Beauty in the Beast

A Functional Semantics for the Awkward Squad

Wouter Swierstra Thorsten Altenkirch

University of Nottingham

{wss, txa}@cs.nott.ac.uk

Abstract

It can be very difficult to debug impure code, let alone prove its correctness. To address these problems, we provide a functional specification of three central components of Peyton Jones's awkward squad: teletype IO, mutable state, and concurrency. By constructing an internal model of such concepts within our programming language, we can test, debug, and reason about programs that perform IO as if they were pure. In particular, we demonstrate how our specifications may be used in tandem with QuickCheck to automatically test complex pointer algorithms and concurrent programs.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Theory of Computation*]: Logics and the Meanings of Programs

General Terms Experimentation, Languages, Reliability, Theory, Verification.

1. Introduction

While we have a solid understanding of pure and total functions, programming with and reasoning about effects is much more difficult. Every functional programmer worth his salt knows how to reverse a list, debug the code, and prove that list reversal is its own inverse. How many could do the same when asked to implement queues using mutable variables?

We address this imbalance by providing a lightweight semantics for side-effecting functions. We demonstrate how to construct pure functional programs that precisely specify the behaviour of effects. Our functional specifications are both pure and executable; as a result we can freely test and debug effectful code in pure functional languages such as Haskell [29]. Reasoning about impure code is reduced to reasoning about the pure functional programs we describe. As we can utilise Haskell's expressivity when specifying our semantics, we can capture a wide range of side-effecting functions:

- We begin by describing how to conduct teletype IO (Section 3). Although the programs we end up with are classic examples of interactive structures in functional programming, the remainder of our paper deals with more complex problems in a similar vein. To illustrate how to reason with our semantics, we prove that the *echo* function does indeed echo any character entered by the user.
- We continue by describing a pure model of mutable state (Section 4). We demonstrate how our semantics may be used in tandem with QuickCheck [7] to test a reverse operator on queues implemented using mutable variables and verify that it runs in constant space.
- We show how concurrent processes can be modeled as functions parametrised by a scheduler (Section 5). Using this intuition, we provide a novel semantics for Concurrent Haskell. We implement an example scheduler and use QuickCheck to verify that a Haskell implementation of channels never duplicates or loses data.
- Finally, we discuss how our functions can be made *total* (Section 6). By restricting ourselves to a total framework, we can avoid some of the hairier corners of Haskell's semantics – such as reasoning in the presence of bottoms.

The pure specifications we present are closely related to the denotational semantics of effects. Implementing them in a functional language, however, is a valuable and novel contribution to the functional programmers' repertoire. It is no longer necessary to treat all side-effecting functions as opaque black boxes: it is finally possible to assign some kind of meaning to programs in the IO monad without leaving the realm of functional programming.

Having such meaning is not only of theoretical interest. Programmers can immediately profit from our specifications. They can test code in the IO monad using QuickCheck without resorting to unpredictable hacks such as *unsafePerformIO*. As our specifications consist entirely of pure values, they can examine the contents of the heap when debugging or experiment with different scheduling algorithms when executing concurrent code. Programmers can study our specifications, without having to learn formal semantics. While the semantics themselves may not be new, it is only by taking it off the blackboard and into the hands of working programmers that theory pays off in practice.

Naturally, we are still left with one central obligation: to show that the actual code the compiler produces matches our functional specification. Although we will discuss different approaches to this problem (Section 8), this issue is beyond the scope of the current paper.

2. Monadic Input/Output

Haskell encapsulates any potentially harmful side-effects in the *IO monad* [31]. Any side-effecting function has a type that marks it as potentially dangerous. For instance, the function *getChar* reads a character from the terminal: it may not format your hard disk, but carelessly executing it might break referential transparency. Its type tells us that it is a side-effecting function that will return a value of type *Char*:

$$\mathit{getChar} :: \mathit{IO}\ \mathit{Char}$$

Similarly, *putChar* prints a given character to the teletype. As we are not interested in the value *putChar* returns, but rather in the effect its execution entails, its result type is *IO* ().

$$\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()$$

There is no safe way to extract the actual character read using *getChar* – doing so would allow seemingly innocent functions to have side-effects: the exact problem we were trying to avoid in the first place. Instead, values in the IO monad are combined using the following two operations:

$$\begin{aligned}
&\mathit{return} :: a \to \mathit{IO}\ a\\
&(\gg\!=) :: \mathit{IO}\ a \to (a \to \mathit{IO}\ b) \to \mathit{IO}\ b
\end{aligned}$$

The *return* function lifts a pure value into the IO monad. The operator $\gg\!=$, usually pronounced 'bind', performs the computation associated with its first argument and passes the result to its second argument. As these are the only primitive operations, programmers must sequence individual computations explicitly using the $\gg\!=$ operator.

As computations are first class values, we can define new control structures. The $\gg$ operator sequences two computations, but discards the result of the first. We can construct a list of computations, and perform them one by one using the *sequence_* combinator:

$$\begin{aligned}
&(\gg) :: \mathit{IO}\ a \to \mathit{IO}\ b \to \mathit{IO}\ b\\
&p \gg q = p \gg\!= \lambda x \to q\\[4pt]
&\mathit{sequence\_} :: [\mathit{IO}\ ()] \to \mathit{IO}\ ()\\
&\mathit{sequence\_}\ [\,] \qquad\;\; = \mathit{return}\ ()\\
&\mathit{sequence\_}\ (x : \mathit{xs}) = x \gg \mathit{sequence\_}\ \mathit{xs}
\end{aligned}$$

Using these building blocks it becomes straightforward to write simple interactive programs:

$$\begin{aligned}
&\mathit{echo} :: \mathit{IO}\ ()\\
&\mathit{echo} = \mathit{getChar} \gg\!= (\lambda c \to \mathit{putChar}\ c) \gg \mathit{echo}\\[4pt]
&\mathit{putString} :: \mathit{String} \to \mathit{IO}\ ()\\
&\mathit{putString} = \mathit{sequence\_} \circ \mathit{map}\ \mathit{putChar}
\end{aligned}$$

Haskell offers syntactic sugar to make large monadic computations a great deal more palatable. A series of monadic expressions can be sequenced using the **do** notation. We can also write the above *echo* as:

$$\begin{aligned}
\mathit{echo} = \mathbf{do}\ &c \leftarrow \mathit{getChar}\\
&\mathit{putChar}\ c\\
&\mathit{echo}
\end{aligned}$$

Haskell provides a large number of built-in functions that can perform all the real world side-effects that every serious programming language should accommodate. The IO monad makes sure that any such side-effecting functions are branded as hazardous.

Unfortunately, but unavoidably, side-effecting functions such as *putChar* are primitive and are not implemented by a pure Haskell expression. This makes debugging or reasoning about such code inherently difficult. The IO monad may prevent unexpected side-effects, but we need a system of formal semantics outside our programming language to prove properties of *putChar*. Throughout this paper, we explore a different avenue of research: we will model *getChar* and *putChar* by pure functions in Haskell.

3. Teletype IO

We begin by defining a data type IO_{tt} that specifies the primitive interactions that can take place with the teletype in Listing 1. Besides getting and putting a single character, we can end the interaction by returning a value.

