The final primitive added to Huggies has the same behavior as runST without the restrictive type:

```
unsafeST :: ST s a -> a
```

This new primitive has to be used with extreme caution. Whereas runST will only encapsulate referentially transparent functions, unsafeST makes no such distinction. This allows the state of a memo-function to flow implicitly throughout a program's execution. In the case of `memo` this effect is not observable, but a misuse of unsafeST could easily subvert Haskell's purity.

4.3 Defining `memo`

The functions `memo` and access are thus defined:

```
memo :: Eval a => (a -> b) -> (a -> b)
memo f = \x -> access t x
     where
     t = (f,emptyEnv)
     emptyEnv = unsafeST (newEnv eql)

access :: Eval a => (a -> b, Env Mem a b)
              -> (a -> b)
access (f,t) x = unsafeST (accessEnv t x (f x))
```

4.4 Recursion

Memoizing syntactically recursive functions with `memo` is clumsy:

```
fib = memo mfib
     where
     mfib 0 = 1
mfib 1 = 1
mfib n = fib (n-1) + fib (n-2)
```

That fib has the correct behavior may be seen by an application of the semantic rules, but it is far from intuitive. However, functions that are defined as the fixed point of functionals can be memoized with memoFix:

```
memoFix f = let g = f h
                h = memo g
            in h
```

Recursive functions, such as Fibonacci, can be written and then memoized as such:

```
fib m 0 = 1
fib m 1 = 1
fib m n = m (n-1) + m (n-2)

memofib = memoFix fib
```

4.5 Dangers of unsafeST

Although unsafeST makes the definition of memo possible, we are wary of its robustness and portability. Programs written with unsafeST can exhibit surprising behaviors. The state monad and runST were carefully designed to avoid functions like unsafeST — and for good reason. Hidden updatable state endangers referential transparency. As an example, we ask the question: does bad () equal bad ()? Not when defined as follows:

```
bad :: () -> Bool
bad = unsafeST (
  do v <- newVar True
     return (\x -> unsafeST (toggle v x))
  )


toggle v () =
  do x <- readVar v
     case x of
        True  -> do () <- writeVar v False
                    return False
        False -> do () <- writeVar v True
                    return True
```

After each application the value of bad () toggles between True and False. Much like bad, memoized functions contain a mutable variable that is updated during applications; however, because memo was carefully written, memoized functions should preserve equational reasoning. The values that result in expressions containing unsafeST, therefore, come with proof obligations to guarantee their safe behavior.

Notice that there is subtle interaction between unsafeST and lazy state in the definition of toggle. If toggle were re-written as

```
toggle v () =
  do x <- readVar v
     case x of
        True  -> do writeVar v False
                    return False
        False -> do writeVar v True
                    return True
```

then the mutable variable is no longer updated. In this case the return value of toggle does not depend on the application of writeVar and it is simply never performed.

5 Profiling Huggies

We have preliminary evidence that the Huggies garbage-collector reclaims the memo-tables of disposable memo-functions. Figure 5 contains heap profiles annotated with the expressions that were executed while generating them. The expressions on the left use disposable memo functions. The expressions on the right use non-disposable memo-functions. Notice that heap consumption returns to zero in the profiles on the left — indicating that the garbage-collector successfully reclaimed the space.

6 Related Work

6.1 Lazy Memo-functions

Rather than defining memoization, Hughes[7] focused on the applications of lazy memo-functions and the implementation issues of obsolescence-based purging optimizations

Hughes proposed that memo-functions be defined with a language construct rather than a higher-order function. In his syntax, the keyword memo was placed in front of the memo-function's definition.

```
memo fib 0 = 1
memo fib 1 = 1
memo fib n = fib (n-1) + fib (n-2)
```

It is easier to define recursive memo-functions in this notation, but clumsy to use when developing a semantics.

The semantics presented in this paper are compatible with Hughes's original work. In fact, we have clarified issues originally raised in Hughes's paper. For example, Hughes states the following definition of map creates a local memo-function that can be garbage-collected after map has been applied:

```
map f l  = m l
 where memo m []     = []
       memo m (x:xs) = f x: m xs
```

However, the paper does not develop principles for reasoning about when a memo-function can be garbage-collected. With our semantics (extended with lists), it could be verified that m is disposable once translated into the appropriate form:

```
map f l  = memoFix m l
 where m g []     = []
       m g (x:xs) = f x: g xs
```

6.2 Memoization in LISP and SML

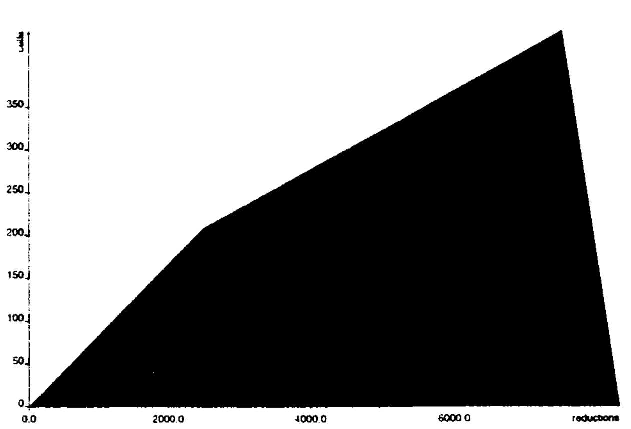
Our research mirrors similar work in LISP[5] and SML[2]. All of the implementations use higher-order functions named memo that return memo-functions.

How is our research different? The challenge that we have faced is referential transparency. Both the LISP and SML implementations were simply defined in the source language; which is easy to do because they are imperative languages (with a functional subset). In Haskell, memo must be defined outside of the language with a semantics that is formal enough to show that referential transparency holds.

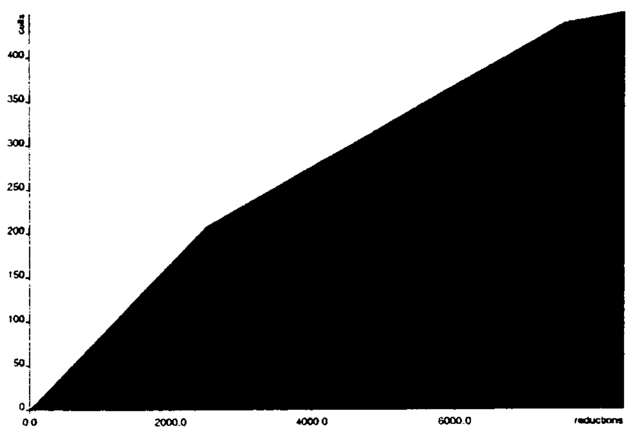
6.3 Memo Gofer

van Dalen[15] extended the Gofer interpreter with a very different notion of lazy memo-functions. As in this paper, memoization is provided with a primitive function memo. However, rather than creating a new function, memo refers to a global memo-table.

In van Dalen's Gofer, a memoized Fibonacci is defined as:

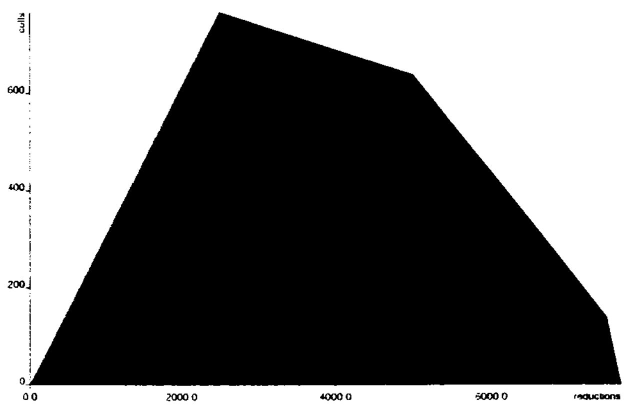


`map (memo f) [1 .. 100]`

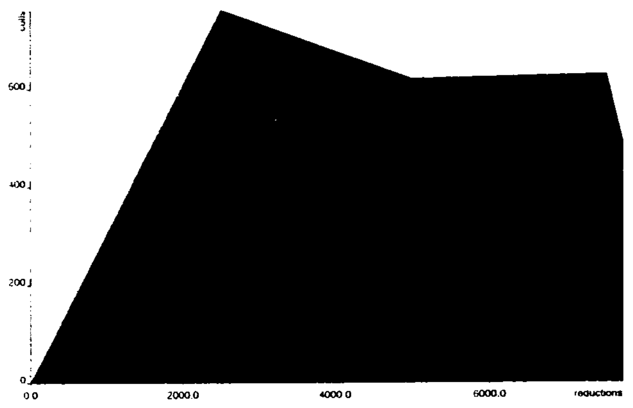


`map f' [1 .. 100]`

where `f x = x+1` and `f' = memo f` at the top-level



```
papply (expr ())
"((((((((((1-1)-1)-1)-1)-2)-2)-2)-1)-3)-4)"
```



```
papply (expr')
"((((((((((1-1)-1)-1)-1)-2)-2)-2)-1)-3)-4)"
```

Figure 5: Heap Profiles of Disposable and Non-disposable Memo-functions

```
memofib = memo mfib
    where
    mfib 0 = 1
    mfib 1 = 1
    mfib n = memo mfib (n-1) + memo mfib (n-2)
```

Notice that, with the semantics in this paper, `memofib` would generate a separate memoized `mfib` for each recursive application.

7 Conclusions

Based on Hughes's lazy memo-functions we have given memoization a formal meaning. The semantics describe memoization at precisely the right level of abstraction. With the semantics, it is possible to reason about the space and performance of memo-functions. Perhaps future research can leverage the semantics and develop better strategies for determining when to memoize.

Huggies is useful because it provides a working prototype and clarifies implementation issues such as overloading, recursion, and polymorphism. The implementation techniques used in Huggies, or even the source code, provide a basis for future refinement.

Acknowledgements

We wish to thank Simon Peyton Jones and Zena Ariola for the original discussions that fostered this research; Bobbie Lavender, Erik Meijer, Koen Claessen, John Matthews, Mark Shields, and the "half-baked talk" attendees for their valuable suggestions; Alastair Reid for his extensive comments on both our code and paper. John Hughes, Mark Jones, and Conal Elliot, for useful insights; the referees of the Haskell Workshop for their helpful comments; and Tito Autrey for the title "Huggies".

References

[1] ARIOLA, Z. M., AND FELLEISEN, M. The call-by-need lambda calculus. *Journal of Functional Programming 1*, 1 (1993), 1–000.

[2] BERRY, D. The Edinburgh SML library. Tech. Rep. ECS-LFCS-91-148, Department of Computer Science, The University of Edinburgh, 1991.

[3] ELLIOTT, C., AND HUDAK, P. Functional reactive animation. To appear in *The International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

[4] GUNTER, C. A. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Science. The MIT Press, 1992.

[5] HALL, M., AND MAYFIELD, J. Improving the performance of AI software: Payoffs and pitfalls in using automatic memoization. In *International Symposium on Artificial Intelligence* (Monterrey, Mexico, Sept. 1993).

