JFP **14** (5): 489–502, September 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796802004471 Printed in the United Kingdom

FUNCTIONAL PEARL

Global variables in Haskell

JOHN HUGHES
Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden
(*e-mail:* `rjmh@cs.chalmers.se`)

Abstract

Haskell today provides good support not only for a functional programming style, but also for an imperative one. Elements of imperative programming are needed in applications such as web servers, or to provide efficient implementations of well-known algorithms, such as many graph algorithms. However, one element of imperative programming, the *global variable*, is surprisingly hard to emulate in Haskell. We discuss several existing methods, none of which is really satisfactory, and finally propose a new approach based on implicit parameters. This approach is simple, safe, and efficient, although it does reveal weaknesses in Haskell's present type system.

1 Introduction

Simon Peyton-Jones calls Haskell (Peyton-Jones *et al.*, 1999) "the world's finest imperative programming language" (Peyton-Jones, 2001) – but is that really true? Something which is very easy in imperative languages is to declare a global variable, and then refer to it and update it from anywhere in the program. Global variables are useful: they may refer to long-lived imperative data structures, such as hash tables, graphics contexts, priority queues, and so on, enabling the programmer to refer to or modify these structures anywhere in the program, without needing to pass a plethora of pointers as parameters. Yet Haskell offers no well-known and really satisfactory way of creating them. In this paper, we review various ways this has been done, and finally present the solution we favour. We shall use the simple example of a global queue to illustrate the methods we discuss.

1.1 Imperative programming in Haskell

First we briefly review the way Haskell supports imperative programming. All operations with side-effects are assigned *monadic types* of the form *m a*, where *a* is the type of the result, and the monad *m* determines the kind of side effects which are possible. For example, operations which perform input/output are assigned types of the form IO *a*. The *readFile* function has the type String → IO String: given a String (filename), it performs input/output and delivers a String (the file contents). We think of a value of type *m a* as an *m-computation delivering an a.*

Many monad types are predefined in Haskell, and new ones can be introduced, but every monad must support the operations *return* and 'bind' ($\gg=$), via an instance of the following class:

$$\begin{array}{l} \textbf{class}\ \mathsf{Monad}\ m\ \textbf{where} \\ \quad \begin{array}{lcl} \mathit{return} & :: & a \to m\ a \\ (\gg=) & :: & m\ a \to (a \to m\ b) \to m\ b \end{array} \end{array}$$

The *return* operation constructs a computation which delivers its argument as the result, while $\gg=$ sequences two computations. Haskell provides syntactic sugar for $\gg=$ via the **do** syntax, for example,

$$\begin{array}{l} \textbf{do}\ \ s \leftarrow \mathit{readFile}\ \text{“foo”} \\ \qquad \mathit{writeFile}\ \text{“baz”}\ s \end{array}$$

which binds *s* to the result of *readFile* in the remainder of the **do**.

These operations do not enable us to extract an *a* from an *m a*, so once we have used a monadic operation, everything built from it will carry a monadic type, thus recording (and sequencing) the use of side-effects. The top-level of a Haskell program has type IO (), thus permitting (sequenced) input/output at the top-level only.

Input/output is the only true form of side-effects permitted by the Haskell 98 standard, but most implementations support more general imperative programming, by providing reference types and functions to read and write them (Launchbury & Peyton-Jones, 1995). The type IORef *a* contains references which can be read and written by operations in the IO monad, thus sequencing these side-effects along with input/output operations. The operations on them are

$$\begin{array}{lcl} \mathit{newIORef} & :: & a \to \mathsf{IO}\ (\mathsf{IORef}\ a) \\ \mathit{readIORef} & :: & \mathsf{IORef}\ a \to \mathsf{IO}\ a \\ \mathit{writeIORef} & :: & \mathsf{IORef}\ a \to a \to \mathsf{IO}\ () \end{array}$$

which create, read, and write references, respectively.