Listing 1 Teletype IO

$$\begin{aligned}
&\mathbf{data}\ \mathit{IO}_{tt}\ a =\\
&\qquad \mathit{GetChar}\ (\mathit{Char} \to \mathit{IO}_{tt}\ a)\\
&\quad\;\, \mid \mathit{PutChar}\ \mathit{Char}\ (\mathit{IO}_{tt}\ a)\\
&\quad\;\, \mid \mathit{Return}\ a\\[4pt]
&\mathbf{instance}\ \mathit{Monad}\ \mathit{IO}_{tt}\ \mathbf{where}\\
&\quad \mathit{return} = \mathit{Return}\\
&\quad (\mathit{Return}\ a) \qquad\ \gg\!= g = g\ a\\
&\quad (\mathit{GetChar}\ f) \quad\ \gg\!= g = \mathit{GetChar}\ (\lambda c \to f\ c \gg\!= g)\\
&\quad (\mathit{PutChar}\ c\ a) \gg\!= g = \mathit{PutChar}\ c\ (a \gg\!= g)\\[4pt]
&\mathit{getChar} \quad :: \mathit{IO}_{tt}\ \mathit{Char}\\
&\mathit{getChar} \quad = \mathit{GetChar}\ \mathit{Return}\\[4pt]
&\mathit{putChar} \quad :: \mathit{Char} \to \mathit{IO}_{tt}\ ()\\
&\mathit{putChar}\ c = \mathit{PutChar}\ c\ (\mathit{Return}\ ())
\end{aligned}$$

This specification is far from original. Gordon describes a similar approach to doing teletype IO in his thesis [13], and cites related work dating back more than twenty years [16, 19]. Rather than use such structures to perform IO, however, we use them to construct a pure model of functions in the IO monad.

Quite conveniently, the IO_{tt} data type also forms a monad. The *return* function corresponds to the *Return* constructor. The bind operator recurses through the interaction specified by its first argument and feeds the computed value to its second argument.

Using this data type we can *define* the *getChar* and *putChar* functions as if they were any other functions in our language. Although they will not actually print characters to the teletype, we can use them to specify any interaction.

Given a value of type $IO_{tt}\ a$, we can calculate its behaviour. What should the result of an interaction be? From a user's point of view one of three things happen: either a value of type *a* is returned, ending the interaction; or the interaction continues after a character is read from the teletype or printed to the screen. Our *Output* data type in Listing 2 captures exactly these three cases.

Listing 2 Teletype IO – semantics

$$\begin{aligned}
&\mathbf{data}\ \mathit{Output}\ a =\\
&\qquad \mathit{Read}\ (\mathit{Output}\ a)\\
&\quad\;\, \mid \mathit{Print}\ \mathit{Char}\ (\mathit{Output}\ a)\\
&\quad\;\, \mid \mathit{Finish}\ a\\[4pt]
&\mathbf{data}\ \mathit{Stream}\ a = \mathit{Cons}\{\mathit{hd} :: a, \mathit{tl} :: \mathit{Stream}\ a\}\\[4pt]
&\mathit{run}_{tt} :: \mathit{IO}_{tt}\ a \to (\mathit{Stream}\ \mathit{Char} \to \mathit{Output}\ a)\\
&\mathit{run}_{tt}\ (\mathit{Return}\ a)\ \mathit{cs} \qquad = \mathit{Finish}\ a\\
&\mathit{run}_{tt}\ (\mathit{GetChar}\ f)\ \mathit{cs} \quad\ = \mathit{Read}\ (\mathit{run}_{tt}\ (f\ (\mathit{hd}\ \mathit{cs}))\ (\mathit{tl}\ \mathit{cs}))\\
&\mathit{run}_{tt}\ (\mathit{PutChar}\ c\ p)\ \mathit{cs} = \mathit{Print}\ c\ (\mathit{run}_{tt}\ p\ \mathit{cs})
\end{aligned}$$

Once we have fixed the type of *Output*, writing the run_{tt} function that models the behaviour of a given interaction is straightforward. We assume that we have a stream of characters that have been entered by the user. Whenever our interaction gets a character, we read the head of the stream and continue the interaction with the tail.

Using the *putChar* and *getChar* functions that we have defined ourselves, we can write the same code for teletype interactions as before, but we now have a good understanding of how they behave. When such code is compiled, we can replace our *putChar* and *getChar* with calls to the primitive version defined in the Haskell Prelude. Before moving on to more complex semantics, we illustrate how to prove properties of teletype interactions.

Example: echo

Using our semantics, we can prove once and for all that *echo* prints out any character entered at the teletype. In particular, we can define the following function that exhibits the behaviour we expect *echo* to have:

$$
\begin{aligned}
&\mathit{copy} :: \mathit{Stream\ Char} \to \mathit{Output}\ ()\\
&\mathit{copy}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{Read}\ (\mathit{Print}\ x\ (\mathit{copy}\ \mathit{xs}))
\end{aligned}
$$

The *copy* function simply copies the stream of characters entered at the teletype to the stream of characters printed to the teletype one at a time. The *Read* constructor is important here: a variation of the echo function that required two characters to be typed before producing any output would not satisfy this specification. We can now prove that running *echo* will behave exactly like the *copy* function.

Using a variation of the take lemma [4], we show that *copy cs* and the result of running *echo* on *cs* are identical, for every input stream *cs*. The proof requires us to define an extra *take* function, analogous to the one for lists:

$$
\begin{aligned}
&\mathit{take} :: \mathit{Int} \to \mathit{Output}\ () \to \mathit{Output}\ ()\\
&\mathit{take}\ (n+1)\ (\mathit{Print}\ x\ \mathit{xs}) = \mathit{Print}\ x\ (\mathit{take}\ n\ \mathit{xs})\\
&\mathit{take}\ (n+1)\ (\mathit{Read}\ \mathit{xs}) \quad\ = \mathit{Read}\ (\mathit{take}\ (n+1)\ \mathit{xs})\\
&\mathit{take}\ 0\ \quad\quad \_ \quad\quad\quad\quad = \mathit{Finish}\ ()
\end{aligned}
$$

We can now prove that:

$$\mathit{take}\ n\ (\mathit{run}_{tt}\ \mathit{echo}\ \mathit{xs}) = \mathit{take}\ n\ (\mathit{copy}\ \mathit{xs})$$

The proof proceeds by induction on n. The base case is trivial; the induction step is in Listing 3.

Proving such an equation is still quite some work. However, most Haskell programmers are already familiar with such equational proofs. There is no external system of semantics needed to prove such a property, but programmers can reason about their code as if it were pure.

Listing 3 The behaviour of *echo*

$$
\begin{aligned}
&\quad \mathit{take}\ (n+1)\ (\mathit{run}_{tt}\ \mathit{echo}\ (\mathit{Cons}\ x\ \mathit{xs}))\\
=&\quad \{\text{by definition of } \mathit{echo}, \mathit{putChar} \text{ and } \mathit{getChar}\}\\
&\quad \mathit{take}\ (n+1)\ (\mathit{run}_{tt}\ (\mathit{GetChar\ Return}\\
&\qquad\qquad\qquad\qquad \gg\!= \lambda c \to \mathit{PutChar}\ c\ (\mathit{Return}\ ())\\
&\qquad\qquad\qquad\qquad \gg \mathit{echo})\\
&\qquad\qquad\qquad (\mathit{Cons}\ x\ \mathit{xs}))\\
=&\quad \{\text{by definition of } \mathit{run}_{tt} \text{ and } (\gg\!=)\}\\
&\quad \mathit{take}\ (n+1)\\
&\qquad (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{Return}\ x\\
&\qquad\qquad\qquad\qquad \gg\!= \lambda c \to \mathit{PutChar}\ c\ (\mathit{Return}\ ())\\
&\qquad\qquad\qquad\qquad \gg \mathit{echo})\\
&\qquad\qquad\qquad \mathit{xs}))\\
=&\quad \{\text{by definition of } (\gg\!=)\}\\
&\quad \mathit{take}\ (n+1)\\
&\qquad (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{PutChar}\ x\ (\mathit{Return}\ () \gg \mathit{echo}))\ \mathit{xs}))\\
=&\quad \{\text{by definition of } (\gg)\}\\
&\quad \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{run}_{tt}\ (\mathit{PutChar}\ x\ \mathit{echo})\ \mathit{xs}))\\
=&\quad \{\text{by definition of } \mathit{run}_{tt}\}\\
&\quad \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{Print}\ x\ (\mathit{run}_{tt}\ \mathit{echo}\ \mathit{xs})))\\
=&\quad \{\text{by definition of } \mathit{take}\}\\
&\quad \mathit{Read}\ (\mathit{Print}\ x\ (\mathit{take}\ n\ (\mathit{run}_{tt}\ \mathit{echo}\ \mathit{xs})))\\
=&\quad \{\text{induction}\}\\
&\quad \mathit{Read}\ (\mathit{Print}\ x\ (\mathit{take}\ n\ (\mathit{copy}\ \mathit{xs})))\\
=&\quad \{\text{by definition of } \mathit{take}\}\\
&\quad \mathit{take}\ (n+1)\ (\mathit{Read}\ (\mathit{Print}\ x\ (\mathit{copy}\ \mathit{xs})))\\
=&\quad \{\text{by definition of } \mathit{copy}\}\\
&\quad \mathit{take}\ (n+1)\ (\mathit{copy}\ (\mathit{Cons}\ x\ \mathit{xs}))
\end{aligned}
$$