[6] HUDAK, P., PETERSON, J., AND FASEL, J. A gentle introduction to Haskell. Available at www.haskell.org, Dec. 1997.

[7] HUGHES, R. J. M. Lazy memo-functions. In *The Conference on Functional Programming and Computer Architecture* (Nancy, France, Sept. 1985), Springer-Verlag.

[8] HUTTON, G., AND MEIJER, E. Monadic parser combinators. Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[9] JOHN PETERSON, E. Report on the programming language haskell: A non-strict, purely functional language, version 1.4. Available at www.haskell.org, Apr. 1997.

[10] JONES, M. P. The implementation of the Gofer functional programming system. Tech. Rep. YALEU/DCS/RR-1030, Deptartment of Computer Science, Yale University, May 1994.

[11] JONES, S. P., AND MEIJER, E. Henk: A typed intermediate language. To appear in *The Workshop on Types in Compilation* (Amsterdam, The Netherlands, June 1997).

[12] KELLER, R. M., AND SLEEP, M. R. Applicative caching: Programmer control of object sharing and lifetime in distributed implementations of applicative languages. In *The Conference on Functional Programming and Computer Architecture* (Wentworth-by-the-sea, Portsmouth, New Hampshire, Oct. 1981).

[13] LAUNCHBURY, J., AND JONES, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.

[14] MICHIE, D. Memo functions and machine learning. *Nature*, 218 (Apr. 1968), 19–22.

[15] VAN DALEN, L. Incremental evaluation through memoization. Master's thesis, Department of Computer Science, Utrecht University, The Netherlands, 1992.

Green card: a foreign-language interface for Haskell

Simon Peyton Jones
Glasgow University and Oregon Graduate Institute

Thomas Nordin
Royal Institute of Technology, Stockholm, and Oregon Graduate Institute

Alastair Reid
Yale University

February 14, 1997

1 Motivation

A foreign-language interface provides a way for software components written in a one language to interact with components written in another. Programming languages that lack foreign-language interfaces die a lingering death.

This document describes Green Card, a foreign-language interface for the non-strict, purely functional language Haskell. We assume some knowledge of Haskell and C.

1.1 Goals and non-goals

Our goals are limited. We do not set out to solve the foreign-language interface in general: rather we intend to profit from others' work in this area. Specifically, we aim to provide the following, in priority order:

1. A convenient way to call C procedures from Haskell.
2. A convenient way to write COM[1] software components in Haskell, and to call COM components from Haskell.

The ability to call C from Haskell is an essential foundation. Through it we can access operating system services and mountains of other software libraries.

In the other direction, should we be able to write a Haskell library that a C program can use? In principle this makes sense but in practice there is zero demand for it. The exception is that the ability to support some sort of call-backs is essential, but that is a very limited form of C calling Haskell.

[1]Microsoft's Common Object Model (COM) is a language-independent software component architecture. It allows objects written in one language to create objects written in another, and to call their methods. The two objects may be in the same address space, in different address spaces on the same machine, or on separate machines connected by a network. OLE is a set of conventions for building components on top of COM.

Should we support languages other than C? The trite answer is that pretty much everything available as a library is available as a C library. For other languages the right thing to do is to interface to a language-independent software component architecture, rather than to a raft of specific languages. For the moment we choose COM, but CORBA[2] might be another sensible choice.

While we do not propose to call Haskell from C, it does make sense to think of writing COM software components in Haskell that are used by clients. For example, one might write an animated component that sits in a Web page.

This document, however, describes only (1), the C interface mechanism.

2 Foreign language interfaces are harder than they look

Even after the scope is restricted to designing a foreign-language interface from Haskell to C, the task remains surprisingly tricky. At first, one might think that one could take the C header file describing a C procedure, and generate suitable interface code to make the procedure callable from Haskell.

Alas, there are numerous tiresome details that are simply not expressed by the C procedure prototype in the header file. For example, consider calling a C procedure that opens a file, passing a character string as argument. The C prototype might look like this:

```
int open( char *filename )
```

Our goal is to generate code that implements a Haskell procedure with type

```
open :: String -> IO FileDescriptor
```

- First there is the question of data representation. One has to decide either to alter the Haskell language implementation, so that is

[2]CORBA is a vendor-independent competitor of COM.

string representation is identical to that of C, or to translate the string from one representation to another at run time. This translation is conventionally called *marshalling.*

Since Haskell is lazy, the second approach is required. (In general, it is tremendously constraining to try to keep common representations between two languages. For example, precisely how does C lay out its structures?)

- Next come questions of allocation and lifetime. Where should we put the translated string? In a static piece of storage? (But how large a block should we allocate? Is it safe to re-use the same block on the next call?) Or in Haskell's heap? (But what if the called procedure does something that triggers garbage collection, and the transformed string is moved? Can the called procedure hold on to the string after it returns?) Or in C's `malloc`'d heap? (But how will it get deallocated? And `malloc` is expensive.)

- C procedures often accept pointer parameters (such as strings) that can be `NULL`. How is that to be reflected on the host-language side of the interface? For example, if the documentation for open told us that it would do something sensible when called with a `NULL` string, we might like the Haskell type for open to be

  ```
  open :: Maybe String -> IO FileDescriptor
  ```

  so that we can model `NULL` by `Nothing`.

- The desired return type, `FileDescriptor`, will presumably have a Haskell definition such as this:

  ```
  newtype FileDescriptor = FD Int
  ```

  The file descriptor returned by `open` is just an integer, but Haskell programmers often use `newtype` declarations create new distinct types isomorphic to existing ones. Now the type system will prevent, say, an attempt to add one to a `FileDescriptor`.

  Needless to say, the Haskell result type is not going to be described in the C header file.

- The file-open procedure might fail; sometimes details of the failure are stored in some global variable, `errno`. Somehow this failure and the details of what went wrong must be reflected into Haskell's `IO` monad.

- The open procedure causes a side effect, so it is appropriate for its type to be in Haskell's `IO` monad. Some C functions really are functions (that is, they have no side effects), and in this case it makes sense to give them a "pure" Haskell type. For example, the C function `sin` should appear to the Haskell programmer as a function with type

  ```
  sin :: Float -> Float
  ```

- C procedure specifications are not explicit about which parameters are in parameters, which out and which in out.

None of these details are mentioned in the C header file. Instead, many of them are in the manual page for the procedure, while others (such as parameter lifetimes) may not even be written down at all.

3 Overview of Green Card

The previous section bodes ill for an automatic system that attempts to take C header files and automatically generate the "right" Haskell functions; C header files simply do not contain enough information.

The rest of this paper describes how we approach the problem. The general idea is to start from the *Haskell* type definition for the foreign function, rather than the *C* prototype. The Haskell type contains quite a bit more information; indeed, it is often enough to generate correct interface code. Sometimes, however, it is not, in which case we provide a way for the programmer to express more details of the interface. All of this is embodied in a program called "Green Card".

Green Card is a Haskell pre-processor. It takes a Haskell module as input, and scans it for Green-Card directives (which are lines prefixed by "`%`"). It produces a new Haskell module as output, and sometimes a C module as well (Figure 1).

Green Card's output depends on the particular Haskell implementation that is going to compile it. For the Glasgow Haskell Compiler (GHC), Green Card generates Haskell code that uses GHC's primitive `ccall`/`casm` construct to call C. All of the argument marshalling is done in Haskell. For Hugs, Green Card generates a C module to do most of the argument marshalling, while the generated Haskell code uses Hugs's `prim` construct to access the generated C stubs.

For example, consider the following Haskell module:

```
module M where

%fun sin :: Float -> Float

sin2 :: Float -> Float
sin2 x = sin (sin x)
```

Everything is standard Haskell except the `%fun` line, which asks Green Card to generate an interface to a (pure) C function `sin`. After the GHC-targeted version of Green Card processes the file, it looks like this[3]:

[3]Only GHC aficionados will understand this code. The whole point of Green Card is that Joe Programmer should not have to learn how to write this stuff.

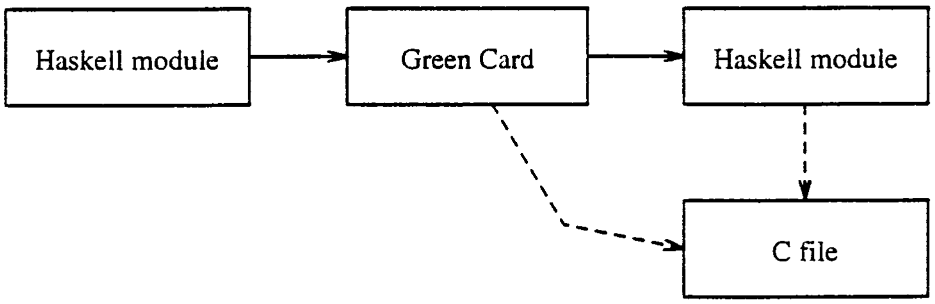


Figure 1: The big picture

```
module M where

sin :: Float -> Float
sin f = unsafePerformPrimIO (
          case f of { F# f# ->
          _casm_ ``%r = sin(%0)'' f#
                   `thenPrimIO` \ r# ->
          returnPrimIO (F# r#)})

sin2 :: Float -> Float
sin2 x = sin (sin x)
```

The %fun line has been expanded to a blob of gruesome boilerplate, while the rest of the module comes through unchanged.

If Hugs were the target, the Haskell source file remains unchanged, but a the Hugs variant of Green Card would generate output that uses Hugs's primitive mechanisms for calling C. Much of the Green-Card implementation is, however, shared between both variants. *(We hope. The Hugs variant isn't even written.)*

4 Green Card directives

Green Card pays attention only to Green-Card directives, each of which starts with a "%" at the beginning of a line. All other lines are passed through to the output Haskell file unchanged.

The syntax of Green Card directives is given in Figure 2). The syntax for *dis* is given later (Figure 3). The form Any_x means any symbol except x.

Green Card understands the following directives:

- %fun begins a *procedure specification*, which describes the interface to a single C procedure (Section 5).
- %dis allows the programmer to describe a new *Data Interface Scheme* (DIS). A DIS describes how to translate, or marshall, data from Haskell to C and back again (Section 6).
- %const makes it easy to generate a collection of new Haskell constants derived from C constants. This can be done with %fun, but %const is much more concise (Section 5.6).
- %prefix makes it easy to remove standard prefixes from the Haskell function name, those are usually not needed since Haskell allows qualified imports (Section 5.7).
- Procedure specifications can, as we shall see, contain fragments of C. %#include tells Green Card to arrange that a specified C header file will be included with the C code in the procedure specifications when the latter is fed to a C compiler (Section 8).

A directive can span more than one line, but the continuation lines must each start with a % followed by some whitespace. For example:

```
%fun draw :: Int       -- Length in pixels
%         -> Maybe Int -- Width in pixels
%         -> IO ()
```

Haskell-style comments are permitted in Green-Card directives.