Haskell also provides a monad ST of 'encapsulated' side-effects, which can be used to define pure (non-monadic) functions with an internal imperative implementation. It is guaranteed that the encapsulated side-effects are invisible from outside. The encapsulation function is

$$\mathit{runST} :: (\forall s.\mathsf{ST}\ s\ a) \to a$$

Here the argument of type ST *s a* is the encapsulated imperative computation, and *s* is a 'state thread identifier', which as well as appearing in the type of the computation, also appears in the types of the references, and the reference creation, reading and writing operations:

$$\begin{array}{lcl} \mathit{newSTRef} & :: & a \to \mathsf{ST}\ s\ (\mathsf{STRef}\ s\ a) \\ \mathit{readSTRef} & :: & \mathsf{STRef}\ s\ a \to \mathsf{ST}\ a \\ \mathit{writeSTRef} & :: & \mathsf{STRef}\ s\ a \to a \to \mathsf{ST}\ s\ () \end{array}$$

Note that the *same s* appears everywhere, so that these types force all the references read, written, or created in a computation of type ST *s a* to carry the same state thread identifier *s*. Now, the rank-2 polymorphism in the type of *runST* demands that the encapsulated computation can be run in *any* state thread *s* – it must be polymorphic – which guarantees in turn that that it cannot have any effect on references created elsewhere.

Our running example will use a Queue module which represents queues in the standard imperative way, as a pair of pointers to the front and back of the queue, adding elements in constant time by overwriting the pointer in the last cell. We could choose to implement such a module using either the IO or the ST monad, but ideally, the queue implementation should be useable with either one. We shall achieve this by using Haskell's overloading: we define a class of monads with references

```
class Monad m ⇒ RefMonad m r | m → r where
    newRef   :: a → m(r a)
    readRef  :: r a → m a
    writeRef :: r a → a → m ()
```

This declares that a monad *m* is a 'RefMonad', with references of type *r*, if it supports the given operations. The '| *m* → *r*' is a *functional dependency* (Jones, 2000), which declares that the monad type determines the reference type – IO determines IORef, ST *s* determines STRef *s*. This is important information for resolving ambiguity during type inference.

Now we can declare IO and ST to be RefMonads as follows:

```
instance RefMonad IO IORef where
    newRef   = newIORef
    readRef  = readIORef
    writeRef = writeIORef

instance RefMonad (ST s) (STRef s) where
    newRef   = newSTRef
    readRef  = readSTRef
    writeRef = writeSTRef
```

Using these overloaded operations, we define a queue package which works with either monad, providing the following operations. Note that the Queue type must be parameterised on the reference type it uses.

```
data Queue r a = ...                         — queue of a using
                                             — references of type r
empty    :: RefMonad m r ⇒                   — create a new queue
              m (Queue r a)
add      :: RefMonad m r ⇒                   — add an element
              a → Queue r a → m ()
remove   :: RefMonad m r ⇒                   — remove an element
              Queue r a → m ()
front    :: RefMonad m r ⇒                   — the front element
              Queue r a → m a
isEmpty  :: RefMonad m r ⇒                   — test for empty
              Queue r a → m Bool
```

We omit the implementations of these operations, which are standard.

We have now implemented queues imperatively, but our goal in this paper is to implement a *single, global* queue. This we have not done: to use the operations above, we must create a queue and bind it to a variable, then pass it explicitly to every function which uses it. In the following sections we will examine various ways of avoiding this.

2 Using *unsafePerformIO*

We aim to define functions *addG*, *removeG*, etc., which refer to a single global queue, and thus do not need to take a queue as a parameter. The natural way to do so is to declare the global queue in the same scope that these functions are declared – that is, via a top-level declaration. But, using the primitives above, there is no way to bind a Queue to a top-level variable! The reason is that a Queue contains references, and reference creation has a monadic type, such as IO (IORef *a*). Now the only way to bind a variable to the reference itself is using (>>=) (or the **do** syntactic sugar), but such a variable is bound by a λ-expression, and not at top level. There is no top-level expression of type IORef *a*, and so no top-level expression of type Queue IORef *a* either.