4. Mutable State

While teletype IO makes an interesting example, an obvious question is whether or not this approach can deal with anything more complicated. Interestingly, we can handle mutable state in a very similar fashion.

Mutable state in Haskell revolves around mutable variables known as *IORefs*. There are three functions that respectively create, write to and read from an *IORef*:

$$
\begin{aligned}
&\mathit{newIORef} \quad :: a \to \mathit{IO}\ (\mathit{IORef}\ a)\\
&\mathit{writeIORef} :: \mathit{IORef}\ a \to a \to \mathit{IO}\ ()\\
&\mathit{readIORef} \quad :: \mathit{IORef}\ a \to \mathit{IO}\ a
\end{aligned}
$$

We begin by defining a data type representing the possible changes to the state in Listing 4. We follow Haskell's lead and introduce separate constructors for each operation on *IORefs*. As with the teletype, we have an additional constructor *Return* that lifts pure values to stateful ones. It is worth pointing out that the signatures of the functions we wish to implement determine the constructors of our data type; the only freedom we have is in the representation of memory locations and data.

We model memory locations using integers. This is rather limited. By using integers to model memory locations, programmers could 'invent' their own locations, perform pointer arithmetic, or access unallocated memory. To address this problem, we propose to use Haskell's module system to hide the constructor of the *IORef* type. As a result, the only operations a programmer can perform with an *IORef* are those supported by our IO_s data type.

We also restrict ourself to mutable variables storing integers. A more flexible approach would be to use Haskell's support for dynamic types [5, 1] to allow references to different types. This does make reasoning about our programs much, much more difficult [10], as the implementation of dynamic types relies on *unsafeCoerce*, for instance. For the sake of presentation, we therefore choose to limit ourself to references storing a fixed *Data* type. The price we pay is, of course, having to update this type every time we wish to change the types stored in mutable references. We will discuss how to tackle both this and the previous problem using a more expressive type system in Section 6.2.

Now that we have all relevant definitions, we construct an interpretation of these operations in Listing 5. Haskell already has a very convenient library for writing stateful computations that pivots around the state monad:

$$\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\{\mathit{runState} :: (s \to (a, s))\}$$

Listing 4 Mutable state – data type

```
type Data = Int
type Loc  = Int

data IO_s a =
     NewIORef Data (Loc → IO_s a)
   | ReadIORef Loc (Data → IO_s a)
   | WriteIORef Loc Data (IO_s a)
   | Return a

instance Monad IO_s where
  return              = Return
  (Return a) ≫= g = g a
  (NewIORef d f) ≫= g
     = NewIORef d (λl → f l ≫= g)
  (ReadIORef l f) ≫= g
     = ReadIORef l (λd → f d ≫= g)
  (WriteIORef l d s) ≫= g
     = WriteIORef l d (s ≫= g)

newtype IORef = IORef Loc

newIORef :: Data → IO_s IORef
newIORef d = NewIORef d (Return ∘ IORef)

readIORef :: IORef → IO_s Data
readIORef (IORef l) = ReadIORef l Return

writeIORef :: IORef → Data → IO_s ()
writeIORef (IORef l) d = WriteIORef l d (Return ())
```

Listing 5 Mutable state – semantics

```
data Store = Store{fresh :: Loc, heap :: Heap}
type Heap = Loc → Data

emptyStore :: Store
emptyStore = Store{fresh = 0}

run_s :: IO_s a → a
run_s io = evalState (runIOState io) emptyStore

runIOState :: IO_s a → State Store a
runIOState (Return a) = return a
runIOState (NewIORef d g)
   = do loc ← alloc
        extendHeap loc d
        runIOState (g loc)
runIOState (ReadIORef l g)
   = do d ← lookupHeap l
        runIOState (g d)
runIOState (WriteIORef l d p)
   = do extendHeap l d
        runIOState p

alloc :: State Store Loc
alloc = do loc ← gets fresh
           modifyFresh ((+) 1)
           return loc

lookupHeap :: Loc → State Store Data
lookupHeap l = do h ← gets heap
                  return (h l)

extendHeap :: Loc → Data → State Store ()
extendHeap l d = modifyHeap (update l d)

modifyHeap :: (Heap → Heap) → State Store ()
modifyHeap f = do s ← get
                  put (s{heap = f (heap s)})

modifyFresh :: (Loc → Loc) → State Store ()
modifyFresh f = do s ← get
                   put (s{fresh = f (fresh s)})

update :: Loc → Data → Heap → Heap
update l d h k
   | l ≡ k      = d
   | otherwise = h k
```

The state monad has several functions to manipulate the otherwise implicit state. In particular, we will make use the following functions:

```
get       :: State s s
gets      :: (s → a) → State s a
put       :: s → State s ()
evalState :: State s a → s → a
execState :: State s a → s → s
```

To access the hidden state, we use the *get* and *gets* functions that respectively return the hidden state and project value from it. The *put* function updates the state. Finally, the functions *evalState* and *execState* run a stateful computation, and project out the final result and the final state respectively.

Before we can use the state monad, we must decide on the type of the state *s* that we wish to use. In our case, there are two important pieces of information the state should record: the next free memory location and the heap that maps memory locations to data. Both are captured by our *Store* data type.

Now we can begin defining the function run_s that evaluates the stateful computation described by a value of type IO_s. We begin by constructing a value of type *State Store a*, and subsequently evaluate this computation, starting with an empty store. Note that we leave the *heap* of the initial state undefined.

Once again, the *Return* case ends the stateful computation. Creating a new *IORef* involves allocating memory and extending the heap with the new data. A *ReadIORef* operation looks up the data stored at the relevant location. Writing to an *IORef* updates the heap with the new data. Although we require a few auxiliary functions to manipulate the state and the heap, the code in Listing 5 should contain very few surprises.

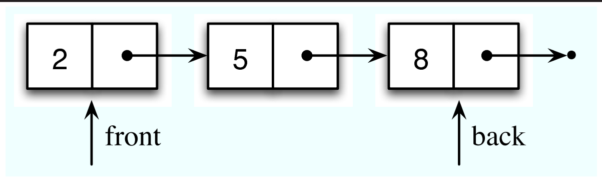
All in all, the definition and semantics of an *IORef* fits on a single page and is remarkably simple. Some might even argue that the semantics are trite and trivial – but this is a good thing! We want our semantics to be easy to understand. There is really no need to make things any more complicated.