A general principle we have followed is to define a single, explicit (and hence long-winded) general mechanism, that should deal with just about anything, and then define convenient abbreviations that save the programmer from writing out the general mechanism in many common cases. We have erred on the conservative side in defining such abbreviations; that is, we have only defined an abbreviation where doing without it seemed unreasonably long-winded, and where there seemed to be a systematic way of defining an abbreviation.

5 Procedure specifications

The most common Green-Card directive is a procedure specification. It describes the interface to a C procedure. A procedure specification has four parts:

Type signature: %fun (Section 5.1). The %fun statement starts a new procedure specification,

Program	*idl*	→	$decl_1, \ldots decl_n$	$n \geq 1$
Declaration	*decl*	→	*proc*	
		|	**%const** *Var* [$Var_1 \ldots Var_n$]	*Constants* $n \geq 1$
		|	**%dis** *Var* $Var_1 \ldots Var_n$ = *dis*	$n \geq 0$
		|	**%#include** *filename*	*Scope over cexp*
		|	**%prefix** *Var*	*Prefix to strip from Haskell function names*
Procedure	*proc*	→	*sig* [*call*] [*ccode*] [*result*]	
Signature	*sig*	→	**%fun** *Var* :: *Type*	*Name and type*
Type	*type*	→	*Var*	
		|	*Var type*	
		|	*type* -> *type*	
		|	($type_1$, ... $type_n$)	*Tuple* $n \geq 0$
Call	*call*	→	**%call** $dis_1 \ldots dis_n$	
Result	*result*	→	**%fail** *cexp cexp* [*result*]	*In IO Monad*
		|	**%result** *dis*	
C Expression	*cexp*	→	{ $Any_{\}}$ }	
		|	*ccode*	
C Code	*ccode*	→	**%code** *Var*	
Filename	*filename*	→	<*Var*>	*Passed to C*
		|	"*Var*"	*Passed to C*

Figure 2: Grammar for Green Card

giving the name and Haskell type of the function.

Parameter marshalling: `%call` (Section 5.2). The `%call` statement tells Green Card how to translate the Haskell parameters into their C representations.

The body: `%code` (Section 5.3). The `%code` statement gives the body and it can contain arbitrary C code. Sometimes the body consists of a simple procedure call, but it may also include variable declarations, multiple calls, loops, and so on.

Result marshalling: `%result`, `%fail` (Section 5.4). The result-marshalling statements tell Green Card how to translate the result(s) of the call back into Haskell values.

Any of these parts may be omitted except the type signature. If any part is missing, Green Card will fill in a suitable statement based on the type signature given in the `%fun` statement. For example, consider this procedure specification:

```
%fun sin :: Float -> Float
```

Green Card fills in the missing statements like this[4]:

```
%fun sin :: Float -> Float
%call (float x1)
%code result = sin(x1);
%result (float result)
```

The rules that guide this automatic fill-in are described in Section 5.5.

A procedure specification can define a procedure with no input parameter, or even a constant (a "procedure" with no input parameters and no side effects). In the following example, `printBang` is an example of the former, while grey is an example of the latter[5]:

```
%fun printBang :: IO ()
%code printf( "!" );

%fun grey :: Colour
%code r = GREY;
%result (colour r)
```

All the C variables bound in the `%call` statement or mentioned in the `%result` statement, are declared by Green Card and in scope throughout the body. In the examples above, Green Card would have declared `x1`, `result` and `r`.

[4]The details of the filled-in statements will make more sense after reading the rest of Section 5

[5]When there are no parameters, the `%call` line can be omitted. The second example can also be shortened by writing a C expression in the `%result` statement; see Section 5.4.

5.1 Type signature

The %fun statement starts a new procedure specification.

Green Card supports two sorts of C procedures: ones that may cause side effects (including I/O), and ones that are guaranteed to be pure functions. The two are distinguished by their type signatures. Side-effecting functions have the result type IO t for some type t. If the programmer specifies any result type other than IO t, Green Card takes this as a promise that the C function is indeed pure, and will generate code that calls unsafePerformIO.

The procedure specification will expand to the definition of a Haskell function, whose name is that given in the %fun statment, with two changes: the longest matching prefix specified with a %prefix (Section 5.7 elaborates)statement is removed from the name and the first letter of the remaining function name is changed to lower case. Haskell requires all function names to start with a lower-case letter (upper case would indicate a data constructor), but when the C procedure name begins with an upper case letter it is convenient to still be able to make use of Green Card's automatic fill-in facilities. For example:

```
%fun OpenWindow :: Int -> IO Window
```

would expand to a Haskell function openWindow that is implemented by calling the C procedure OpenWindow.

```
%prefix Win32
%fun Win32OpenWindow :: Int -> IO Window
```

would expand to a Haskell function openWindow that is implemented by calling the C procedure Win32OpenWindow.

5.2 Parameter marshalling

The %call statement tells Green Card how to translate the Haskell parameters into C values. Its syntax is designed to look rather like Haskell pattern matching, and consists of a sequence of zero or more Data Interface Schemes (DISs), one for each (curried) argument in the type signature. For example:

```
%fun foo :: Float -> (Int,Int) -> String -> IO ()
%call (float x) (int y, int z) (string s)
...
```

This %call statement binds the C variables x, y, z, and s, in a similar way that Haskell's pattern-matching binds variables to (parts of) a function's arguments. These bindings are in scope throughout the body and result-marshalling statements.

In the %call statement, "float", "int", and "string" are the names of the DISs that are used to translate between Haskell and C. The names of these DISs are deliberately chosen to be the same as the corresponding Haskell types (apart from changing the initial letter to lower case) so that in many cases, including foo above, Green Card can generate the %call line by itself (Section 5.5). In fact there is a fourth DIS hiding in this example, the (_,_) pairing DIS. DISs are discussed in detail in Section 6.

5.3 The body

The body consists of arbitrary C code, beginning with %code. The reason for allowing arbitrary C is that C procedures sometimes have complicated interfaces. They may return results through parameters passed by address, deposit error codes in global variables, require #include'd constants to be passed as parameters, and so on. The body of a Green Card procedure specification allows the programmer to say exactly how to call the procedure, in its native language.

The C code starts a block, and may thus start with declarations that create local variables. For example:

```
%code int x, y;
%       x = foo( &y, GREY );
```

Here, x and y are declared as local variables. The local C variables declared at the start of the block scope over the rest of the body *and* the result-marshalling statements.

The C code may also mention constants from C header files, such as GREY above. Green Card's %#include directive tells it which header files to include (Section 8).

5.4 Result marshalling

Functions return their results using a %result statement. Side-effecting functions — ones whose result type is IO t — can also use %fail to specify the failure value.

5.4.1 Pure functions

The %result statement takes a single DIS that describes how to translate one or more C values back into a single Haskell value. For example:

```
%fun sin :: Float -> Float
%call (float x)
%code ans = sin(x);
%result (float ans)
```

As in the case of the %call statement, the "float" in the %result statement is the name of a DIS, chosen as before to coincide with the name of the type. A single DIS, "float", is used to denote both the translation from Haskell to C and that from C to Haskell, just as a data constructor can be used both to construct a value and to take one apart (in pattern matching).

All the C variables bound in the %call statement, and all those bound in declarations at the start of the body, scope over all the result-marshalling statements (i.e. `%result` and `%fail`).

5.4.2 Arbitrary C results

In a result-marshalling statement an almost arbitrary C expression, enclosed in braces, can be used in place of a C variable name. The above example could be written more briefly like this[6]:

```
%fun sin :: Float -> Float
%call (float x)
%result (float {sin(x)})
```

The C expression can neither have assignments nor nested braces as that could give rise to syntactic ambiguity (Section 2 elaborates).

5.4.3 Side effecting functions

A side effecting function returns a result of type `IO t` for some type `t`. The `IO` monad supports exceptions, so Green Card allows them to be raised.

The result-marshalling statements for a side-effecting call consists of zero or more `%fail` statements, each of which conditionally raise an exception in the `IO` monad, followed by a single `%result` statement that returns successfully in the `IO` monad.

Just as in Section 5.4, the `%result` statement gives a single DIS that describes how to construct the result Haskell value, following successful completion of a side-effecting operation. For example:

```
%fun windowSize :: Window -> IO (Int,Int)
%call (window w)
%code struct WindowInfo wi;
%     GetWindowInfo( w, &wi );
%result (int {wi.x}, int {wi.y})
```

Here, a pairing DIS is used, with two `int` DISs inside it. The arguments to the `int` DISs are C record selections, enclosed in braces; they extract the relevant information from the `WindowInfo` structure that was filled in by the `GetWindowInfo` call[7].

The `%fail` statement has two fields, each of which is either a C variable, or a C expression enclosed in braces. The first field is a boolean-valued expression that indicates when the call should fail; the second is a `(char *)`-valued that indicates what sort of failure occurred. If the boolean is true (i.e. non zero) then the call fails with a `UserError` in the `IO` monad containing the specified string.

For example:

```
%fun fopen :: String -> IO FileHandle
%call (string s)
%code f = fopen( s );
%fail {f == NULL} {errstring(errno)}
%result (fileHandle f)
```

[6]It can be written more briefly still by using automatic fill-in (Section 5.5).

[7]This example also shows one way to interface to C procedures that manipulate structures.

The assumption here is that `fopen` puts its error code in the global variable `errno`, and `errstring` converts that error number to a string.

`UserErrors` can be caught with `catch`, but exactly which error occurred must be encoded in the string, and parsed by the error-handling code. This is rather slow, but errors are meant to be exceptional.

5.5 Automatic fill-in

Any or all of the parameter-marshalling, body, and result-marshalling statements may be omitted. If they are omitted, Green Card will "fill in" plausible statements instead, guided by the function's type signature. The rules by which Green Card does this filling in are as follows:

- A missing `%call` statement is filled in with a DIS for each curried argument. Each DIS is constructed from the corresponding argument type as follows:
  - A tuple argument type generates a tuple DIS, with the same algorithm applied to the components.
  - All other types generate a DIS function application (Section 6.1). The DIS function name is derived from the type of the corresponding argument, except that the first letter of the type is changed to lower case. The DIS function is applied to as many argument variables as required by the arity of the DIS function.
  - The automatically-generated argument variables are named left-to-right as arg1, arg2, arg3, and so on.
- If the body is missing, Green Card fills in a body of the form:

  `%code` $r = f(a_1, \ldots, a_n);$

  where
  - f is the function name given in the type signature.
  - $a_1 \ldots a_n$ are the argument names extracted from the `%call` statement.
  - r is the variable name for the variable used in the `%result` statement. (There should only be one such variable if the body is automatically filled in.)
- A missing `%result` statement is filled in by a `%result` with a DIS constructed from the result type in the same way as for a `%call`. The result variables are named `res`, `res2`, `res3`, and so on.
- Green Card never fills in `%fail` statements.