It might seem that *runST* enables us to create a top-level STRef using

```
runST (newSTRef a)
```

but recall that the purpose of *runST* is to *encapsulate* side effects. Since the state thread variable appears in the type of the reference created, and would thus appear in the type of *runST*'s result, it cannot be bound in *runST*'s argument, as the rank-2 type requires. So this expression is rejected by the type-checker.

The solution presented by Peyton-Jones (2001) is to use a new primitive

```
unsafePerformIO :: IO a → a
```

which solves the problem of extracting the reference returned by *newIORef* from its monadic type. We can now define a global queue by

```
globalQ :: Queue IORef a
globalQ = unsafePerformIO empty
```

and go on to define

```
addG :: a → IO ()
addG a = add a globalQ
```

and so on. Peyton-Jones reports that this is one of three 'very common' uses of *unsafePerformIO*.

What is wrong with this solution? Well, one objection is that it only works for the IO monad – there is no *unsafePerformST* (although GHC (ghc, n.d.) permits an ST value to be converted to an IO value, thus providing a somewhat roundabout way to achieve the same effect). However, the biggest objection is that *unsafePerformIO* is – well – unsafe! It clearly violates the property that monads are designed to ensure, that computations with side effects have monadic types. Peyton-Jones says:

"*unsafePerformIO* is a dangerous weapon, and I advise you against using it extensively. *unsafePerformIO* is best regarded as a tool for systems programmers and library writers, rather than for casual programmers. . . . you need to know what you are doing."
Such a function should not be used for such a common task as defining a global variable!

It is worth pointing out that the dangers of using *unsafePerformIO* strike in this very example. It is well known that unrestricted references and assignment make the Hindley–Milner type system unsound (Tofte, 1990). To restore soundness ML (Milner *et al.*, 1997) imposes the 'value restriction' on bindings, and Haskell assigns these operations monadic types. Using *unsafePerformIO* circumvents these restrictions and makes the type system unsound.

Look back at the definition of *globalQ* again. Notice it is declared with a *polymorphic* type – the one the type checker would infer. Consequently *addG* is also polymorphic, and can be used to *add elements of different types to the same global queue*. The *frontG* operation is also polymorphic, which allows an element to be removed *with a different type from the one it was added with*! Clearly this may lead to run-time type errors. To avoid this, the programmer *must* declare global variables with a completely *monomorphic* type. It is dangerous, to say the least, to expect that programmers will always do so.

3 A queue monad

If we cannot bind a top-level variable to the global queue, perhaps we can at least make the process of passing it as a parameter less painful. A well established way to do so is to define another monad (a 'reader' monad), in which the queue is passed as an extra parameter to every computation (Wadler, 1995). Let us define

```
newtype QMonad m r a b = QMonad (Queue r a → m b)
```

to introduce a new monad, parameterised on an underlying monad *m* with references of type *r*. (This **newtype** declaration defines a new type QMonad, with constructor also called QMonad, isomorphic to Queue *r a* → *m b*). The monad operators just pass the global queue everywhere:

```
instance Monad m ⇒ Monad (QMonad m r a) where
    return b          =  QMonad (λq → return b)
    QMonad f >>= g    =  QMonad (λq → do  b ← f q
                                           let QMonad h = g b
                                           h q)
```

Then we can define the global queue operations just by

```
addG n      =  QMonad (add n)
removeG     =  QMonad remove
frontG      =  QMonad front
isEmptyG    =  QMonad isEmpty
```

with types of the form

```
addG :: RefMonad m r ⇒ a → QMonad m r a ()
```

We also need a function to extract a result from a QMonad. We define *withQueue* to do so, returning a computation in the underlying monad *m*. This is a natural place to initialise the queue to be empty:

```
withQueue :: RefMonad m r ⇒ QMonad m r a b → m b
withQueue (QMonad f) = do q ← empty
                          f q
```

Now we can use the global queue operations in programs which need not mention a queue explicitly, such as

```
runST (withQueue (do addG 1
                     addG 2
                     removeG
                     frontG))
```

which evaluates to 2.