Example: queues

To demonstrate how our semantics can be used in practice, we implement queues using mutable references. Such queues consist of two references to the first and last cell of the queue. Every cell stores an integer value together with a pointer to the next cell. The last cell's reference points to a null value. Figure 1 illustrates what an example queue might look like.

Although the implementation is standard, it is all too easy to make a mistake. Listing 6 gives the type signatures of the operations involved. To begin with, we must fix the type of our references. From Figure 1 we can see that every pointer is either null, or points to a cell storing an integer and another pointer. We need to

Figure 1 An example queue implemented using mutable references



change the type of data stored by a pointer, *Data*, in our semantics accordingly.

A *Queue* consists of a pair of pointers to the front and back of the queue. In an empty queue, both these pointers are null. A complete implementation of the type signatures in Listing 6 is provided in an appendix for the sake of completeness.

Listing 6 Implementing queues using mutable variables

```
data Data   = Cell Int IORef | NULL
type Queue = (IORef, IORef)
emptyQueue :: IO_s Queue
enqueue    :: Queue → Int → IO_s ()
dequeue    :: Queue → IO_s (Maybe Int)
```

Claessen and Hughes have shown how to use QuickCheck to test properties of a similar implementation of queues in the *ST* monad [8]. Rather than follow suit, we implement queue reversal.

Listing 7 shows how to reverse a queue. If the queue is empty, we are done. Otherwise, we traverse the linked list, reversing every pointer as we go and finish off by swapping the *front* and *back* pointers.

Listing 7 Reversing queues

```
reverseQueue :: Queue → IO_s ()
reverseQueue (front, back)
  = do f ← readIORef front
       case f of
         NULL → return ()
         Cell x nextRef → do
           flipPointers NULL (Cell x nextRef)
           b ← readIORef back
           writeIORef front b
           writeIORef back f
flipPointers :: Data → Data → IO_s ()
flipPointers prev NULL = return ()
flipPointers prev (Cell x next)
  = do nextCell ← readIORef next
       writeIORef next prev
       flipPointers (Cell x next) nextCell
revRevProp :: [Int] → Bool
revRevProp xs = xs ≡ run_s (revRev xs)
  where revRev xs = do q ← listToQueue xs
                       reverseQueue q
                       reverseQueue q
                       queueToList q
```

Operations that rely heavily on pointer manipulations are very easy to get wrong. How can we be sure that *reverseQueue* does what we expect? We should not just use equality to compare queues – this will just compare the addresses of the head and tail, but not the content of the queue.

One solution is to define a pair of functions *listToQueue* and *queueToList*. The former enqueues all the elements of a list to the empty queue; the latter dequeues elements from a queue until it is empty. Using these functions, we can define the property *revRevProp* in Listing 7 that compares a list of integers to the result of enqueueing the integers, reversing the resulting queue twice, and finally dequeuing all the elements of the queue.

When we run QuickCheck on the resulting property, we can claim with some degree of certainty that our operation is its own inverse:

```
*Main> quickCheck revRevProp
OK, passed 100 tests.
```

This is, of course, a useless property to check—the identity function satisfies the same specification. It does illustrate, however, that proofs and properties of pure functions, such as the famous $reverse \circ reverse = id$ on lists, do not need to be treated differently from those of impure functions.

In contrast to the work by Claessen and Hughes, we can also verify that queue reversal does not allocate any new memory. We accomplish this by inspecting the state after running a computation. If new memory has been allocated, the *fresh* counter that points to the next free memory cell will have been incremented.

The *memoryUsage* function in Listing 8 returns the number of memory cells needed by a computation. Using this function we can compare the amount of memory needed to store a queue, *queueMemSize*, and the amount of memory allocated after reversing a queue, *revQueueMemSize*. The *revMemProp* property then formulates the desired property: reversing a queue should not allocate new memory.

Listing 8 Memory usage of queue reversal

```
memoryUsage :: IO_s a → Int
memoryUsage io
  = fresh (execState (runIOState io) emptyStore)
queueMemSize :: [Int] → Int
queueMemSize xs = memoryUsage (listToQueue xs)
revQueueMemSize :: [Int] → Int
revQueueMemSize xs
  = memoryUsage (listToQueue xs >>= reverseQueue)
revMemProp :: [Int] → Bool
revMemProp xs = queueMemSize xs ≡ revQueueMemSize xs
```

This example shows how we can use the tools most functional programmers are comfortable with to reason about effectful programs. As the store is modeled by a pure value, we can check properties of our programs that we could not even express if we wrote them using the *ST* monad.

The model for mutable state is more complicated than our previous model for teletype IO. Proofs using this model can easily become quite complex. Even formulating properties involving the heap layout, for instance, can become rather onerous. Fortunately, as illustrated by Bird [3], we can introduce high-level combinators to facilitate reasoning about and formulating properties of pointer algorithms. Just writing down the low-level semantics of mutable state is by no means the whole story, but rather forms a starting point from which to embark on more serious analyses.

5. Concurrency

Although the models for teletype IO and mutable state were relatively straightforward, concurrency poses a more serious problem. Concurrent Haskell enables programmers to fork off a new thread with the *forkIO* function:

```
forkIO :: IO a → IO ThreadId
```

The new thread that is forked off will evaluate the argument of the *forkIO* call. The programmer can subsequently use a *ThreadId* to kill a thread or throw an exception to a specific thread.

Threads can communicate with one another using a synchronised version of an *IORef* called an *MVar*. As with an *IORef* there are three functions to create, write to and read from an *MVar*:

```
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a → a → IO ()
takeMVar     :: MVar a → IO a
```

Unlike an *IORef*, an *MVar* can be empty. Initially, there is no value stored in an *MVar*. An empty *MVar* can be filled using the function *putMVar*. A filled *MVar* can be emptied using the function *takeMVar*. If a thread tries to fill a non-empty *MVar*, the thread is blocked until another thread empties the *MVar* using *takeMVar*. Dually, when a thread tries to take a value from an empty *MVar*, the thread is blocked until another thread puts a value into the *MVar*.

Although there are several other functions in Haskell's concurrency library, we choose to restrict ourselves to the four functions described above for the moment.

In what should now be a familiar pattern, we begin by defining the data type IO_c for concurrent input/output in Listing 9. Once again, we add a constructor for every primitive function together with an additional *Return* constructor. As is the case in our *IORef* implementation, we model memory addresses and the data stored there as integers. Forked off threads have a unique identifier, or *ThreadId*, which we also model as an integer. The type of *Fork* is interesting as it will take an IO_c *b* as its first argument, regardless of what *b* is. This corresponds to the parametric polymorphism that the *forkIO* function exhibits – it will fork off a new thread, regardless of the value that the new thread returns.

Once we have defined the data type IO_c, we can show it is a monad just in the same fashion that IO_s and IO_{tt} are monads. We continue by defining the basic functions, corresponding to the constructors.

Running the computation described by a value of type IO_c *a* is not as straightforward as the other models we have seen so far. Our model of concurrency revolves around an explicit scheduler that determines which thread is entitled to run. The *Scheduler* is a function that, given an integer *n*, returns a number between 0 and $n-1$, together with a new scheduler. Intuitively, we inform the scheduler how many threads are active and it returns the scheduled thread and a new scheduler. Listing 10 describes how initially to set up the semantics of our concurrency operations.

Besides the scheduler, we also need to keep track of the threads that could potentially be running. The thread *soup* is a finite map taking a *ThreadId* to a *ThreadStatus*. Typically, such a *ThreadStatus* consists of the process associated with a given *ThreadId*. Note, however, that once a thread is finished, there is no value of IO_c that we could associate with its *ThreadId* so we have an additional *Finished* constructor to deal with this situation. Besides the thread soup we also store an integer, *nextTid*, that represents the next unassigned *ThreadId*.