5.6 Constants

Some C header files define a large number of constants of a particular type. The `%const` statement provides a convenient abbreviation to allow these constants to be imported into Haskell. For example:

```
%const PosixError [EACCES, ENOENT]
```

This statement is equivalent to the following `%fun` statements:

```
%fun EACCES :: PosixError
%fun ENOENT :: PosixError
```

After the automatic fill-in has taken place we would obtain:

```
%fun EACCES :: PosixError
%result (posixError { EACCES })

%fun ENOENT :: PosixError
%result (posixError { ENOENT })
```

Each constant is made available as a Haskell value of the specified type, converted into Haskell by the DIS function for that type. (It is up to the programmer to write a `%dis` definition for the function — see Section 6.2.)

5.7 Prefixes

In C it is common practise to give all function names in a library the same prefix, to minimize the impact on the common namespace. In Haskell we use qualified imports to achieve the same result. To simplify the conversion of C style namespace management to Haskell the `%prefix` statement specifies which prefixes to remove from the Haskell function names.

```
module OpenGL where

%prefix OpenGL
%prefix gl

%fun OpenGLInit :: Int -> IO Window
%fun glSphere :: Coord -> Int -> IO Object
```

This would define the two procedures Init and Sphere which would be implemented by calling OpenGLInit and glSphere respectively.

6 Data Interface Schemes

A *Data Interface Scheme*, or DIS, tells Green Card how to translate from a Haskell data type to a C data type, and vice versa.

6.1 Forms of DISs

The syntax of DISs is given in Figure 3. It is designed to be similar to the syntax of Haskell patterns. A DIS takes one of the following forms:

1. *The application of a DIS function to zero or more arguments.* Like Haskell functions, a DIS function starts with a lower-case letter. DIS function are described in Section 6.2. Standard DIS functions include `int`, `float`, `double`; the full set is given in Section 7. For example:

   ```
   %fun foo :: This -> Int -> That
   %call (this x y) (int z)
   %code r = c_foo( x, y, z );
   %result (that r)
   ```

   In this example `this` and `that` are DIS functions defined elsewhere.

2. *The application of a Haskell data constructor to zero or more DISs.* For example:

   ```
   newtype Age = Age Int
   %fun foo :: (Age,Age) -> Age
   %call (Age (int x), Age (int y))
   %code r = foo(x,y);
   %result (Age (int r))
   ```

   As the `%call` line of this example illustrates, tuples are understood as data constructors, including their special syntax. Haskell record syntax is also supported. For example:

   ```
   data Point = Point { px,py::Int }

   %fun foo :: Point -> Point
   %call (Point { px = int x, py = int y })
   ...
   ```

   The use of records is also the reason for the restriction that simple C expressions can't contain assignment. Without this restriction examples like this would be ambiguous:

   ```
   %result Foo { a = bar x, b = bar y }
   ```

   Green Card does not attempt to perform type inference; it simply assumes that any DIS starting with an upper case letter is a data constructor, and that the number of argument DISs matches the arity of the constructor.

3. *A C type cast, enclosed in braces, followed by a C variable name.* It only makes sense in a version of Haskell extended with unboxed types, because only they need no translation. Examples:

   ```
   %fun foo :: Int# -> IO ()
   %call ({int} x)
   ...

   data T = MkT Int#
   %fun baz :: T -> IO ()
   %call (MkT ({int} x))
   ...
   ```

DIS	*dis*	→	*disfun* $arg_1 \ldots arg_n$	*Application*
		|	*Cons* $arg_1 \ldots arg_n$	*Constructor* $n \geq 0$
		|	*Cons*$\{field_1 = dis_1, \ldots field_n = dis_n\}$	*Record* $n \geq 1$
		|	*adis*	
ADIS	*adis*	→	(*dis*)	
		|	*tc cexp*	**result** *only*
		|	*tc var*	
		|	*var*	*Bound by* `%dis`
		|	$(dis_1, \ldots dis_n)$	*Tuple* $n \geq 0$
Arg	*arg*	→	*adis*	
		|	*cexp*	
		|	*var*	
DisFun	*disfun*	→	*var*	
TypeCast	*tc*	→	*cexp*	*C Expression*
Variable	*var*	→	*Var*	*Initial letter lower case*

Figure 3: DIS grammar

6.2 DIS functions

It would be unbearably tedious to have to write out complete DISs in every procedure specification, so Green Card supports *DIS functions* in much the same way that Haskell provides functions. (The big difference is that DIS functions can be used in "patterns" — such as %call statements — whereas Haskell functions cannot.)

Green Card supports two sorts of DIS function: DIS macros (Section 6.2.1) and user-defined DISs (Section 6.2.2).

6.2.1 DIS macros

DIS macros allow the programmer to define abbreviations for commonly-occurring DISs. For example:

```
newtype This = MkThis Int (Float, Float)
%dis this x y z = MkThis (int x)
                         (float y, float z)
```

Along with the newtype declaration the programmer can write a %dis function definition that defines the DIS function this in the obvious manner.

DIS macros are simply expanded out by Green Card before it generates code. So for example, if we write:

```
%fun f :: This -> This
%call (this p q r)
...
```

Green Card will expand the call to this:

```
%fun f :: This -> This
%call (MkThis (int p) (float q, float r))
...
```

(In fact, int and float are also DIS macros defined in Green Card's standard prelude, so the %call line is further expanded to:

```
%fun f :: This -> This
%call (MkThis (I# ({int} p))
              (F# ({float} q), F# ({float} r)))
...
```

The fully expanded calls describe the marshalling code in full detail; you can see why it would be inconvenient to write them out literally on each occasion!)

Notice that DIS macros are automatically bidirectional; that is, they can be used to convert Haskell values to C *and vice versa*. For example, we can write:

```
%fun f :: This -> This
%call (MkThis (int p) (float q, float r))
%code int a, b, c;
%     f( p, q, r, &a, &b, &c);
%result (this a b c)
```

The form of DIS macro definitions, given in Figure 3, is very simple. The formal parameters can only be variables (not patterns), and the right hand side is simply another DIS. Only first-order DIS macros are permitted.

6.2.2 User-defined DISs

Sometimes Green Card's primitive DISs (data constructors) are insufficiently expressive. For recursive types, such as lists, it is obviously no good to write a single data constructor.

Green Card therefore provides a "trap door" to allow a sufficiently brave programmer to write his or her own marshalling functions. For example:

```
data T = Zero | Succ T

%fun square :: T -> T
%call (t (int x))
%code r = square( x );
%result (t (int r))
```

Use of `t` requires that the programmer define two ordinary Haskell functions, `marshall_t` to convert from Haskell to C, and `unmarshall_t` to convert in the other direction. In this example, these functions would have the types:

```
marshall_t   :: T -> Int
unmarshall_t :: Int -> T
```

The functions must have precisely these names: "`marshall_`" followed by the name of the DIS, and similarly for unmarshall. Notice that these marshalling functions have pure types (e.g. `marshall_t` has type `T -> Int` rather than `T -> IO Int`). Sometimes one wants to write a marsalling function that is internally stateful. For example, it might pack a `[Char]` into a `ByteArray`, by allocating a `MutableByteArray` and filling it in with the characters one at a time. This can be done using `runST`, or even `unsafePerformIO`. (These are all GHC-specific comments; so far as Green Card is concerned it is simply up to the programmer to supply suitably-typed marshalling functions.)

Green Card distinguishes user-defined DISs from DIS macros by omission: if there is a DIS macro definition for a DIS function `f` then Green Card treats `f` as a macro, otherwise it assumes `f` is a user-defined DIS and generates calls to `marshall_t` and/or `unmarshall_t`.

6.3 Semantics of DISs

How does Green Card use these DISs to convert between Haskell values and C values? We give an informal algorithm here, although most programmers should be able to manage without knowing the details.

To convert from Haskell values to C values, guided by a DIS, Green Card does the following:

- First, Green Card rewrites all DIS function applications, replacing left hand side by right hand side.
- Next, Green Card works from outside in, as follows:
  - For a data constructor DIS (in either positional or record form), Green Card generates a Haskell case statement to take the value apart.
  - For a user-defined DIS, Green Card calls the DIS's `marshall` function.
  - For a type-cast-with-variable DIS, Green Card does no translation.

Much the same happens in the other direction, except that Green Card calls the `unmarshall` function in the user-defined DIS case.

7 Standard DISs

Figure 4 gives the DIS functions that Green Card provides as a "standard prelude".

The "T" variants allow the programmer to specify what type is to be used as the C representation type. For example, the `int` DIS maps a Haskell `Int` to a C `int`, whereas `intT {FD}` maps a Haskell `Int` onto a C value with type `FD`.

7.1 GHC extensions

Several of the standard DISs involve types that go beyond standard Haskell:

- `Addr` is a GHC type large enough to contain a machine address. The Haskell garbage collector treats it as a non-pointer, however.
- `ForeignObj` is a GHC type designed to contain a reference to a foreign resource of some kind: a `malloc`'d structure, a file descriptor, an X-windows graphic context, or some such. The size of this reference is assumed to be that of a machine address. When the Haskell garbage collector decides that a value of type `ForeignObj` is unreachable, it calls the object's finalisation routine, which was given as an address in the argument of the DIS. The finalisation routine is passed the object reference as its only argument.
- The `stable` DIS maps a value of any type onto a C `int`. The `int` is actually an index into the *stable pointer table*, which is treated as a source of roots by the garbage collector. Thus the C procedure can effectively get a reference into the Haskell heap. When `stable` is used to map from C to Haskell, the process is reversed.

7.2 Maybe

Almost all DISs work on single-constructor data types. It is much less obvious how to translate values of multi-constructor data types to and from C. Nevertheless, Green Card does deal in an *ad hoc* fashion with the `Maybe` type, because it seems so important.

The syntax for the `maybeT` DIS is:

```
maybeT cexp dis
```

DIS	Haskell	C type	Comments
int x	Int	int x	
intT t x	Int	t x	
char c	Char	char c	
charT t c	Char	t c	
float f	Float	float f	
floatT t f	Float	t f	
double d	Double	double d	
doubleT t d	Double	t d	
bool b	Bool	int b	0 for False, 1 for True
boolT t b	Bool	t b	
addr a	Addr	void *a	An immovable C-land address
addrT t a	Addr	t a	
string s	String	char *s	Persistence not required in either direction.
foreign x f	ForeignObj	void *x, void *f()	f is the free routine; it takes one parameter, namely x, the thing to be freed.
foreignT t x f	ForeignObj	t x, void *f()	
stable x	*any*	int	Makes it possible to pass a Haskell pointer to C, and perhaps get it back later, without breaking the garbage collector.
stableT t x	*any*	t	
maybe dis	Maybe dis	type of dis	Converts to and from Maybe's, with 0 as Nothing
maybeT cexp dis	Maybe dis	type of dis	Converts to and from Maybe's

Figure 4: Standard DISs

where dis is any DIS, and cexp is a C expression which represents the Nothing value in the C world.