This approach brings us the benefits of global variables, without their disadvantages: we need not pass the queue explicitly any more, but even so we can use *more than one instance* of the queue library without risking interference. Each application of *withQueue* creates a separate queue, which enclosed uses of the queue operations implicitly refer to.

However, suppose we want to use *both* a global queue, and a global hash table, in the same piece of code? With this approach, we would also have defined a HashMonad to pass a pointer to the hash table implicitly. Now, to use the global queue operations, we must work in the QMonad, while to use the hash table operations we must work in the HashMonad. We cannot combine computations in different monads – the result would be a type error. Therefore we cannot use a queue and a hash table together.

Of course, we could define a QHashMonad to pass *both* global variables together – but this is not an acceptable solution! We naturally expect queues and hash tables to be implemented by *independent* libraries; for them to share a monad would break the abstraction barrier between them. They would have to be written together, and should really then be regarded as a *single* library, with a single global state containing both a queue and a hash table.

It is unreasonable to be limited to using a *single* global variable at a time, which severely limits the usefulness of the technique in this section.

4 A queue monad transformer

The astute reader will have noticed that Queue defined above is not only a monad, it is a *monad transformer* (Liang *et al.*, 1995). A monad transformer is a monad parameterised on another monad, such that computations in the underlying monad can be 'lifted' to the new one. We demonstrate that Queue is indeed a monad transformer by defining this lifting[1]

[1] It is possible to overload the *lift* operation, by defining a MonadTransformer class, but this would force us to give the parameters to the Queue type in a different order, which I find less 'natural'.

```
liftQ :: m b → QMonad m r a b
liftQ m = QMonad (λq → m)
```

Now, suppose we have also defined a monad transformer HashMonad *m r k v a*, where *m* and *r* are the underlying monad and reference types, *k* and *v* are the types of the keys and values in the global hash table, and *a* is the result type as usual. We cannot combine HashMonad and QMonad values with the same underlying monad, but we *can* form a *combined* monad

```
HashMonad (QMonad m r a) r k v
```

which passes both a hash table and a queue as parameters to each computation.

In principle, computations of this type can manipulate both a global queue and a global hash table. In practice, there is a little more to do before that is possible.

First, the hash table operations are defined in terms of reference operations *in the underlying monad*. Originally we expected this to be IO or ST, but now the underlying monad is the Queue monad. We must therefore provide implementations of the reference operations in the Queue monad:

```
instance RefMonad m r ⇒ RefMonad (QMonad m r a) r where
    newRef a       =  liftQ (newRef a)
    readRef r      =  liftQ (readRef r)
    writeRef r a   =  liftQ (writeRef r a)
```

Secondly, the global queue operations are defined only for the Queue monad itself. The combined monad is a HashMonad with an underlying QMonad. To use the queue operations with this type also, we must overload them. We therefore define a new class

```
class IsQMonad m a | m → a where
    addG       ::  a → m ()
    removeG    ::  m ()
    frontG     ::  m a
    isEmptyG   ::  m Bool
```

and declare both QMonad and HashMonad to be instances:

```
instance RefMonad m r ⇒ IsQMonad (QMonad m r a) a where
    . . . same definitions as before. . .

instance IsQMonad m a ⇒ IsQMonad (HashMonad m r k v) a where
    addG n     =  liftHash (addG n)
    removeG    =  liftHash removeG
    frontG     =  liftHash frontG
    isEmptyG   =  liftHash isEmptyG
```

With these definitions, we can mix queue operations, hash table operations, and (provided we implement RefMonad for HashMonad too) ordinary reference operations in the same code. We have achieved all we set out to do.