In addition to information required to deal with concurrency, we also need a lot of machinery to cope with mutable state. In particular, we keep track of a *heap* and *fresh* just as we did for our model of mutable state. Unlike an *IORef*, an *MVar* can be empty; hence the *heap* maps locations to *Maybe Data*, using *Nothing* to

Listing 9 Concurrency – data type

```
type ThreadId = Int
type Data     = Int
type Loc      = Int

data IO_c a =
    NewEmptyMVar (Loc → IO_c a)
  | TakeMVar Loc (Data → IO_c a)
  | PutMVar Loc Data (IO_c a)
  | ∀ b . Fork          (IO_c b) (ThreadId → IO_c a)
  | Return a

newtype MVar = MVar Loc

instance Monad IO_c where
  return = Return
  Return x ≫= g         = g x
  NewEmptyMVar f ≫= g = NewEmptyMVar (λl → f l ≫= g)
  TakeMVar l f ≫= g     = TakeMVar l (λd → f d ≫= g)
  PutMVar c d f ≫= g    = PutMVar c d (f ≫= g)
  Fork p1 p2 ≫= g       = Fork p1 (λtid → p2 tid ≫= g)

newEmptyMVar        :: IO_c MVar
newEmptyMVar        = NewEmptyMVar (Return ∘ MVar)

takeMVar            :: MVar → IO_c Data
takeMVar (MVar l)   = TakeMVar l Return

putMVar             :: MVar → Data → IO_c ()
putMVar (MVar l) d = PutMVar l d (Return ())

forkIO              :: IO_c a → IO_c ThreadId
forkIO p            = Fork p Return
```

Listing 10 Concurrency – initialisation

```
newtype Scheduler =
  Scheduler (Int → (Int, Scheduler))

data ThreadStatus =
    ∀ b . Running (IO_c b)
  | Finished

data Store = Store{fresh :: Loc
                  , heap :: Loc → Maybe Data
                  , nextTid :: ThreadId
                  , soup :: ThreadId → ThreadStatus
                  , scheduler :: Scheduler
                  }

initStore :: Scheduler → Store
initStore s = Store{fresh     = 0
                  , nextTid = 1
                  , scheduler = s
                  }

runIO_c :: IO_c a → (Scheduler → a)
runIO_c io s = evalState (interleave io) (initStore s)
```

represent an empty *MVar*. All these ingredients together form the *Store*.

To interpret a value of type $IO_c\ a$, we define a function that will run the concurrent process that it represents. Once again, we use Haskell's state monad to encapsulate the implicit plumbing involved with passing around the *Store*. To run a concurrent process we must tackle two more or less separate issues: how to perform a single step of computation and how to interleave these individual steps. We will begin defining the single steps in Listing 11, leaving the *interleave* function undefined for the moment.

The *step* function closely resembles our semantics for mutable variables, with a few minor adjustments. In contrast to the situation for mutable variables, we do not guarantee that we return a value of type *a*, but rather distinguish three different possible results.

First of all, a thread might terminate and produce a result. Secondly, a thread might have a side-effect, such as taking the value stored in an *MVar*, and return a new, shorter process. Finally, a thread might be blocked, for instance when it tries to take a value from an empty *MVar*. These three cases together form the *Status* data type that is returned by the *step* function.

Note that we have omitted a few functions that modify a specific part of the state, analogous to *modifyFresh* and *modifyHeap* in Listing 5.

There are a few differences with the model of mutable state. When we return a value, the thread is finished and we wrap our result in a *Stop* constructor. Creating a new *MVar* is almost identical to creating a new *IORef*. The only difference is that an *MVar* is initially empty, so we extend the heap with *Nothing* at the appropriate location.

The case for *TakeMVar* and *PutMVar* is more interesting. When we read an *MVar* we look up the appropriate information in the heap. If the *MVar* is filled, we empty it and perform a single step. When the *MVar* is empty, the thread is blocked and we cannot make any progress. The situation for writing to an *MVar* is dual.

The final case of the *step* function deals with forking off new threads. We begin by generating a *ThreadId* for the newly created thread. Subsequently, we extend the thread soup with the new thread. Finally, we return the parent thread wrapped in the *Step* constructor as the thread has made progress, but is not yet finished.

Although it was relatively easy to perform a single step, the interleaving of separate threads is more involved. Listing 12 finally defines the *interleave* function.

Different threads may return different types. In particular, the main thread has type $IO_c\ a$, but auxiliary threads have type $IO_c\ b$ for some unknown type *b*. To make this distinction, we introduce the *Process* data type.

Essentially, to interleave a concurrent process we begin by consulting the scheduler to determine the next active thread. Initially, this will always be the main process. Once the main process forks off child threads, however, such threads may be scheduled instead. The result of scheduling is a value of type *Process a* together with the *ThreadId* of the thread that has been scheduled. Although we have omitted the code for the *schedule* function, it is relatively straightforward: given the main process, it consults the scheduler for the next *ThreadId*, and returns that *ThreadId* together with the corresponding process from the thread soup. We need to pass the main process to the scheduler, as it is not in the thread soup, but could still be scheduled.

If we want to use the *Process* returned by the scheduler, we need to be careful. We would like to allow the scheduled process to perform a single step – but what should we do with the result? If the main thread returns a final value, we can wrap things up and return that value. If an auxiliary thread returns a value, we are not particularly interested in its result, but rather want to terminate

Listing 11 Concurrency – performing a single step

```
data Status a = Stop a | Step (IO_c a) | Blocked
step :: IO_c a → State Store (Status a)
step (Return a) = return (Stop a)
step (NewEmptyMVar f)
  = do loc ← alloc
       modifyHeap (update loc Nothing)
       return (Step (f loc))
step (TakeMVar l f)
  = do var ← lookupHeap l
       case var of
         Nothing → return Blocked
         (Just d) → do emptyMVar l
                       return (Step (f d))
step (PutMVar l d p)
  = do var ← lookupHeap l
       case var of
         Nothing → do fillMVar l d
                      return (Step p)
         (Just d) → return Blocked
step (Fork l r)
  = do tid ← freshThreadId
       extendSoup l tid
       return (Step (r tid))
lookupHeap :: Loc → State Store (Maybe Data)
lookupHeap l      = do h ← gets heap
                       return (h l)
freshThreadId :: State Store ThreadId
freshThreadId     = do tid ← gets nextTid
                       modifyTid ((+) 1)
                       return tid
emptyMVar :: Loc → State Store ()
emptyMVar l       = modifyHeap (update l Nothing)
fillMVar :: Loc → Data → State Store ()
fillMVar l d      = modifyHeap (update l (Just d))
extendSoup :: IO_c a → ThreadId → State Store ()
extendSoup p tid
  = modifySoup (update tid (Running p))
```

the thread. As we want to treat the main and auxiliary threads differently, we need to pattern match on the scheduled process.

Regardless of which thread was scheduled, we allow it to perform a single step. There are five possible outcomes of this step, that we cover one by one:

The main thread stops When the main thread terminates, the entire concurrent process is finished. We simply return the value that the step produced. Any auxiliary threads that have unfinished work will never be scheduled.

An auxiliary thread stops If an auxiliary thread finished its computation and returns a value, we discard this value and finish the thread. We update the thread soup to indicate that this thread is finished and continue the interleaving.

