# Programming with Arrows

John Hughes

Department of Computer Science and Engineering,
Chalmers University of Technology,
S-41296 Sweden.

## 1 Introduction

### 1.1 Point-free programming

Consider this simple Haskell definition, of a function which counts the number of occurrences of a given word `w` in a string:

```
count w = length . filter (==w) . words
```

This is an example of "point-free" programming style, where we build a function by composing others, and make heavy use of higher-order functions such as `filter`. Point-free programming is rightly popular: used appropriately, it makes for concise and readable definitions, which are well suited to equational reasoning in the style of Bird and Meertens [2]. It's also a natural way to assemble programs from components, and closely related to connecting programs via pipes in the UNIX shell.

Now suppose we want to modify `count` so that it counts the number of occurrences of a word in a *file*, rather than in a string, and moreover prints the result. Following the point-free style, we might try to rewrite it as

```
count w = print . length . filter (==w) . words . readFile
```

But this is rejected by the Haskell type-checker! The problem is that `readFile` and `print` have side-effects, and thus their types involve the `IO` monad:

```
readFile :: String -> IO String
print :: Show a => a -> IO ()
```

Of course, it is one of the *advantages* of Haskell that the type-checker can distinguish expressions with side effects from those without, but in this case we pay a price. These functions simply have the wrong types to compose with the others in a point-free style.

Now, we can write a point-free definition of this function using combinators from the standard `Monad` library. It becomes:

```
count w = (>>=print) .
          liftM (length . filter (==w) . words) .
          readFile
```

But this is no longer really perspicuous. Let us see if we can do better.

In Haskell, functions with side-effects have types of the form `a -> IO b`. Let us introduce a type synonym for this:

```
type Kleisli m a b = a -> m b
```

So now we can write the types of `readFile` and `print` as

```
readFile :: Kleisli IO String String
print :: Show a => Kleisli IO a ()
```

We parameterise `Kleisli` over the `IO` monad because the same idea can be used with any other one, and we call this type `Kleisli` because functions with this type are arrows in the Kleisli category of the monad `m`.

Now, given two such functions, one from `a` to `b`, and one from `b` to `c`, we can "compose" them into a Kleisli arrow from `a` to `c`, combining their side effects in sequence. Let us define a variant composition operator to do so. We choose to define "reverse composition", which takes its arguments in the opposite order to `(.)`, so that the order in which we write the arrows in a composition corresponds to the order in which their side effects occur.

```
(>>>) :: Monad m =>
           Kleisli m a b -> Kleisli m b c -> Kleisli m a c
(f >>> g) a = do b <- f a
                 g b
```

We can use this composition operator to define functions with side-effects in a point-free style — for example, the following function to print the contents of a file:

```
printFile = readFile >>> print
```

Returning to our original example, we cannot yet reprogram it in terms of `(>>>)` because it also involves functions *without* side-effects, and these have the wrong type to be composed by `(>>>)`. Fortunately, we can easily convert a pure function of type `a -> b` into a Kleisli arrow with no side-effects. We define a combinator to do so, and we call it `arr`.

```
arr :: Monad m => (a->b) -> Kleisli m a b
arr f = return . f
```

Using this combinator, we can now combine side-effecting and pure functions in the same point-free definition, and solve our original problem in the following rather clear way:

```
count w = readFile >>>
          arr words >>> arr (filter (==w)) >>> arr length >>>
          print
```

### 1.2 The `Arrow` class

Now we have two ways to write point-free definitions: using functions and composition, or Kleisli arrows and arrow composition. We can unify them by *overloading* the arrow operators, so that the same operations can be used with both. To do so, we introduce an `Arrow` class with `arr` and `(>>>)` as methods:

```
class Arrow arr where
  arr :: (a -> b) -> arr a b
  (>>>) :: arr a b -> arr b c -> arr a c
```

It is trivial to define an instance for the function type:

```
instance Arrow (->) where
  arr = id
  (>>>) = flip (.)
```

But in order to make Kleisli arrows an instance, we have to make `Kleisli` a new type rather than a type synonym:

```
newtype Kleisli m a b = Kleisli {runKleisli :: a -> m b}
```

We can now declare

```
instance Monad m => Arrow (Kleisli m) where ...
```

where the method definitions are those we have already seen, modified to add and remove the `Kleisli` constructor appropriately.