But at what cost? Suppose we write *N* libraries, each implementing an imperative data structure, and each defining its own monad transformer to provide access to its global variables. With this approach, the operations in each library must be declared in a class, and each monad transformer must be made into an instance of RefMonad, so that the others can build on top of it. However, most seriously, every monad transformer must be made into an instance of each library's class, so that whichever library's monad is outermost, all the libraries' operations will be available. Thus we must write $O(N^2)$ instance declarations. Sadly, this approach does not scale.

In principle, we could avoid the quadratic growth in the number of instance declarations by defining a class MonadTransformer $t\ m$, with an operation

$$lift :: m\ a \rightarrow t\ m\ a$$

corresponding to *liftQ*, *liftHash*, and so on. That would permit us to write a *single* instance declaration

$$
\begin{array}{l}
\textbf{instance}\ (\mathsf{IsQMonad}\ m\ a, \mathsf{MonadTransformer}\ t\ m) \Rightarrow \mathsf{IsQMonad}\ (t\ m)\ a\ \textbf{where} \\
\quad \begin{array}{lcl}
addG\ n & = & lift\ (addG\ n) \\
removeG & = & lift\ removeG \\
frontG & = & lift\ frontG \\
isEmptyG & = & lift\ isEmptyG
\end{array}
\end{array}
$$

to lift the queue operations through *every* other monad transformer. The problem with this approach is that this single instance declaration *overlaps* with the instance for the QMonad itself. It is critical, then, that the compiler choose the right instance in each case. For this reason, overlapping instances are somewhat dangerous, and indeed Peyton-Jones, Jones and Meijer concluded they should be prohibited in their exploration of Haskell's design space (Peyton-Jones *et al.*, 1997), so we prefer to avoid using them[2].

Note that there is no ambiguity if we avoid this overlapping instance declaration: a type such as QMonad (HashMonad (QMonad m)) a is an instance of both IsQMonad and IsHashMonad, but the queue operations refer unambiguously to the outermost QMonad, since of course we do not define an instance

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{IsQMonad}\ m\ a \Rightarrow \mathsf{IsQMonad}\ (\mathsf{QMonad}\ m)\ a\ \textbf{where} \\
\quad \ldots
\end{array}
$$

to lift *other* queue operations from the underlying monad to the QMonad.

5 Using implicit parameters

Our goal in the last two sections has been to pass the global queue *implicitly* to each function that uses it. We have seen that this is hard to do satisfactorily using monads. Yet a recent extension to Haskell provides implicit parameters directly (Lewis *et al.*, 2000), and they have been implemented in both GHC and Hugs (hug, n.d.) – let us try using them.

[2] At the time of writing, the current version of GHC chose the *wrong* instance, and the Hugs type-checker reported non-existent errors involving functional dependencies. This illustrates the practical risks of depending on deprecated and little used features!

An implicit parameter is referred to via a name beginning with '?', for example *?queue*. Such names are simply used, without declaration, in the functions which require them. Thus we can define

```
addG a = add a ?queue
removeG = remove ?queue
frontG = front ?queue
isEmptyG = isEmpty ?queue
```

Use of implicit parameters is recorded in these functions' types. For example, the *addG* function has the type

```
addG :: (RefMonad m r, ?queue :: Queue r a) ⇒ a → m ()
```

The presence and type of the implicit parameter is recorded in the *context* of the type, just like class contraints on polymorphic type variables. A caller need not explicitly provide a value for *?queue* – instead, callers inherit the use of the implicit parameter. For example, we can define

```
test  ::  (RefMonad m r, ?queue :: Queue r Int) ⇒ m Int
test  =   do addG 1
             addG 2
             removeG
             frontG
```

The type checker infers that *test* needs the implicit parameter *?queue*, and that it should be passed on to the queue operations. It also ensures that implicit parameters are always used with the same type, and so attempts to add elements of two different types to the global queue would lead to a type error in the definition of *test*.