The main threads performs a step If the main thread manages to successfully perform a single step, we continue by calling the *interleave* function again. The argument we pass to the

Listing 12 Concurrency – interleaving

```
data Process a =
    Main (IO_c a)
  | ∀b . Aux (IO_c b)
interleave :: IO_c a → State Store a
interleave main
  = do (tid, t) ← schedule main
       case t of
         Main p →
           do x ← step p
              case x of
                Stop r    → return r
                Step p    → interleave p
                Blocked → interleave main
         Aux p →
           do x ← step p
              case x of
                Stop _    → do finishThread tid
                                interleave main
                Step q    → do extendSoup q tid
                                interleave main
                Blocked → interleave main
finishThread tid = modifySoup (update tid Finished)
```

interleave function is the new main process that was wrapped in a *Step* constructor.

An auxiliary thread performs a step When an auxiliary thread makes progress, we proceed much in the same way as we do for the main thread. Instead of passing the new computation to *interleave*, however, we update the thread soup. Once the soup has been updated, we continue by interleaving with the same main thread as we started with.

Blocked If the scheduled thread can make no progress, for instance because it is waiting for an empty *MVar* to be filled, scheduling that thread will return *Blocked*. In that case, we schedule a new thread, until progress is made.

The semantics for concurrency are more complicated than those for teletype IO and mutable state. Actually using them, however, is no more difficult.

Example: channels

When Peyton Jones describes the semantics of concurrency in Haskell [28], he illustrates the expressive power of *MVars* by giving an implementation of channels.

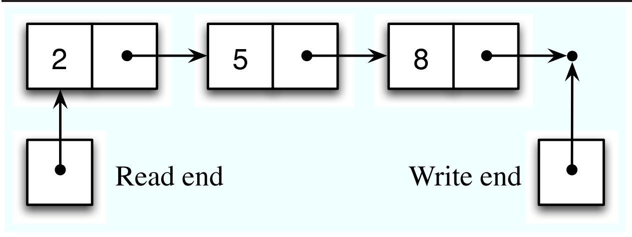
Channels enable separate threads to communicate safely. They generalise the queues we have seen previously, as a channel allows multiple processes to read from and write to it. This is accomplished by having a pair of *MVars* storing pointers to the read end and write end of the channel. Whenever a process wants to read from or write to the channel, it must first acquire access to the appropriate end of the queue. Storing these pointers in *MVars* ensures that separate writes or reads can never interfere with one another. One example of a channel is illustrated in Figure 2.

Peyton Jones claims that:

> . . . each value read will go to exactly one process.

Unfortunately, there is no justification of this claim. Proving such statements can, of course, be really difficult. In fact, it is already

Figure 2 An example channel



hard to formulate precisely what it means for data to be lost or duplicated.

Rather than repeat the implementation of channels, we once again focus on how to use QuickCheck to demonstrate that certain properties are at least plausible. Listing 13 gives the types of channels and the data stored by references, together with the type signatures of the channel operations. We do not discuss how to implement these operations, but refer the implementation discussed in [28]. Our main concern is checking whether or not the above property is plausible.

Listing 13 Channels

```
type Channel = (MVar, MVar)
data Data =
    Cell Int MVar
  | Ref MVar
  | Res [Int]
newChan :: IO_c Channel
putChan  :: Channel → Int → IO_c ()
getChan  :: Channel → IO_c Int
```

Before we can implement the channel operations, we need to fix the data type *Data*, i.e. the type of data stored in an *MVar*. As we can see from Figure 2, the data stored in an *MVar* is not always a cell. In particular, the references to the read end and write end of the channel are also stored in an *MVar*. Therefore, we need to add an extra constructor *Ref* to our *Data* data type. Finally, we will later use an *MVar* to store a list of integers in the test we propose to run; therefore, we add a final constructor *Res*.

Listing 14 shows the test we would like to run. The *chanTest* function takes a list of integers, and forks off a thread for each integer that will write that integer to an initially empty channel. It also forks off a thread for each integer that attempts to read from the channel. Once a thread manages to read from the channel, it records the value read in a shared *MVar* called *result*. The main thread then waits until every thread has successfully read from the channel, and concludes by returning the list of all values that have been read. This final result should, of course, be a permutation of our original list.

The semantics of concurrency we have presented abstracts over the scheduling algorithm. Before we can run the test we have in mind, we must therefore decide what scheduler to use. As we are already using QuickCheck, we implement a random scheduling algorithm in an attempt to maximize the number of interleavings. Listing 15 gives one possible implementation of such a scheduler.

The *streamSch* function defines a scheduler, given a stream of integers. The definition of the *Stream* data type can be found in Listing 2. Whenever it is asked to schedule a thread, it uses the appropriate modulus on the head of the stream and continues scheduling with its tail. As we can use QuickCheck to generate

Listing 14 Testing the implementation of channels

```
chanTest :: [Int] → IO_c [Int]
chanTest ints
 = do ch ← newChan
      result ← newEmptyMVar
      putMVar result (Res [])
      forM ints (λi → forkIO (putChan ch i))
      replicateM (length ints) (forkIO (reader ch result))
      wait result ints

reader :: Channel → MVar → IO_c ()
reader channel var
   = do x ← getChan channel
        (Res xs) ← takeMVar var
        putMVar var (Res (x : xs))

wait :: MVar → [Int] → IO_c [Int]
wait var xs
 = do (Res r) ← takeMVar var
      if length r ≡ length xs
        then return r
        else do putMVar var (Res r)
                wait var xs
```

Listing 15 Random scheduling

```
streamSch :: Stream Int → Scheduler
streamSch xs =
  Scheduler (λk → (hd xs ‘mod‘ k, streamSch (tl xs)))

instance Arbitrary a ⇒ Arbitrary (Stream a) where
  arbitrary = do x ← arbitrary
                 xs ← arbitrary
                 return (Cons x xs)
```

a random stream of integers, we use the *streamSch* to produce a random scheduler.

The following property should hold:

```
chanProp ints stream =
  sort (runIO_c (chanTest ints) (streamSch stream))
     ≡ sort ints
```

Once again, QuickCheck informs us that the above property holds for 100 test runs. When we classify the input lists according to their length, it is reassuring to see that this property even holds for lists of more than 90 elements: that's almost 200 randomly scheduled pseudothreads vying for access to a single channel!

Clearly, this property is insufficient to verify Peyton Jones's claim. We should also check that the resulting channel is empty and all the threads are finished. Even then, we have only checked one kind of scenario, where every thread either writes or reads a single value. Yet our semantics are capable of providing *some* form of sanity check. It is not clear how such a check could be realized using Peyton Jones's semantics.

It may not be a surprise that the implementation of channels using *MVars* is correct. Running this test, however, found a very subtle bug in our scheduling function. Recall that the *schedule* function returns the *ThreadId* and process of the scheduled thread. If we schedule a finished thread, we call the *schedule* function again, in search of a thread that is not yet finished. In a faulty version of our specification, if we encountered a finished thread, we called the schedule function again, but returned the *ThreadId* of the finished thread. This caused quite some chaos in the thread soup, as threads were lost and duplicated.

As the entire state of concurrent computations is a pure value, we can access otherwise inaccessible data, such as the size of the heap or the number of threads that have finished. In particular, abstracting over the scheduler allows us to check certain algorithms with specific schedulers or check a large number of interleavings using a random scheduler as we see fit.

Extensions

Haskell cognoscenti will have spotted that we have not included all the primitive functions provided by Concurrent Haskell. Adding new primitives to our semantics is, however, not difficult to do. In particular, we do not need to extend the code that deals with the interleaving and scheduling, but can restrict ourselves to adapting the IO_c data type and the *step* function. For instance, it is fairly straightforward to extend our semantics with functions such as:

```
killThread :: ThreadId → IO ()
yield :: IO ()
```

The *killThread* function simply removes a certain thread from the thread soup; the *yield* function merely passes control to some other thread, whenever it is scheduled.