The extra constructor clutters our definitions a little. We must now redefine `count` as

```
count w = Kleisli readFile >>>
          arr words >>> arr (filter (==w)) >>> arr length >>>
          Kleisli print
```

and invoke it via

```
runKleisli (count w) filename
```

rather than simply `count w filename`, but this is a small price to pay for a uniform notation. Indeed, Jansson and Jeuring used arrows in the derivation of matched parsers and prettyprinters *purely* for the notational benefits in equational proofs [11]. This notation is available in any Haskell program just by importing the hierarchical library `Control.Arrow`, which defines `Arrow`, `Kleisli`, and a number of other useful classes.

Now that we have defined an `Arrow` class, it's natural to ask if we can find other interesting instances — and indeed we can. Here, for example, is the arrow of *stream functions*:

```
newtype SF a b = SF {runSF :: [a] -> [b]}
```

The arrow operations are defined as follows:

```
instance Arrow SF where
  arr f = SF (map f)
  SF f >>> SF g = SF (f >>> g)
```

and might be used like this:

```
StreamFns> runSF (arr (+1)) [1..5]
[2,3,4,5,6]
```

Just like monads, arrow types are useful for the *additional* operations they support, over and above those that every arrow provides. In the case of stream functions, one very useful operation is to *delay* the stream by one element, adding a new element at the beginning of the stream:

```
delay x = SF (x:)
```

The `delay` arrow might be used like this:

```
StreamFns> runSF (delay 0) [1..5]
[0,1,2,3,4,5]
```

It will appear many times in examples below.

Most applications of arrows do not, in fact, use the function or Kleisli arrows — they use other instances of the `Arrow` class, which enables us to program in the same, point-free way with other kinds of objects. In real applications an arrow often represents some kind of a *process*, with an input channel of type `a`, and an output channel of type `b`. The stream functions example above is perhaps the simplest case of this sort, and will be developed in some detail.

### 1.3 Arrows as computations

We are used to thinking of monads as modelling computations, but monads are used in two distinct ways in Haskell. On the one hand, the `IO` and `ST` monads provide a referentially transparent interface to imperative operations at a lower level. On the other hand, monads used in libraries of parsing combinators, for example, help to structure purely functional code, with not an imperative operation in sight. In this second kind of example, the `Monad` class is used as a *shared interface* that many different combinator libraries can provide.

Why bother to share the same interface between many combinator libraries? There are several important reasons:

- We know from experience that the `Monad` interface is a good one. The library designer who chooses to use it knows that it will provide users with a powerful tool.
- The library designer can exploit tools for implementing the interface systematically — monad transformers nowadays make it easy to construct complex implementations of class `Monad` [13], thus reducing the library author's work.

– We can write overloaded code that works with many different libraries — the functions in the standard `Monad` library are good examples. Such code provides free functionality that the library author need neither design nor implement.
– When a shared interface is sufficiently widely used, it can even be worthwhile to add specific language support for using it. Haskell's `do` syntax does just this for monads.
– Library users need learn less to use a new library, if a part of its interface is already familiar.

These are compelling advantages — and yet, the monadic interface suffers a rather severe restriction. While a monadic program can produce its *output* in many different ways — perhaps not at all (the `Maybe` monad), perhaps many times (the list monad), perhaps by passing it to a continuation — it takes its *input* in just one way: via the parameters of a function.

We can think of arrows as computations, too. The `Arrow` class we have defined is clearly analogous to the usual `Monad` class — we have a way of creating a pure computation without effects (`arr`/`return`), and a way of sequencing computations (`(>>>)`/`(>>=)`). But whereas monadic computations are parameterised over the type of their output, but not their input, arrow computations are parameterised over both. The way monadic programs take input cannot be varied by varying the monad, but arrow programs, in contrast, can take their input in many different ways depending on the particular arrow used. The stream function example above illustrates an arrow which takes its input in a different way, as a stream of values rather than a single value, so this is an example of a kind of computation which cannot be represented as a monad.