Implicit parameters are bound using the '**with**' construction. For example, we can define *withQueue* as follows:

```
withQueue :: RefMonad m r ⇒ (?queue :: Queue r a ⇒ m b) → m b
withQueue m = do q ← empty
                 m with ?queue = q
```

which binds *?queue* to the newly created empty queue in *m*.

Notice the type of *withQueue*: its argument depends upon an implicit parameter, while its result does not. This type signature must be stated explicitly, since it is not 'rank 1' (Peyton-Jones, n.d.). We might call it a 'rank 2 qualified type', since a qualifier (*?queue* :: Queue *r a*) appears under a function arrow, but it is not rank 2 polymorphic since there is no **forall**. Such rank 2 qualified types are entirely useless except in the presence of implicit parameters, since without polymorphism, a class constraint *C a* qualifying a parameter's type can be resolved in only one way – and thus can equally well be resolved at the top-level. It is the possibility of binding implicit parameters to different values of the same type which has made rank 2 qualified types interesting[3].

[3] Perhaps this is why, at the time of writing, the definition of *withQueue* above was wrongly rejected due to (different) bugs in the type-checkers of both Hugs and GHC, even though both support rank 2 polymorphism!

With the implicit parameter approach, only one monad is required, and thus we can freely mix functions which refer to a global queue with those which refer to a global hash table, and so on. There is no need to 'lift' operations from one monad to another. The type-checker infers which parameters are required where, and implicitly generates code to pass them where necessary. Provided we bind them eventually, by invoking *withQueue* and similar operations, everything works without problem.

The only disadvantage of this approach is that function types become more complicated, because of the references to the implicit parameters. These parameters are implicit, as far as the code itself is concerned, but *explicit* where types are concerned. This is not so serious if the programmer allows the type-checker to infer most type signatures, but could be a major problem if many type signatures are given explicitly. Small changes to code which uses global variables could force the programmer to change a large number of type signatures, making maintenance more difficult. However, explicit type signatures are normally unnecessary in Haskell, so by choosing not to state them, programmers can largely avoid this problem. It is an *advantage* that type inference can tell us exactly which global variables each function uses – information which is not readily available at all in an imperative language.

The most awkward problem is raised by Haskell's *monomorphism restriction*, which requires a type signature on variable bindings with a non-empty context. Thus every variable definition involving an implicit parameter requires a type signature. For example, the type signature on the definition of *test* above is not optional. However, the monomorphism restriction does not apply to function definitions, and so we can avoid the need for a type signature just by redefining *test* as folllows:

```
test () = do addG 1
             addG 2
             removeG
             frontG
```

It is not a great sacrifice to adopt this style when using implicit parameters.

6 Performance

Does the choice of global variable representation have a significant impact on performance? This is a difficult question to answer in general, since it depends upon many factors such as the number of global variables, the dynamic and static frequency with which they are used, perhaps even exactly where they are used. In small experiments with very global-variable-intensive programs, we found that the implicit parameter approach performed best, *unsafePerformIO* was almost 10% slower, and the two monadic approaches were up to 30% slower than using implicit parameters. These figures would no doubt vary somewhat in real programs, and probably exaggerate the performance differences. The conclusion we draw is only that, when choosing between *unsafePerformIO* and the implicit parameter method we advocate, there is no *a priori* reason to think that the better software engineering properties of implicit parameters are accompanied by a significant performance cost.

7 Language design issues

The style we advocate in this paper does reveal weaknesses in Haskell's type system. It is distressing that, to write maintainable code, programmers must *omit* type signatures: many like to include them, as documentation checked by the compiler, or even as a partial specification of functions yet to be written. Haskell should really offer more flexibility here, so that type signatures can be included without hindering maintenance.