In the following example, the function foo takes an argument of type Maybe Int. If the argument value is Nothing it will bind x to 0; if it is Just a it will bind x to the value of a. The return value will be Just r unless r == -1 in which case it will be Nothing.

```
%fun foo :: Maybe Int -> Maybe Int
%call (maybeT { 0 } (int x))
%code r = foo(x);
%result (maybeT { -1 } (int r))
```

There is also a maybe DIS wich just takes the DIS and defaults to 0 as the Nothing value.

8 Imports

Green Card "connects" with code in other modules in two ways:

- Green Card reads the source code of any modules directly imported by the module being processed. It extracts %dis function definitions (only) from these modules. This provides an easy mechanism for Green Card to import DIS functions defined elsewhere.
- It is often important to arrange that a C header file is #included when the C code fragments in Green Card directives is compiled.

The %#include directive performs this delayed #include. The syntax is exactly that of a C #include apart from the initial %.

9 Invoking Green Card

The general syntax for invoking Green Card is:

```
green-card [options] [filename]
```

Green Card reads from standard input if no filename is given. The options can be any of those:

--version Print the version number, then exit successfully.

--help Print a usage message listing all available options, then exit successfully.

--verbose Print more information while processing the input.

--include-dir <directories> Search the directories named in the colon (:) separated list for imported files. The directories will be searched in a left to right order.

--fgc-safe Generates code that can use callbacks to Haskell. This makes the generated code slower.

10 Related Work

- *A Portable C Interface for Standard ML of New Jersey*, by Lorenz Huelsbergen, describes the implementation of a general interface to C for SML/NJ.
- *Simplified Wrapper and Interface Generator* (SWIG) generate interfaces from (extended) ANSI C/C++ function and variable declarations. It can generate output for Tcl/Tk, Python, Perl5, Perl4 and Guile-iii. SWIG lives at `http://www.cs.utah.edu/~beazley/SWIG/`
- *Foreign Function Interface GENerator* (FFIGEN) is a tool that parses C header files and presents an intermediate data representation suitable for writing backends. FFIGEN lives at `http://www.cs.uoregon.edu/~lth/ffigen/`
- *Header2Scheme* is a program which reads C++ header files and compiles them into C++ code. This code implements the back end for a Scheme interface to the classes defined by these header files. Header2Scheme can be found at:

  `http://www-white.media.mit.edu/~kbrussel/Header2Scheme/`

11 Alternative design choices and avenues for improvement

Here we summarise aspects of Green Card that are less than ideal, and indicate possible improvements.

DIS function syntax. DIS functions are a bit like Haskell functions (which is why they start with a lower case letter), but they are also very like a "view" of a data type; that is, a pseudo-constructor that allows you to build a value or pattern-match on it. Maybe, therefore, DIS functions should start with a capital letter. (Then user-defined DISs could start with a plain lower-case letter.) Trivial but important.

Automatic DIS generation. Pretty much every `newtype` or single-constructor declaration that is involved in a foreign language call needs a corresponding `%dis` definition. Maybe this `%dis` definition should be automated. On the other hand, there are many fewer data types than procedures, so perhaps it isn't too big a burden to define a `%dis` for each.

User defined DISs. Should user-defined DISs be explicitly declared, rather than inferred by the omission of a DIS macro definition? Should it be possible for the programmer to specify the name of the marshall/unmarshall functions? (Omitted for now because not strictly necessary.)

Imports. Should the `%dis` import mechanism be recursive? That is, should Green Card read the source of all modules in the transitive closure of the module's imports?

Structures. Green Card lacks explicit support for translating structures between C and Haskell. How important is it? What is the "right" way to provide such support?

Error handling. The error handling provided by `%fail` is fairly rudimentary. It isn't obvious how to improve it in a systematic manner.

Heap Compression and Binary I/O in Haskell

Malcolm Wallace Colin Runciman
Dept of Computer Science
University of York, UK
{malcolm,colin}@cs.york.ac.uk

Abstract

Two new facilities for Haskell are described: compression of data values in memory, and a new scheme for binary I/O. These facilities, although they can be used individually, can also be combined because they use the same binary representations for values. Heap compression in memory is valuable because it enables programs to run on smaller machines, or conversely allows programs to store more data in the same amount of memory. Binary I/O is valuable because it makes the file storage and retrieval of heap data structures smooth and painless. The combination of heap compression and binary I/O allows data transfer to be both fast and space-efficient.

All the facilities described have been implemented in a variant of Röjemo's *nhc* compiler. Example applications are demonstrated, with performance results for space and speed.

1 Introduction

1.1 Data representation

Implementors of lazy functional languages tend to use an internal representation of data which is uniform, based on graphs of heap cells. A value is either atomic, occupying one unit of space, or it is structured and each of its components occupies one unit of space, in turn either an atomic value or a pointer to another structured value.

There are good reasons for this memory model. Functional programming systems make little distinction between values and expressions, so a memory cell must be capable of representing either form. Also, functions may be polymorphic and it is therefore very useful for one unit of memory space to be able to hold a value or expression of any type, no matter how simple or complex.

However there are various occasions when this internal representation is inconvenient:

1. **Static data.** Some programs have a large quantity of essentially static data, such as the import table in a compiler, or the dictionary in a natural-language processing system. The standard representation of this data is somewhat bulky. Also, the reasons for using the standard representation do not apply: the data can be fully evaluated early in the computation (so it contains no expressions), and its type is also fully known (there is no remaining polymorphism). In such a situation, a more compact representation can allow more data to be stored. With care, the overhead of garbage collection can also be reduced.

2. **Secondary storage.** Some programs perform output with the intention of being able to retrieve the information from secondary storage in a later input operation. For instance, some compilers generate interface files which are read back during later compilation of separate modules; some applications save large quantities of internal state information for later analysis (perhaps by a tracer or profiler). Haskell's standard mechanism for I/O allows program data to be transferred only in a textual representation. This requires an expensive translation at both output and input stages. It is much more convenient to be able to use the same binary data representation both in memory and on files, so that I/O can be a cheap bulk transfer from one to the other. The standard graph representation does not lend itself to this approach.

3. **Foreign language interfaces.** A Haskell programmer may occasionally wish to call a routine written in C, for instance to perform a system call.

Data holding the same semantic information is frequently represented differently in each language, and it must therefore be *marshalled* before passing from one to the other. Again, the ability to define and use a common representation would be very convenient.

4. **Embedded systems.** Device control is also related to I/O. Programs have a high-level view of certain data structures used for control and monitoring. At some stage these structures must be mapped onto narrow bitfields within individual machine registers.

5. **Communication.** I/O in the form of dynamic transmission of data between processes, whether across a network or on the same machine, can be hindered if it must rely on either a shared memory model or on textual representations.

In this paper we show how the programmer can have control over data representation with little compromise of the high-level abstraction facilities which make functional languages attractive.

We provide a means by which the internal representation of program data can be specified to a fine-grained binary level. The type of a value is used to determine how it can be represented as a sequence of bits. Functions are defined to transfer a value between the standard graph representation and these compressed bit sequences. The two conversion functions form the methods of a type class. A small set of primitive operators is used in the definition of the conversion functions, and the I/O monad is used for sequencing. Instances of the compression type class can be derived automatically by the compiler. The programmer can also define custom instances.

An extension to the I/O library is also provided, allowing values to be transferred between the functional program and the rest of the world in compact binary form.

All the examples and underlying facilities described in this paper have been implemented using *nhc* version 1.3 [9].

We leave the question of data representations for foreign language interfaces and embedded systems control as interesting lines of future work. (See however our earlier work on embedded systems [15, 16].)

1.2 Motivation

This work is funded by Canon Research Centre Europe Ltd., who are developing complex software for new ranges of products. Functional languages are very attractive for reasons of programmer productivity, the ease of rapid prototyping, and maintainability of the emerging software. Together, these benefits can bring a product to market more quickly, which is a considerable commercial advantage. Five issues of concern however are:

1. saving memory space in the final product;
2. achieving fast and efficient I/O;
3. interfacing to other product modules written in C/C++;
4. running software in an embedded product;
5. communication between multiple processes.

The most important issue from a commercial perspective is probably the first: saving memory leads to a direct saving on mass-production costs. This sets the scene for our work on compressing heap data and performing binary I/O. However, as the introduction has outlined, one common theme which underlies all five issues is data representation. A declarative solution to the representation problem has the potential to address many challenges from the software engineering arena.

2 A Class of Types with Bit-Vector Representations

To recap, data in a functional language system is usually represented as a linked graph structure, where each link and terminal node typically occupies one machine word. However, it is possible to use type information about values to squeeze the representation down to a much smaller sequence of bits.

2.1 Types and compression

If a type admits just n different values, any value of that type can be represented within *log(n)* bits. It is clear how this can be applied to an enumerated type, with only nullary constructors. But the same observation also applies to more structured types. In Haskell a structured data value of a type T consists of an n-ary constructor followed by a sequence of n values, each of which belongs to some type $t_0..t_{n-1}$. Hence, a structured value can be represented in binary form by a vector of bits. The first portion of the vector identifies the constructor, and the remainder of the vector is a sequence of values, each also represented in binary form. Where a data type has more than one constructor, different values of the same type may occupy very different amounts of memory.

Because the precise details of bit-vector representation differ from type to type, the obvious mechanism to use is the ad-hoc polymorphism of type classes.

```
class Compress a where
  compress  :: a -> IO (Bin a)
  expand    :: Bin a -> a
```

The type `Bin a` is an *abstract* type, implemented by an extension of the Haskell runtime system. Notice that the `compress` function uses the I/O monad, whereas the `expand` function does not. The I/O monad is used primarily to enforce a correct sequence of operations during compression. Also, `compress` is strict in its first argument. It would be no good to us if objects were compressed lazily, because the whole idea is to save space; a lazy `compress` could easily retain a closure containing the original full-sized value against the day when the compressed version was used! For the same reason however, the expansion operation must remain pure and lazy. It is not known which components of the compressed value will be needed in the computation, and so it does not make sense to enforce strictness or sequencing via the I/O monad. We discuss these design choices further in the Future Work section.

2.2 Implementation issues

In order to be able to write instances of `Compress`, we introduce the following primitives.

```
wBin :: Int -> Int -> IO (Bin a)
rBin :: Int -> Bin a -> (Int, Bin b)
```

The intuition is that `wBin s n` writes integer value `n` into a bit vector of width `s`, returning a pointer to the beginning of the vector. Conversely, `rBin s b` reads an integer value of width `s` bits from the vector `b`, returning both the value and a pointer to the remainder of the bit vector beyond the value that has just been read.