Arrows thus offer a competing way to represent computations in Haskell. But their purpose is not to replace monads, it is to bring the benefits of a shared interface, discussed above, to a wider class of computations than monads can accomodate. And in practice, this often means computations that represent processes.

### 1.4 Arrow laws

One aspect of monads we have not touched on so far, is that they satisfy the so-called *monad laws* [26]. These laws play a rather unobtrusive rôle in practice — since they do not appear explicitly in the code, many programmers hardly think about them, much less prove that they hold for the monads they define. Yet they are important: it is the monad laws that allow us to write a sequence of operations in a **do** block, without worrying about how the sequence will be bracketed when it is translated into binary applications of the monadic bind operator. Compare with the associative law for addition, which is virtually never explicitly used in a proof, yet underlies our notation every time we write $a+b+c$ without asking ourselves what it means.

Arrows satisfy similar laws, and indeed, we have already implicitly assumed the associativity of `(>>>)`, by writing arrow compositions without brackets!

Other laws tell us, for example, that `arr` distributes over `(>>>)`, and so the definition of `count` we saw above,

```
count w = Kleisli readFile >>>
          arr words >>> arr (filter (==w)) >>> arr length >>>
          Kleisli print
```

is equivalent to

```
count w = Kleisli readFile >>>
          arr (words >>> filter (==w) >>> length) >>>
          Kleisli print
```

Now, it would be very surprising if this were *not* the case, and that illustrates another purpose of such laws: they help us avoid "surprises", where a slight modification of a definition, that a programmer would reasonably expect to be equivalent to the original, leads to a different behaviour. In this way laws provide a touchstone for the implementor of an arrow or monad, helping to avoid the creation of a design with subtle traps for the user. An example of such a design would be a "monad" which measures the cost of a computation, by counting the number of times bind is used. It is better to define a separate operation for consuming a unit of resource, and let bind just combine these costs, because then the monad laws are satisfied, and cosmetic changes to a monadic program will not change its cost.

Nevertheless, programmers do sometimes use monads which do *not* satisfy the stated laws. Wadler's original paper [26] introduced the "strictness monad" whose only effect is to force sequencing to be strict, but (as Wadler himself points out), the laws are not satisfied. Another example is the random generation "monad" used in our QuickCheck [4] testing tool, with which terms equivalent by the monad laws may generate different random values — but with the same distribution. There is a sense in which both these examples "morally" satisfy the laws, so that programmers are not unpleasantly surprised by using them, but strictly speaking the laws do not hold.

In the same way, some useful arrow instances may fail to satisfy the arrow laws. In fact, the stream functions we are using as our main example fail to do so, without restrictions that we shall introduce below. In this case, if we drop the restrictions then we may well get unpleasant surprises when we use stream function operations later.

Despite the importance of the arrow laws, in these notes I have chosen to de-emphasize them. The reason is simple: while monads can be characterised by a set of three laws, the original arrows paper states twenty [10], and Paterson's tutorial adds at least seven more [18]. It is simply harder to characterise the expected behaviour of arrows equationally. I have therefore chosen to focus on understanding, using, and implementing the arrow interface, leaving a study of the laws for further reading. Either of the papers cited in this paragraph is a good source.

## 2 The Arrow classes

As we already noted, the monadic interface is a powerful one, which enables programmers to build a rich library of operations which work with any monad. There is an important difference between the monadic interface, and the `Arrow` class that we have seen so far, that has a major impact on how arrows can be implemented and used. Compare the types of the sequencing operators for monads and arrows:

```
class Monad m where
  (>>=) :: m b -> (b -> m c) -> m c
  ...
class Arrow arr where
  (>>>) :: arr a b -> arr b c -> arr a c
  ...
```

In the case of monads, the second argument of `(>>=)` is a Haskell function, which permits the user of this interface to use all of Haskell to map the result of the first computation to the computation to be performed next. Every time we sequence two monadic computations, we have an opportunity to run arbitrary Haskell code in between them. But in the case of arrows, in contrast, the second argument of `(>>>)` is just an arrow, an element of an abstract datatype, and the only things we can do in that arrow are things that the abstract data type interface provides. Certainly, the `arr` combinator enables us to have the output of the first arrow passed to a Haskell function — but this function is a *pure* function, with the type `b -> c`, which thus has no opportunity to perform further effects. If we want the *effects* of the second arrow to depend on the output of the first, then we must construct it using operations other than `arr` and `(>>>)`.