One way to do so would be to allow partial specification of contexts. One might write

```
test :: (RefMonad m r,...) => m Int
```

to indicate that there may be further constraints in the context of *test*. This permits programmers to specify the *type* of *test*, without fixing all of its global variables (or other constraints on its type variables). If the definition is later changed to use additional globals, then the type signature need not be. Alternatively, one might name parts of a context, and just use the name in function type signatures, thus allowing a new global variable to be added by a change in just one place. It is not clear what the trade-offs are in this design space, but it is clear that there is a problem to be solved.

Reasoning about implicit parameters can sometimes be surprisingly subtle. For example, suppose that within a computation which uses a global queue, we need a subcomputation which uses its own queue – so that we need to manipulate two queues simultaneously. For example, suppose we write

```
do ...
   addG 1
   ...
   withQueue $ do ...
                  addG 2
                  ...
```

As expected, the first *addG* adds an element to the outer (global) queue, and the second *addG* adds to the inner (local) one.

Now suppose we want to add an element to the *outer* queue, from *within* the computation using the inner one. We can do so by binding a name to the outer *addG* operation, as follows:

```
do ...
   addG 1
   ...
   let addOuter = addG in
     withQueue $ do ...
                    addG 2
                    addOuter 3
                    ...
```

As we would expect, the third addition (using *addOuter*) adds an element to the outer queue, and all seems well.

Now suppose we change the definition of *addOuter* to

$$\textbf{let}\ \mathit{addOuter}\ n = \mathit{addG}\ n$$

instead, a subtle change, involving η-converting the definition. Now the call of *addOuter* adds an element to the *inner* queue instead! Likewise if we add a type signature to the binding,

$$\begin{array}{lll} \textbf{let}\ \mathit{addOuter} & :: & (\mathsf{RefMonad}\ m\ r, ?\mathit{queue} :: \mathsf{Queue}\ r\ \mathit{Int}) \Rightarrow \mathsf{Int} \to m\ () \\ \phantom{\textbf{let}}\ \mathit{addOuter} & = & \mathit{addG} \end{array}$$

then the call also adds to the *inner* queue!

The reason is that Haskell's monomorphism restriction applies to the first form of the binding, but to neither of the others. Both the second and third definitions define *addOuter* with the type given in the signature: it is *polymorphic* in *m*, *r*, and the *?queue*, and thus takes the binding for *?queue* from the point where it is called. With the first form of definition, the type of *addOuter* is monomorphic: it is

$$\mathit{addOuter} :: \mathit{Int} \to m\ ()$$

where *m* is *not* quantified here, but is a type in scope at the point of definition, and the context *of the definition* must guarantee

$$(\mathsf{RefMonad}\ m\ r, ?\mathit{queue} :: \mathsf{Queue}\ r\ \mathit{Int})$$

Thus, the binding for *?queue* is taken from the point where the definition appears – and so refers to the *outer* queue in this case, as we hoped. The problem is that we have no way to write down this type for *addOuter* in Haskell, and so the *only* way we can express a monomorphic binding is by writing a variable definition with no type signature – one that the monomorphism restriction applies to.

Clearly, such a subtle semantics may lead to errors. I believe Haskell should syntactically distinguish two forms of binding: polymorphic/overloaded binding with a call-by-name semantics, and monomorphic binding with call-by-need semantics. If = is used for the former, then := might be used for the latter. Then in this example, it would make no difference whether we wrote

$$\textbf{let}\ \mathit{addOuter} := \mathit{addG}$$

or

$$\textbf{let}\ \mathit{addOuter}\ n := \mathit{addG}\ n$$

It is clear in both cases that the binding is monomorphic, and references to implicit parameters are resolved at the definition. Likewise, if the definition used =, then it would clearly be overloaded in both cases. Another nice consequence is that the equality

$$(\lambda x \to e)\ e' = \textbf{let}\ x = e'\ \textbf{in}\ e$$

which holds in Haskell 'except for typing', is replaced by

$$(\lambda x \to e)\ e' = \textbf{let}\ x := e'\ \textbf{in}\ e$$

which holds, full stop. Sadly, though, this would be a major and incompatible change to the language[4]. In the meantime, programmers must just be careful!