Where are bit vectors stored? For our present purposes, it is convenient to store the vectors off the heap, in a separate area of memory which is not garbage-collected. (Again, this pragmatic choice is re-evaluated in the Future Work section.) When creating a bit vector, this area of memory is treated like a sequential file with an internal state determining where to begin writing the next value. There is no operation to glue two bit vectors together. Rather, the explicit sequence of I/O operations performed during compression ensures that vectors are placed next to each other. The polymorphic return types of `wBin` and `rBin` ensure that the type inference system can regard these simple pointers as correctly typed. The class system ensures type safety by always selecting the correct instances of `compress` and `expand` for the values involved in any particular computation.

2.3 A small example: truth trees

Figure 1 shows a datatype of binary trees of Booleans, together with the instance definitions needed in order to compress it, and an example tree `t`.

In a standard graph representation such as that used in *nhc* [9], each Boolean occupies one word, each `Branch` occupies three words, and each `Leaf` occupies one word. The total space needed to represent `t` is at worst 27 words, or at best (assuming maximal sharing) 16 words. Implementations such as *Gofer* [4] use four words per `Branch`, bringing the total to between 32 and 20 words.

Under the bit-packing scheme, one bit is sufficient to distinguish branches from leaves, and one further bit distinguishes `True` from `False`. In total, the compressed `t` occupies exactly 17 *bits*. This is better than an order of magnitude saving – the compression ratio is between 15x and 60x, depending on word-size and the extent of sharing. Clearly this example is a best-case due to the high compressibility of Booleans, but we shall obtain very worthwhile compression ratios for more realistic applications (see section 3).

2.4 Derived and explicit instances

It would be tedious to write explicit `Compress` instance definitions for every datatype used in a program. Since the compression scheme we have described is very regular, we have modified the *nhc* compiler to generate instances of `Compress` automatically for datatypes with a `deriving` clause, for example:

```
data Tree = Branch Tree Tree
          | Leaf Bool
          deriving Compress
```

The programmer is still free to try more aggressive compression algorithms, by defining custom instances of the `Compress` class. We have experimented with alternative coding schemes where a knowledge of the expected frequency of values can be used to great advantage. For example, we have written a Haskell program which takes a simple value/frequency table for a type and generates a Haskell module containing the appropriate instance declarations for Huffman compression [1]. In our experience of specific applications, Huffman coding can roughly double the compression ratio. There are other fruitful avenues for compression, especially for character strings.

2.5 Limitations on compressible values

It can be seen that the `compress` function is strict. This means that compression is really only suitable for data which is largely static: it may be computed once, but

```
data Tree = Branch Tree Tree
          | Leaf Bool

instance Compress Bool where
  compress = wBin 1 . fromEnum
  expand   = toEnum . fst . rBin 1

instance Compress Tree where
  compress (Leaf b) =
    wBin 1 0 >>= \x->
        compress b >>
        return x
  compress (Branch l r) =
    wBin 1 1 >>= \x->
        compress l >>
        compress r >>
        return x
  expand c =
    let (i,c') = rBin 1 c in
    case i of
      0 -> Leaf (expand c')
      1 -> let (l,c'') = expand c'
               (r,_)   = expand c''
           in Branch l r

t :: Tree
t = Branch
      (Branch
        (Branch
          (Leaf True)
          (Branch (Leaf False)
                  (Leaf True)))
        (Branch (Leaf False)
                (Leaf True)))
      (Leaf False)
```

Figure 1: Compression for truth trees. The full bit-vector for t is 11101100011000100.

it then remains constant and useful for the rest of the program's run. Examples of such applications have already been noted: a compiler's import table, a natural-language dictionary, a program-reduction tracer.

There are some other limitations to the representation scheme outlined here.

1. During compression, any sharing in the original structure is lost, because an in-lined copy is made at every site of the sharing. This is inevitable because the purpose of compression is to flatten out the links from the graph structure, keeping only the terminal values and their sequence. As a result of this restriction, neither cyclic nor infinite structures can be compressed.

2. The heap representation of functions cannot be compressed, since a machine address cannot easily be reduced in size. However, it is certainly possible to compress the code itself, expanding it only when it is needed. Just-in-time dynamic compilation is showing good results in this area – see for instance Wakeling's recent work [14]. The main idea is that function code is generated in a compact bytecode representation. This is then expanded at runtime by an on-the-fly compiler into native code which is stored temporarily in the heap. When the heap is full, the native code is thrown away. The bytecode is re-compiled to native code if it is required again.

3. Compressed values are accessed sequentially. For instance, in a compressed binary tree one locates the right subtree by first deciphering the left subtree. This is fine if the left subtree will be used in the same computation anyway, but in general, access to right-lying components is more expensive than to left-lying components. There are at least two ways of improving this situation. Firstly, when constructing a bit-sequence to represent a component, one can precede it with a short bit-sequence representing its length. This costs more space, but allows unneeded components to be skipped over quickly. See section 4.4 for a fuller sketch of this idea in the context of I/O. A second alternative is to choose the level of data-structure at which compression is best employed. Section 3.3 describes a judicious choice in an example application, where the entries at the leaves of a tree are compressed, but the tree structure itself remains in ordinary form. If the tree structure still takes up too much space, one might use an array of binary pointers instead.

3 Example application: a dictionary of types

For a realistic illustration of the value of runtime heap-compression of data, we have chosen a small but significant part of the *nhc* Haskell compiler and made a useful stand-alone tool from it.

When compiling a module which contains `import` declarations, *nhc* reads an interface file for each imported module. The interface file contains only type declarations, instance declarations, and function names annotated with their type. These declarations are needed for type inference in the importing module.

The interface files are stored in textual format (an issue to which we shall return in a later section), which is parsed to an internal tree-like structure for representing types. Under normal circumstances the type data is

```
data IndTree t =
    Leaf t
  | Fork Int (IndTree t) (IndTree t)

itind  :: Int -> IndTree a -> a
--   itind i it
--   returns the i'th entry from the tree it

itmap  :: (a->b)
          -> IndTree a
          -> IndTree b
--   itmap f it
--   applies the function f to every leaf in the tree it

itmapm :: (a->IO b)
          -> IndTree a
          -> IO (IndTree b)
--   itmapm f t
--   applies the monadic function f to every leaf in
--   the tree it, returning the result in the I/O monad
```

Figure 2: Index tree and operations.

retained for the type-checking phase of compilation and then discarded. However, we have re-used the interface parser in writing a browser for the type information. Interface files are read and stored in a hashed lookup structure. The user enters function names at an interactive command line, and the tool reports the types for those functions. This is a very simple database application, but the database contains recursively structured information rather than pure text.

3.1 Data structure

For the database, we use a simple indexed binary tree type `IndTree`, outlined in Figure 2. In addition to some standard tree operations, we use a version of `itmap` embedded within the I/O Monad, called `itmapm`, which both ensures that the updated tree is built before the program continues (as a way of avoiding a build-up of update-closures [10]), and also permits the mapped function to be in the I/O monad (for instance `compress`).

The entries in this tree are buckets of *(function-name, type)* pairs. We use a simple hash function on every name to produce an index into the tree. If two names collide at the same index, we store both entries in the same bucket. By choosing an appropriately sized tree, the average bucket size is small and hence a bucket can be efficiently represented just as a list.

3.2 Introducing compression

An appropriate level at which to introduce compression to this lookup structure is on buckets of entries: whilst the index structure is relatively small and accessed frequently, the buckets are individually relatively large and are accessed infrequently. Perhaps an equally good choice would be to compress only individual entries within buckets.

Figure 3 gives an outline of the main program. We omit the datatype definitions and requests for derived instances for strings, pairs, and type declaration trees. The program first builds the `IndTree` and then performs lookup on it. It differs from a compression-less program only in the definition and use of the monadic function `leafCompress`, and a single application of `expand`, each shown in bold type.

3.3 Results

We test the type-browsing tool by supplying as input the interface file for the Standard Prelude, and then requesting the type of all 260 prelude functions. We first compare speeds (running *nhc*'s byte-code interpreter on a 50MHz microSparc processor). The compressed version inevitably has a slower access rate than the standard version. We then compare memory space, finding the compressed version to be much more compact than the standard version, as expected.

Time

Without compression, it takes 24.75s to read the file, parse it, and build the index tree, and a further 11.55s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 23.1 entries returned per second, and a total computation time of 36.30s.

With compression, it takes the same 24.75 seconds to read the file, parse it, and build the initial index tree. It takes a further 25.40 seconds to compress the entries into a new tree, followed by 30.0 seconds to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.7 entries returned per second, and a total computation time of 80.15s.

Space

Without compression, the parsing stage uses a peak of 250kb, followed by a constant usage of 150kb of heap memory for the lookup stage.

With compression, the parsing stage again uses a peak of 250kb, followed by a constant usage of about 8kb of heap memory for the lookup stage, and about 16kb of off-heap bit vectors, totalling 24kb.

```
declstree ::
 Int -> [Decl TokenId]
 -> IO (IndTree
        [(String, Decl TokenId)])
leafCompress ::
 IndTree [(String, Decl TokenId)]
 -> IO
    (IndTree
      (Bin
        [(String, Decl TokenId)]))
leafCompress = itmapm compress

main =
  readFiles ".hi" >>= \inp->
  let h = chooseHashTreeSize inp in
  declstree h (parsedecls inp)
    >>= \pt->
  leafCompress pt >>= \cpt->
  browse cpt

browse cpt =
  untilCatch isEOFError
    (putStr "type browser> " >>
     getLine >>= \inp->
     mapM_
       (putStrLn.showDecl.snd)
       (map (select cpt)
            (words inp))
    )

select cpt w =
  filter
    ((==w).fst)
    (expand (itind (hash w) cpt))
```

Figure 3: Type browser tool

In this example application, the compression ratio is greater than 6x. We have studied some other applications which demonstrate a broadly similar compression ratio.

3.4 Issues raised by the example

The main problem with the program as it stands is that although the compression achieved during lookup is very worthwhile, the space profile is dominated by the requirement for a large initial heap before the data can ever be compressed. A large proportion of both time and space in the computation are being devoted to the initial parsing of the text file. The compression stage also accounts for a significant part of the time, though this is compensated by the subsequent reduction in space usage.

One might wonder whether the 250kb of space needed for parsing a data structure that turns out to need only 150kb is excessive – perhaps due to a poor choice of parser combinators? In fact the combinators used in the example, in common with all the components of *nhc*, were designed for space efficiency [8]. Even carefully crafted text parsers can cause space irregularities: the difficulty of avoiding them emphasises a need for an alternative more efficient mechanism.

A possible solution to the space problem would be to rework the program structure for greater laziness – to compress as we parse, rather than having two essentially separate passes over the data.

However, another solution addresses the time problem as well as the space problem: store the compressed data directly in a binary file. The text file is parsed and compressed once; all subsequent uses avoid this stage and load the binary representation directly. The next section describes how we have added binary file I/O to Haskell in a manner analogous to compression.

4 A Class of Types for Binary I/O

It would be very convenient to be able to perform I/O directly to/from heap memory, in order to store data structures in a file for later use by a different run of the same program, or perhaps to transmit values between two processes. The linked-graph model of the heap makes this tricky to implement however [13]. Some form of *flattening* is required before a value is amenable to storage or transmission.