Thus the simple `Arrow` class that we have already seen is *not* sufficiently powerful to allow much in the way of useful overloaded code to be written. Indeed, we will need to add a plethora of other operations to the arrow interface, divided into a number of different classes, because not all useful arrow types can support all of them. Implementing all of these operations makes defining a new arrow type considerably more laborious than defining a new monad — but there is another side to this coin, as we shall see later. In the remainder of this section, we will gradually extend the arrow interface until it *is* as powerful as the monadic one.

### 2.1 Arrows and pairs

Suppose we want to sequence two computations delivering integers, and add their results together. This is simple enough to do with a monad:

```
addM a b = do x <- a
              y <- b
              return (x+y)
```

But the arrow interface we have seen so far is not even powerful enough to do this!

Suppose we are given two arrows `f` and `g`, which output integers from the same input. If we could make a pair of their outputs, then we could supply that to `arr (uncurry (+))` to sum the components, and define

```
addA :: Arrow arr => arr a Int -> arr a Int -> arr a Int
addA f g = f_and_g >>> arr (uncurry (+))
```

But clearly, there is no way to define `f_and_g` just in terms of `f`, `g`, `(>>>)` and `arr`. Any composition of the form `... >>> f >>> ...` loses all information other than `f`'s output after the appearance of `f`, and so neither `g`'s output nor the input needed to compute it can be available afterwards.

We therefore add an operator to construct `f_and_g` to the arrow interface:

```
class Arrow arr where
  ...
  (&&&) :: arr a b -> arr a c -> arr a (b,c)
```

which enables us to define `addA` by

```
addA f g = f &&& g >>> arr (uncurry (+))
```

(The composition operator binds less tightly than the other arrow operators).

The new operator is simple to implement for functions and Kleisli arrows:

```
instance Arrow (->) where
  ...
  (f &&& g) a = (f a, g a)

instance Monad m => Arrow (Kleisli m) where
  ...
  Kleisli f &&& Kleisli g = Kleisli $ \a -> do b <- f a
                                               c <- g a
                                               return (b,c)
```

For stream functions, we just zip the output streams of `f` and `g` together. We can conveniently use the arrow operators on *functions* to give a concise point-free definition!

```
instance Arrow SF where
  ...
  SF f &&& SF g = SF (f &&& g >>> uncurry zip)
```

As an example, here is a stream function which maps a stream to a stream of pairs, by pairing together each input value and its predecessor:

```
pairPred = arr id &&& delay 0
```

Running `pairPred` on an example gives

```
StreamFns> runSF (arr id &&& delay 0) [1..5]
[(1,0),(2,1),(3,2),(4,3),(5,4)]
```

The `(&&&)` operator is convenient to use, but it is not the simplest way to add this functionality to the arrow interface. Arrow types can be complex to implement, and as we observed above, there are many operations that need to be defined. To make implementing arrows as lightweight as possible, it is important to dissect each combinator into parts which are the same for each arrow type, and so can be implemented once and for all, and the minimal functionality that must be reimplemented for each new `Arrow` instance. In this case, the `(&&&)` operator, among other things, duplicates the input so it can be fed to both arguments. Duplication can be performed using `arr (\x->(x,x))`, so we factor this out and define `(&&&)` in terms of a simpler operator `(***)`:

```
f &&& g = arr (\x->(x,x)) >>> f *** g
```

The new operator in turn is added to the `Arrow` class:

```
class Arrow arr where
  ...
  (***) :: arr a b -> arr c d -> arr (a,c) (b,d)
```

The combination `f *** g` constructs an arrow from pairs to pairs, that passes the first components through `f`, and the second components through `g`.