8 Conclusions

Many imperative algorithms require global variables. The "world's finest imperative programming language" should provide a safe and easy way to use them. *unsafePerformIO* is not safe, and the monadic approaches are not easy. However, implicit parameters are both safe and easy: they provide exactly the functionality required. Moreover, when we bind an implicit parameter (as in *withQueue*), we delimit the scope of the "global" variable, and we can thus – in contrast to conventional imperative languages – use several instances of an algorithm with global variables without the instances interfering with each other.

This paper underlines the importance of implicit parameters: every Haskell implementation ought to provide them. It also raises questions about aspects of Haskell's type system. The monomorphism restriction, already infamous, forces a somewhat artificial style in this setting. Replacing it with a (safe) alternative should be a high priority. It is even more disturbing that, to write maintainable code, a programmer must *omit* type signatures! A mechanism for abbreviating contexts might make type signatures readable and maintainable, even in this kind of program.

Finally, the reader may wonder whether our conclusion isn't so obvious as to be common knowledge – surely once implicit parameters were introduced it was obvious they should be used for this purpose? We note that

- the original paper on implicit parameters (Lewis *et al.*, 2000) does not explicitly mention global variables as an application;
- the *unsafePerformIO* solution is in common use (Peyton-Jones, 2001), despite its dangers;
- functions which bind global variables, such as *withQueue*, cannot have been used previously, since all type-checkers supporting implicit parameters had bugs preventing these functions from being compiled.

We conclude that the idea, although simple, is not obvious, and deserves wider exposure.

References

Hugs online. www.haskell.org/hugs/.

The Glasgow Haskell Compiler. www.haskell.org/ghc/.

Jones, M. P. (2000) Type classes with functional dependencies. *Proceedings 9th European Symposium on Programming, ESOP 2000: Lecture Notes in Computer Science 1782*. Springer-Verlag.

[4] Adding := as a monomorphic binding operator would be a simple and relatively harmless extension; the dangerous part would be to change the semantics of = by removing the monomorphism restriction. Interestingly, `nhc` does not implement the restriction anyway, and GHC provides a flag to disable it – which suggests that this incompatible change is already accepted by a part of the community, at least.

Launchbury, J. and Peyton-Jones, S. (1995) State in Haskell. *Lisp & Symbolic Computation*, **8**(4), 293–341.

Lewis, J., Shields, M., Meijer, E. and Launchbury, J. (2000) Implicit parameters: Dynamic scoping with static types. *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 108–118.

Liang, S., Hudak, P. and Jones, M. (1995) Monad transformers and modular interpreters. *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 333–343.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML* (*Revised*). MIT Press.

Peyton-Jones, S. *Explicit quantification in Haskell.* http://research.microsoft.com/users/simonpj/Haskell/quantification.html.

Peyton-Jones, S. (2001) Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Hoare, T., Broy, M. and Steinbruggen, R. (eds.), *Engineering Theories of Software Construction*, pp. 47–96. IOS Press.

Peyton-Jones, S., Jones, M. and Meijer, E. (1997) Type classes: exploring the design space. *Haskell workshop.* ACM SIGPLAN.

Peyton-Jones, S., Hughes, J. (eds.), Augustsson, L., Barton, D., Boutel, B., Burton, W., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C. and Wadler, P. (1999) *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language.* Available from `http://haskell.org`

Tofte, M. (1990) Type inference for polymorphic references. *Infor. & Computation*, **89**(1).

Wadler, P. (1995) Monads for functional programming. In: Jeuring, J. and Meijer, E. (eds.), *Advanced Functional Programming: Lecture Notes in Computer Science 925*, pp. 24–52. Springer-Verlag.