The only current standard Haskell mechanism for transferring data is by conversion to and from a textual format, using the `Show` and `Read` classes. While having the benefit of readability, this approach can often be very inefficient. Good implementations of the `Read` class are rare, and programmers still frequently write custom parsers for their Haskell data. Even these, as we have seen, can be slow and memory-hungry.

Our scheme for heap data compression offers an alternative flattening operation which follows a uniform pattern and like the textual classes can be derived automatically for almost any datatype (the restrictions are noted in an earlier section). A binary I/O library based on these ideas should be much more efficient than parsing and printing text.

4.1 The programmer's view

As before, we define a type class for values which can be transmitted in binary format.

```
class BinIO a where
  put  :: BinHandle -> a -> IO ()
  get  :: BinHandle -> IO a
```

The type BinHandle is an abstract type analogous to the ordinary text-file Handle, but specific to binary files. Like compress, but unlike expand, both put and get operations return results in the I/O monad, because here we are dealing with true I/O. Also note that get does *not* take a pointer to a value as an argument. Rather, it reads values from the file sequentially, starting at the current position recorded in the state of the I/O monad.

Instances of the BinIO class are written using the primitives:

```
putBits :: BinHandle
           -> Int -> Int -> IO ()
getBits :: BinHandle
           -> Int -> IO Int
```

The intuition is that putBits h s n writes integer value n into a field of width s bits at the current position in the file denoted by h. Conversely, getBits h s reads an integer value of width s bits from the current position in the file h, returning just the value, and implicitly updating the file pointer.

As before, the explicit sequence of I/O operations performed during output or input ensures that components of a value are placed next to each other in the file, and read back in the same order.

Various other auxiliary functions are needed to complete the library, such as the operations to open and close binary files. These just mirror the existing operations in the textual I/O library.

```
openBinFile  :: FilePath -> IOMode
                 -> IO BinHandle
closeBinFile :: BinHandle -> IO ()
```

One point worth mentioning is that binary files, like textual files, are not type safe across runs. That is, one can write a value to a file as one type and read it back as another. We do not attempt to address this question (see however [7]), leaving it to the programmer to do the sensible thing.

4.2 Implementation of buffering

The buffering required for I/O on binary files is more complicated than that for textual files. Textual I/O assumes that every value is transferred as a whole number of bytes, and hence the minimum buffering unit is the byte. With binary I/O it must be possible to transfer a single bit at a time.

```
instance BinIO Bool where
  put h =
    putBits h 1 . fromEnum
  get h =
    getBits h 1 >>=
    return . toEnum

instance BinIO Tree where
  put h (Leaf b) =
    putBits h 1 0 >>
    put h b
  put h (Branch l r) =
    putBits h 1 1 >>
    put h l >>
    put h r
  get h =
    getBits h 1 >>= \i->
    case i of
      0 -> get h >>= return . Leaf
      1 -> get h >>= \l->
           get h >>= \r->
           return (Branch l r)
```

Figure 4: Binary I/O instances for truth trees.

The approach taken in our prototype implementation is to layer a second buffer on top of the standard byte-oriented buffers. This second layer consists of a single byte per file: for output, it accumulates bits until it is full, at which time it is flushed into the ordinary byte-oriented system; for input, it receives a byte from the ordinary system, which is then drained bit-by-bit into the Haskell program. We have implemented this mechanism as a Haskell module.

4.3 Truth trees revisited

Recall the binary trees of Booleans from section 2.3. Instance definitions for binary I/O, using the same bit encoding as for compression, are shown in Figure 4. As before, these can be derived automatically by the compiler.

4.4 Bit transfer between memory and files

There is a broad similarity between the Compress and BinIO classes. The same binary representation can be produced and interpreted by both, whether in memory or on file. One of our aims in developing binary I/O was to use this identity of representation to increase the efficiency of transfer. Yet the situation described so far permits only values in the standard graph-in-the-heap representation to be put into files or retrieved. If the

```
newtype SizedBin a =
        SB (Int, Bin a)

class Bin a =>
      SizedCompress a where
  sizedCompress ::
          a -> IO (SizedBin a)
  sizedExpand ::
          SizedBin a -> a
  sizedCompress v =
      compress v >>= \bv->
      primBinSize bv >>= \s->
      return SB (s,bv)
  sizedExpand (SB (s,bv)) =
      expand bv

instance BinIO (SizedBin a) where
  put h (SB (s,bv)) =
      put h s >>
      primDirectPut h s bv
  get h =
      get h >>= \s->
      primDirectGet h s >>= \bv->
      return (SB (s,bv))
```

Figure 5: Sized binary values, and efficient bulk I/O.

compressed representation is wanted in *both* memory and file, then the value must 'pass through' the standard representation, suffering the process of two very similar translations.

So we need an instance of the BinIO class for the memory-compressed Bin a types, using special primitives to implement the bulk transfer of the binary values.

```
instance BinIO (Bin a) where
  put = primDirectPut
  get = primDirectGet
```

But this scheme has a hidden difficulty. Bulk transfer can only be efficient if the size of the value is known. If the size is not known, then the transfer must remain interpretive, since Bin a values are of variable size. Fortunately, this oversight is easy to correct, at the cost of a little overhead in space. We introduce a new type of sized binary values, and extend the Compress class with a subclass SizedCompress which simply attaches size information to compressed values (see Figure 5).

No other instances of SizedCompress need ever be written – the default definitions given in the class declaration are sufficient. Our new instance of BinIO for all SizedBin a types can now implement bulk transfer correctly, by placing or reading the size of the binary value immediately before the value itself. Note that a size need only be attached to the *outermost* structure of a value: the internal structure within a value may comprise many compressed components stored without sizes.

The extra space needed for storing size information could reduce or even cancel the benefit of compression if applied to many values of only modest size. However, our intention is that a program should only attach sizes to larger compressed values involved in bulk I/O. The larger and more complex a compressed value is, the more valuable it will be to transfer it in bulk rather than interpretively, and the smaller the proportion of space taken up by the size information.

One further, less serious, space cost is associated with bulk transfer of bits. The problem is a potential misalignment of values. Imagine the file pointer is set at bit 3 and the memory compression pointer is set at bit 6, immediately before a transfer takes place. It would be possible to fix the alignment byte-by-byte during the transfer, but this would be detrimental to efficiency. The alternative is to insist that sized binary values are exactly aligned to a byte boundary. On average, this solution costs 3.5 bits per value, since between 0 and 7 extra bits are needed to pad the value to a byte boundary. This is a minor cost compared to what has already been paid to store sizes. In practice, byte-alignment is easy to implement.

4.5 An extension for random-access file I/O

A more radical way to avoid the remaining costs of bulk I/O and memory storage for large data structures is to compute with the data structure held entirely (in compressed form) in a file. This is a common technique in the database world. The main requirement over and above what is already available is the need for random-access to files of compressed data.

We achieve this by an extension to the binary I/O class:

```
class BinIO a =>
      RandomAccessBinIO a where
  putAt :: BinHandle
           -> a -> IO (FilePtr a)
  getAt :: BinHandle
           -> FilePtr a -> IO a
```

The intuition for the new operations is that putAt returns a pointer to the start location of the value, and getAt uses such a pointer to find the value. Values of the new abstract type FilePtr a do not refer specifically to any file. At the implementation level, they contain only a byte offset from the start of the file and a bit offset within the referenced byte. It is up to the

```
main =
  readFiles ".hi" >>= \inp->
  if parsingText then
    let h= chooseHashTreeSize inp
    declstree h (parsedecls inp)
      >>= \pt->
    leafCompress pt   >>= \cpt->
    sizedCompress cpt >>= \ccpt->
    openBinFile "db.dat" WriteMode
      >>= \db->
    put db ccpt >>
    closeBinFile db
  else return ()
  >>
  openBinFile "db.dat" ReadMode
    >>= \db->
  get db >>= \ccpt->
  closeBinFile db >>
  let cpt = sizedExpand ccpt in
  browse cpt
```

Figure 6: Direct binary transfer to memory.

programmer to use these file pointers sensibly on the correct files.

One difficult issue is whether it should be permissible to read and write the same random-access binary file. The possibility of intermixed read and write access does complicate the implementation of buffering still further. For the moment, we disallow the possibility, not on point of principle, but to keep things simple.

5 Type dictionary revisited

We return to the type-browser tool of section 3 to illustrate the facilities of binary I/O. Here are two new versions of the tool which use file I/O.

5.1 Direct binary transfer to memory

In the first version of the type tool, not only did textual parsing take a long time, but compression took a similar amount of time. We can eliminate both of these stages of the computation by storing the compressed data structure in a file between program runs. Figure 6 shows the additions to the original program in order to store the entire data structure into a file, and to reload it in subsequent runs.

5.2 Layered compression

One point worth noting about this example is the type of the value stored in the file.

```
ccpt ::
    SizedBin
      (IndTree
        (Bin
          [(String, Decl TokenId)]))
```

Not only are the entries at the leaves of the index tree compressed, but the whole of the index tree itself is compressed too. What happens when a compressed value is compressed again by virtue of residing within another value? The bit-vector is simply copied in-line, without modification. When the outer value is expanded, the inner compressed values simply remain in compressed form, true to the type signature. So in this example, because there are two stages of compression, there are also two stages of expansion. The result of `sizedExpand ccpt` is a tree with a branch structure in the heap but leaves of compressed entries.

5.3 Indexed binary files

So far the indexed data structure containing compressed entries has been loaded (one way or another) into memory. We now illustrate how both the compressed entries and the indexing structure may be stored in files. The intention here is to store data entries in one file, but the file-pointers which reference the data file in a different index file. The index file is flat, containing just a sequence of file pointers. Lookup proceeds as follows: first apply the hash function to the string key, giving an integer n; now read the n'th entry from the index file; and finally use this value as a pointer into the data file to retrieve the true compressed entry. This is a very similar mechanism to that used in many previous systems, for example, the *gdbm* library of C database routines.

Figure 7 shows the version of the program using indexed files. We gloss over the issue of file-pointer arithmetic here, which is easy to program but tedious to read. Pointer arithmetic is needed simply to allow the n'th value of compressed size s to be read from the index file – that is, starting at the $(n-1)*s$'th bit.

5.4 Results

To test the new versions of the type-browsing tool we again supply as input the interface file for the Standard Prelude, and request the types of all 260 prelude functions.

```
treeIO ::
    BinHandle -> BinHandle
    -> IndTree
           [(String, Decl TokenId)]
    -> IO ()
treeIO datf indf =
  itmapm
    (\v->
      sizedCompress v >>= \bv->
      putAt datf bv   >>= \fp->
      put indf fp)

main =
  ...
  treeIO datf indf pt >>
  ...
  browse datf indf

browse datf indf =
  untilCatch isEOFError
    (putStr "type browser> " >>
     getLine >>= \inp->
     mapM_
       (\w->
         select datf indf w >>=
         mapM_
           (putStrLn.showDecl.snd)
       ) (words inp)
    )

select datf indf w =
  getAt indf (hash w) >>= \fp->
  getAt datf fp >>= \e->
  return (filter ((==w).fst)
                 (sizedExpand e))
```

Figure 7: Indexed binary files.