These semantics could also be extended to deal with asynchronous exceptions and explicitly delayed threads. Haskell's exception mechanism allows threads to throw exceptions to other threads. In our semantics, throwing an exception to another thread, would involve updating the thread soup, i.e. alerting the thread that receives the exception. Besides asynchronous exceptions, programmers can also delay threads for a number of milliseconds. A delayed thread will never be scheduled until enough time has elapsed. We hope to be able to address this in the future by a more refined functional semantics that takes time into account explicitly, as is already done in functional reactive programming systems such as Yampa [17]. Such semantics require a judicious choice of supported operations – adding explicitly delayed threads may add new functionality, but could drastically complicate the semantics.

6. Totality

The semantics we have provided are very suitable for what has been dubbed 'fast and loose' reasoning [9]. We use QuickCheck and freely perform equational reasoning without worrying about undefined values or non-terminating functions. While this justifies our results to a degree, we may sometimes be interested in a watertight proof of correctness. The semantics we have provided so far, however, are unsuitable for such proofs.

Fortunately, we can make our semantics more precise. If we make sure that all our *run* functions are total, then any equality we prove between programs written in a total calculus will be valid. This is particularly relevant for programming languages with dependent types where all functions are guaranteed to be total by construction, such as Epigram [23] or *Gallina*, the functional core of Coq [2]. In such systems, we can not only write our programs, but also prove that they meet their specification.

Throughout our semantics we have occasionally used general recursion and undefined values, such as the initial heap. By avoiding bottoms and restricting ourselves to primitive recursion, the total *run* functions we describe below will assign sensible semantics to every program.

6.1 Total semantics for teletype IO

The run_{tt} function in Listing 2 is total. When we restrict ourself to a total language, however, all data is finite. In particular, there can be no infinite sequence of *PutChar* constructors that produce an infinite stream of output. This is rather unfortunate: there are situations where we would like to repeatedly print a character to the teletype.

One solution is to distinguish between inductively defined data and coinductively defined codata, as has been proposed by Turner [33]. If we consider IO_{tt} and *Output* to be codata, the *stream* function below is total:

$stream :: Char \to IO_{tt}\ ()$
$stream\ c = PutChar\ c\ (stream\ c)$

Once again, it becomes possible to output an infinite stream of characters to the teletype. Similarly, we could write a *sink* function that consume input from the user, without ever producing any output.

There is slight subtlety here. We have chosen to make both reading and printing visible in our *Output* data type. While this makes sense for teletype interactions, it is questionable whether you should be able to observe how much data a process reads from a handle that is not *stdin*. If we drop the *Read* constructor of our *Output* data type, our semantics become more abstract: we describe a process's behaviour as stream processor. Ghani et al. [11] use a mixed data-codata structure that can be used to specify exactly this behaviour in a total setting.

6.2 Total semantics for mutable state

There are a few problems with the semantics of mutable state in Section 4. Although the *runIOState* function in Listing 5 only uses primitive recursion, the semantics makes use of undefined values, such as the empty heap. As a result, programmers may access unallocated memory, resulting in unspecified behaviour. This is easily fixed, provided our type system is sufficiently expressive.

In a dependently typed setting we can model the heap as an *n*-tuple. We can then model an *IORef* as a pointer into the heap that will never go out of bounds. Finally, we index the IO_s data type by the size of the source and a target heap, reminiscent of the technique used to type stack operations [24]. Every constructor then explicitly records how it modifies the heap. Our *runIOState* function than becomes total – our types guarantee that it is impossible to access unallocated memory. We have implemented these ideas in Agda 2 [27]. Using such a dependently typed system, we can even correctly handle a heterogeneous heap, storing different types of data, and well-scoped, well-typed pointers. We defer the discussion of this implementation to further work.

6.3 Total semantics for concurrency

The above remarks about mutable state are also pertinent for our semantics of concurrency. A total implementation of the *step* function in Listing 11 function should forbid programmers from accessing unallocated memory.

A more serious problem, however, is that the heart of our semantics for concurrency, the *interleave* function in Listing 12, uses general recursion. Whenever a blocked thread is scheduled, no progress is made, and an unguarded recursive call is made. If there is a deadlock, however, we will continue scheduling blocked threads in the hope that some thread will make progress, and the *interleave* function loops.

Fortunately, we can avoid this problem by detecting deadlocks. Instead of hoping that our scheduler will indeed find a thread that can make progress, we should keep track of threads that we know are blocked. We sketch the idea here, but omit the details of our implementation.

We begin by changing the return type of our $runIO_c$ function to *Maybe a*, using *Nothing* to represent a deadlock. Whenever we learn that a thread is blocked, we record its *ThreadId*. When every thread is either blocked or finished, and the main process cannot make progress, we are in a deadlock and return *Nothing*. Whenever any thread makes progress, we empty the *entire* list of blocked threads; a thread might be blocked because it is waiting for an *MVar* to be filled. If another thread makes progress, it may have filled the *MVar* our blocked thread was waiting on – thereby unblocking the original thread. By dynamically detecting deadlocks in this fashion, we claim our specification can be made total.

7. Related work

The idea of providing functional specifications of IO is hardly new. Early versions of the Haskell Report [29] contained an appendix with a functional specification of interactions with the operating system. Curiously, this appendix disappeared after the introduction of monads. Similar specifications have been proposed to teach semantics to undergraduates. Our proposal to use these specifications both for programming and reasoning, is an important step forwards.

This work has been influenced strongly by the semantics of Haskell's IO as described by Peyton Jones [28]. This semantics use a process calculus containing Haskell's purely functional core to silently evaluate pure functions as the need arises. While this work has been widely cited as an excellent tutorial on IO in Haskell, the semantics presented have, to the best of our knowledge, never been used on the scale of the examples we present here. Our specifications are intended to be more 'user-friendly.' They require no external mathematical system of reasoning, but rather present the semantics in terms with which programmers are already comfortable.

Besides Peyton Jones's work, there is a huge amount of research in the semantics of programming languages. Many of the problems we discuss here have been covered elsewhere. The semantics for mutable state are fairly well-understood. Our development of teletype IO was heavily influenced by earlier work on understanding IO in a purely functional language [13, 16, 19].

There are several papers that model concurrency within Haskell worth discussing separately. First of all, Claessen has described a 'concurrency monad transformer' [6]. Using continuation passing ingenously, he shows how to add interleaved computations in any monad. The monad transformer he describes can even model *MVars*. While continuations are very expressive, it can be rather difficult to reason about them. This makes it a bit less suitable to reason with, when compared to our approach.

Harrison shows how the *resumption monad* can be used to interleave stateful computations [15]. To interleave computations he introduces a pair of mutual recursive functions: *sched* and *handler*. The *sched* function is a round robin scheduler that essentially consults the thread soup and passes the next active thread to the handler. The handler processes the active thread and invokes the *sched* function again. We feel that our separation of interleaving and processing threads makes it easier to extend the semantics with new functions, such as *killThread* and *yield*, without having to worry about interleaving. Harrison mentions that 'it is not the intention of the current work to model the awkward squad,' and does not explore this line of research further.

Finally, Nanevski *et al.* have proposed a type theory that allows programmers to reason about effectful programs [25, 26]. Instead of giving a concrete implementation of the specification as we have done here, they formulate several axioms that characterise how effectful programs behave. Both approaches have their merits and further research is warranted to fully understand how they relate.

8. Further work

There are two important issues that we hope to address in future work: composing the individual semantic models and proving their validity.

Composing semantics

Although we have discussed the semantics of several members of the awkward squad separately, the real challenge involves combining these semantics. We do have good hope that there is a certain degree of modularity we can exploit.

Combining arbitrary monads is a difficult problem and still subject to active research. Besides monad transformers [21] and distributivity laws [20], more recent work has focused on combining monads by taking their coproduct [22]. Unfortunately, the general formula to compute the coproduct of two monads is rather difficult.