Time

Reading the entire structure from binary file into memory, it takes 0.20s to read the file and perform a first-stage decompression, and a further 30.24s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.6 entries returned per second, and a total computation time of 30.44s.

When the index structure is stored in and read from two files, it takes 36.74s to read the text, parse it, build the initial index tree, and write it out to the new files, then a further 32.01s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.1 entries returned per second, and a total computation time of 68.75s.

initial I/O	time (s)		space (kb)	
+ data repn.	set-up	queries	set-up	queries
text+heap	24.75	11.55	250	150
text+bits	50.13	30.01	250	8+16
binfile+bits	0.20	30.24	10	8+16
text+binfile	36.74	32.01	240	150
none+binfile	0.09	32.01	4	2

Table 1: Time and space costs for different versions of the type-dictionary program when the type of every function in the prelude is requested.

When the index structure is kept only in the pre-computed files, it takes 0.09s to prepare for reading, and a further 32.01s to retrieve and display all the entries in turn. This gives a rate (discounting initialisation time) of 8.1 entries returned per second, and a total computation time of 32.10s.

Space

When the entire structure is read from a binary file into memory, there is no initial parsing stage. Memory usage peaks at 10kb of heap, and averages at 8kb of heap, plus a constant 16kb of off-heap bit vectors.

When the index structure is kept only in the pre-computed files, again there is no parsing stage. Memory usage peaks at 4.5kb of heap, and averages at 2.5kb of heap. There no bit vectors in memory. The files occupy 16kb (data) and 408 bytes (index).

A complete comparison with the earlier versions of the tool is shown in Table 1. (All figures are for *nhc*'s byte-code interpreter measured on a 50Mhz microSparc processor; we achieve nearly identical speed ratios on a newer and faster 150Mhz R4000. The compression ratio of course remains constant across platforms.) It can be seen clearly that once the initial work has been done to store a data structure as a binary file, it is much quicker to read the binary file than to re-parse the textual equivalent. Access to individual entries is slower, because the work of interpretation has been deferred until the moment of expansion. Even so, a complete traversal of the compressed structure takes less time than the original version of the tool took to parse a text file and then traverse the lookup structure.

6 Related work

The Haskell language definition, prior to version 1.3, had a `Binary` class with two methods, `showBin` and `readBin`, converting data to and from a `Bin` datatype – the intended use was primarily for binary file I/O. We know of no implementations which supported the class, and the idea has now been dropped from the language standard. Two improvements on the previous design, introduced by our classes, are: parameterisation of `Bin` on the type being represented; and the choice of either interpretive I/O direct between uncompressed values in memory and a binary representation on disk (avoiding an intermediate binary representation in memory), or fast transfer of the binary representation.

The *hbc* compiler has a library which defines a class for `Native` conversions. The methods convert between a value and a list of bytes, which can then be used in textual I/O. There are three differences from our compression class: the data representation is byte-oriented, rather than bit-oriented; the byte vectors are otherwise untyped; and there is no use of monadic sequencing to control the evaluation order of conversion. The intended use of the `Native` class is for foreign language interfaces, and data transmission between processes (either via the file system or across a network). Our scheme permits a flexible style of data compression in addition to these applications.

Johan Jeuring is working on a *polytypic* scheme for data compression [3]. There is a close correspondence between polytypic programming and type classes. (Jansson and Jeuring's language system *PolyP* [2] is the first to provide facilities for polytypism.) His method separates a value's structure from its content, compressing the structural component in a very similar manner to our scheme, and then relying on standard textual methods to compress the content.

There is of course a wide literature on compression algorithms – see for example the comprehensive survey by Lelewer and Hirschberg [5].

Other work on reducing the amount of space used for data representation in functional languages (although without true compression) includes a significant body of work on *unboxing*; see for instance [6].

7 Conclusions and Future Work

A declarative approach to bit-level data representation opens up whole new application areas for functional languages. We have implemented mechanisms for the following types of computation, and demonstrated examples of their use:

1. Computing with a large data structure held in compressed form in memory.
2. Storage and retrieval of a large data structure to/from a binary file, with full representation in memory.
3. Computing with a large data structure held in compressed form in memory, and its bulk storage and retrieval to/from a binary file.
4. Computing with a large indexed data structure held entirely in files, not in memory.

The compression scheme presented in this paper can be derived automatically for almost all datatypes, following a standard pattern which gives significant space savings. However, it also leaves the programmer free to try more aggressive compression algorithms, by defining custom instances of the `Compress` class.

We suggest that typically, the use of heap compression will be considered by a programmer once the program is complete and has been profiled to identify and correct any space faults [11]. When the profile reveals that there is still a large amount of data occupying the heap, and that this data really is needed, the opportunity to compress it should be taken.

Compression and binary I/O could improve the performance of Haskell compilers. Recall that the type browser tool initially reads machine-generated interface files produced by *nhc*. Röjemo has found through profiling that a significant portion of the time taken to separately-compile a module is spent in reading interface files for imported modules. If the interface files were stored as binary files rather than textually, we conjecture that the compiler would run consistently faster. For human readability, a short and fast program to translate the binary format to text could be provided.

Many other applications could benefit from the ability to manipulate values in compact and binary representations, particularly those where it is desirable to hold a very large amount of information either in main memory or on secondary storage. Examples include databases, natural-language processing, and image processing. One application we intend to study fully is a tracer/debugger for Haskell programs [12]. It is very difficult to construct a complete trace of a large computation because of the huge amount of space required. A compressed store makes the task more feasible, especially if the compressed data structures can be stored progressively into a random-access file.

Other uses of bit-vector representations that we have not yet had time to explore include the description of machine registers for embedded-systems control, and the marshalling of data for foreign-language interfaces and inter-process communication. It could be argued that these latter applications are word- or byte-oriented rather than bit-oriented. However, the flexibility of representing data components at the finest granularity remains useful – for instance consider storing a Huffman-coded string inside an aligned block of bytes for transmission across a network.

One valid criticism of our compression class is that it uses the I/O monad, and so the description of binary representations is somewhat imperative. Since compression is not really doing proper input or output, a different choice, though still retaining the imperative style, would be to use a state-transformer monad. (However, the use of the I/O monad does suggest an opportunity for convergence between compression and binary I/O; see below). The further possibility of a more declarative description of representations deserves some investigation. Such a description would perhaps use basic combinators for juxtaposition (placing two bit vectors side-by-side), alignment, padding, trimming, and so on. One difficulty would be the treatment of strictness – when to force the compression of values.

A second criticism of our compression and binary I/O classes is that if instances are hand-written, they must define functions for conversion in *both* directions. This opens the possibility of errors where the value created by one operation is not correctly read back by the other. The two operations are in some sense inverses of each other. Perhaps a facility to define a single description of the binary representation should be provided, from which both conversion operations could be derived.

This also raises the issue of convergence between the two classes. Binary I/O and compression are very similar, and by default use the same representations. However, the possibility of error is again introduced because custom instances written by hand may fail to match each other. To what extent could the operations be merged into a single class, perhaps by regarding the off-heap bit-vector memory as just a special sort of file? The closest resemblance exists between the classes `RandomAccessBinIO` and `Compress`, yet there is still at least one important difference between them, namely that `expand` is a pure function while `getAt` is in the I/O monad.

In our implementation we chose to store bit vectors out of the heap, in a separate area of memory, without garbage collection. As noted above, this lends itself to the treatment of bit-space as just a special kind of file. However, a different implementation could allocate space for bit vectors in the heap, allowing full garbage-collection of compressed values. This would be attractive for some applications, for instance a network server which compresses packets of data for transmission but then discards them. A heap-based implementation of bit vectors would cost a small amount of extra space for tagging the compressed values, and we conjecture that it could cost a significant amount of speed too. We intend to develop this alternative design for comparison with the current system.

Finally, there is a distant possibility that data compression and expansion could be applied to parts of the heap automatically. Where in our scheme the programmer must judiciously select data structures and apply the compression and expansion functions textually, perhaps in future the memory management system will be able to identify long-lived portions of the heap and transparently compress them during garbage collection, allowing re-expansion lazily by need.

Acknowledgements

Thanks are due to Canon Research Centre Europe Ltd., who have wholly funded this work. We also acknowledge the referees' very full and helpful comments on the first draft of this paper.

References

[1] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[2] P. Jansson and J. Jeuring. Polyp – a polytypic programming language extension. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 470–482, Paris, January 1997. ACM Press.

[3] J. Jeuring. Polytypic data compression. *In preparation*, 1997.

[4] M. P. Jones. The implementation of the Gofer functional programming system. Technical report, Department of Computer Science, Yale University, May 1994.

[5] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, September 1987.

[6] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Proc. 5th ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'91)*, pages 636–666, Cambridge, MA, August 1991. Springer LNCS 523.

[7] M. Pil. First class file I/O. In W. Kluge, editor, *Proc. 8th Intl. Workshop on Implementation of Functional Languages (IFL'96)*, pages 341–350, Bonn, Germany, September 1996. Institute of Computer Science, Christian-Albrechts-University, Kiel.

[8] N. Röjemo. *Garbage collection and memory efficiency in lazy functional languages.* PhD thesis, Department of Computer Science, Chalmers University of Technology, Sweden, 1995.

[9] N. Röjemo. Highlights from nhc - a space efficient haskell compiler. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292, La Jolla, June 1995. ACM Press.

[10] C. Runciman. A virtual terminal. In C. Runciman and D. Wakeling, editors, *Applications of functional programming*, pages 60–73. UCL Press, 1995.

[11] C. Runciman and N. Röjemo. Heap profiling for space efficiency. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Intl. School on Advanced Functional Programming*, pages 159–183, Olympia, WA, August 1996. Springer LNCS Vol. 1129.

[12] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. *submitted to PLILP'97*, Dept. of Computer Science, University of York, UK, April 1997.

[13] I. Toyn and A.J. Dix. Efficient binary transfer of pointer structures. *Software — practice and experience*, 24(11):1001–23, 1994.

[14] D. Wakeling. A throw-away compiler for a lazy functional language. In M. Takeichi and T. Ida, editors, *Fuji Intl. Workshop on Functional and Logic Programming*, pages 287–300, Susono, Japan, July 1995. World Scientific.

[15] M. Wallace. *Functional programming and embedded systems.* DPhil Thesis YCST 95/04, Department of Computer Science, University of York, January 1995.

[16] M. Wallace and C. Runciman. Lambdas in the Liftshaft — functional programming and an embedded architecture. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 249–258, La Jolla, June 1995. ACM Press.