The monads we have described so far, however, all have the same shape: constructors for every supported operation; and a single *Return* constructor. This general pattern is known as a *free monad*. To compute the coproduct of free monads, we only need to collect all the constructors for the supported operations by taking their coproduct. Previous work on modular interpreters [21] describes how to do so, while minimalizing the overhead associated with finding the right injection into a large coproduct.

One major advantage of composing such monads using their coproduct, is that we can compose the semantics of such constructs – that is we could construct a function run_{c+s} that will assign semantics to a program in IO_{c+s} that uses both concurrency and mutable state. Essentially, this involves pasting together the state associated with individual semantics, such as the scheduler and heap, and allowing each operation to update the relevant pieces.

This would greatly refine the current state of affairs, in which the colossal *IO* monad jumbles together all these separate issues. Refining the IO monad is one of the major open problems Peyton Jones identifies in his retrospective on Haskell's development [30]. This is really a problem – if you have a value of type *IO* () it could do everything from format your hard drive to print `"Hello World!"` – it's a bit worrying that we really have no idea of what kind of side-effects such a value has.

Correctness

Although we have defined semantics for teletype IO, mutable state, and concurrency, we cannot be sure that the models we have constructed are indeed a faithful representation of the real side-effects. We need to guarantee that that the semantics we have presented here can actually be trusted.

We could try prove that our semantics are equivalent to those presented by Peyton Jones [28]. One problem with this is that Peyton Jones's semantics are not completely formal – there is no specification of how pure functions should be silently evaluated. Moreover, this still does not guarantee that our specifications are semantically equivalent to the code produced by any Haskell compiler, but merely proves the two sets of semantics are equivalent.

An alternative approach would be to describe how Haskell compiles to code for some (virtual) machine. We could then compare the behaviour of the primitive *putChar* with the *putChar* we have defined ourselves. If these two are semantically equivalent on the machine level, we know that it is safe to reason using the functions we have defined. Hutton and Wright take a very similar approach to proving the correctness of a compiler for a simple language with exceptions [18].

This is actually an instance of a much wider problem: how many compilers have been proven to satisfy a language's specification? We have provided a very usable specification of effects, but can only share the burden of proof together with compiler implementors. There is an enormous gap between theory and practice that we cannot hope to bridge unilaterally.

9. Conclusions

We feel that this work has several significant merits. We conclude by reiterating why we believe this approach to semantics for the awkward squad is worth pursuing further:

Simplicity In contrast to process calculi, operational and denotational semantics, you don't need a theoretical background to understand these functional semantics. A programmer can use them to debug or test impure code, without having a deep mathematical understanding of all the issues involved.

Transparency One of the joys of Haskell is that there is no magic. Once someone understands higher order functions and algebraic data types, they could almost write the entire Prelude. Using these functional semantics, there is no need to lose this transparency.

Tool Support There are a large number of tools available to test, trace, and debug Haskell code [7, 32, 12]. Such tools typically do not cope well with functions in the IO monad. By constructing a faithful model within the programming language, such tools could be used to debug our pure model – a massive improvement over the status quo!

Granularity We have presented a fine grained semantics for pieces of the IO monad. Further refinement could really pay off. A case in point is made by Haskell's software transactional memory [14]. The distinction between the STM monad and the IO monad make sure that transactions can roll back. Similarly, we can guarantee that a teletype interaction of type IO_{tt} will never cause a deadlock in a concurrent process of type IO_c – the *type* of a side-effecting function suddenly means something.

Mutable state, concurrency, and teletype IO are considered beasts of programming language design by the purest of functional programmers. With the advent of monads, these issues have become managable – monads contain the havoc that such beasts can wreak. The semantics we present here take things one step further – there is no longer a hard distinction between pure and impure functions. There is, perhaps, beauty in these beasts after all.

Acknowledgments

We are greatly indebted to our colleagues in the Foundations of Programming group for their encouragement and entertaining discussions. Diana Fulger, Peter Hancock, Graham Hutton, Mauro Jaskelioff, Andres Löh, and Nicolas Oury all deserve a particular mention for their valuable feedback on earlier versions of this paper, for which we are forever grateful. We would also like to express our gratitude for the helpful feedback we received from anonymous referees.

References

[1] Arthur I. Baars and S. Doaitse Swierstra. Typing Dynamic Typing. In *ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[3] Richard Bird. Functional Pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, 2001.

[4] Richard Bird and Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988.

[5] James Cheney and Ralf Hinze. A Lightweight Implementation of Generics and Dynamics. In Manuel Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM-Press, October 2002.

[6] Koen Claessen. A Poor Man's Concurrency Monad. In *Journal of Functional Programming*, volume 9, pages 313–323. Cambridge University Press, May 1999.

[7] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000.

[8] Koen Claessen and John Hughes. Testing Monadic Code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, 2002.

[9] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and Loose Reasoning is Morally Correct. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–217, 2006.

[10] Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: an approach to high-assurance microkernel development. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, 2006.

[11] Neil Ghani, Peter Hancock, and Dirk Pattinson. Continuous Functions on Final Coalgebras. *Electronic Notes in Theoretical Computer Science*, 164(1):141–155, 2006.

[12] Andy Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Proceedings of the 4th Haskell Workshop*, 2000.

[13] Andrew D. Gordon. *Functional Programming and Input/Output*. PhD thesis, University of Cambridge, 1992.

[14] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[15] William L. Harrison. The Essence of Multitasking. In Michael Johnson and Varmo Vene, editors, *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2006.

[16] Sören Holmström. PFL: A Functional Language for Parallel Programming. In *Declarative Programming Workshop*, pages 114–139, 1983.

[17] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2003.

[18] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*. Springer, 2004.

[19] Kent Karlsson. Nebula: A Functional Operating System. Technical report, Chalmers University of Technology, 1981.

[20] David J. King and Philip Wadler. Combining monads. In John Launchbury and Patrick M. Sansom, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, pages 134–143, Glasgow, 1992. Springer.

[21] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, 1995.

[22] Christoph Lüth and Neil Ghani. Composing Monads Using Coproducts. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, 2002.

[23] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[24] James McKinna and Joel Wright. A type-correct, stack-safe, provably correct, expression compiler in Epigram. Submitted to the Journal of Functional Programming, 2006.

[25] Aleksandar Nanevski and Greg Morrisett. Dependent type theory of stateful higher-order functions. Technical Report TR-24-05, Harvard University, 2005.

[26] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of th Eleventh ACM SIGPLAN Internation Conference on Functional Programming*, 2006.

[27] Ulf Norell. Agda II. Available online.

[28] Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, 2002.

[29] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[30] Simon Peyton Jones. Wearing the hair shirt: a retrospective on Haskell. Invited talk at the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2003.

[31] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference record of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[32] Bernard Pope. Declarative debugging with Buddha. In *Summer School on Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[33] D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

A. Appendix

Listing 16 An implementation of queues using mutable references

```
data Data = Cell Int IORef | NULL
type Queue = (IORef, IORef)

emptyQueue :: IO_s Queue
emptyQueue = do
  front ← newIORef NULL
  back ← newIORef NULL
  return (front, back)

enqueue :: Queue → Int → IO_s ()
enqueue (front, back) x =
  do  newBack ← newIORef NULL
      let cell = Cell x newBack
      c ← readIORef back
      writeIORef back cell
      case c of
        NULL → writeIORef front cell
        Cell y t → writeIORef t cell

dequeue :: Queue → IO_s (Maybe Int)
dequeue (front, back) = do
  c ← readIORef front
  case c of
    NULL → return Nothing
    (Cell x nextRef) → do
      next ← readIORef nextRef
      writeIORef front next
      return (Just x)
```