Now, `(***)` turns out not to be the simplest way to provide this functionality either. It combines *two* arrows into an arrow on pairs, but we can obtain the same functionality using a combinator that just lifts *one* arrow to an arrow on pairs. We therefore introduce the combinator `first`, which lifts an arrow to operate on pairs by feeding just the first components through the given arrow, and leaving the second components untouched. Its type is

```
class Arrow arr where
  ...
  first :: arr a b -> arr (a,c) (b,c)
```

Its implementations for functions, Kleisli arrows, and stream functions are:

```
instance Arrow (->) where
  ...
  first f (a,c) = (f a,c)

instance Monad m => Arrow (Kleisli m) where
  ...
  first (Kleisli f) = Kleisli (\(a,c) -> do b <- f a
                                            return (b,c))

instance Arrow SF where
  first (SF f) = SF (unzip >>> first f >>> uncurry zip)
```

If we had taken `(***)` as primitive, then we could have defined `first` by

```
first f = f *** arr id
```

But we can instead define `(***)` in terms of `first`, by first defining

```
second :: Arrow arr => arr a b -> arr (c,a) (c,b)
second f = arr swap >>> first f >>> arr swap
  where swap (x,y) = (y,x)
```

which lifts an arrow to work on the second components of pairs, and then defining

```
f *** g = first f >>> second g
```

This definition also has the advantage that it clarifies that the effects of `f` come before the effects of `g`, something that up to this point has been unspecified.

The `Arrow` class defined in `Control.Arrow` includes all of these combinators as methods, with the definitions given here as defaults. That permits an implementor to declare an instance of this just be defining `arr`, `(>>>)` and `first`. It also permits implementors to give specialised definitions of *all* the arrow operations, which in general will be more efficient. In that case, the specialised definitions should, of course, respect the semantics of those given here. An implementation of `first` is often only half the size of a corresponding implementation of `(***)` or `(&&&)`, and so, at least in the earlier stages of development, the simplification made here is well worth while.

### 2.2 Arrows and conditionals

The combinators in the previous section allow us to combine the results from several arrows. But suppose we want to make a choice between two arrows, on the basis of a previous result? With the combinators we have seen so far, every arrow in a combination is always "invoked" — we cannot make any arrow conditional on the output of another. We will need to introduce further combinators to make this possible.

At first sight, one might expect to introduce a combinator modelling an "if-then-else" construct, perhaps

```
ifte :: Arrow arr => arr a Bool -> arr a b -> arr a b -> arr a b
```

where `ifte p f g` uses `p` to compute a boolean, and then chooses between `f` and `g` on the basis of its output. But once again, we can simplify this combinator considerably.

First of all, we can easily factor out `p` by computing its result *before* the choice: we can do so with `p &&& arr id`, which outputs a pair of the boolean and the original input. We would then define `ifte` by

```
ifte p f g = p &&& arr id >>> f ||| g
```

where `f ||| g` chooses between `f` and `g` on the basis of the first component of the pair in its input, passing the second component on to `f` or `g`. But we can do better than this: note that the input type of `f ||| g` here, `(Bool,a)`, carries the same information as `Either a a`, where `(True,a)` corresponds to `Left a`, and `(False,a)` to `Right a`. If we use an `Either` type as the input to the choice operator, rather than a pair, then the `Left` and `Right` values can carry *different* types of data, which is usefully more general. We therefore define

```
class Arrow arr => ArrowChoice arr where
  (|||) :: arr a c -> arr b c -> arr (Either a b) c
```

Note the duality between `(|||)` and `(&&&)` — if we reverse the order of the parameters of `arr` in the type above, and replace `Either a b` by the pair type `(a,b)`, then we obtain the type of `(&&&)`! This duality between choice and pairs recurs throughout this section. As we will see later, not all useful arrow types can support the choice operator; we therefore place it in a new subclass of `Arrow`, so that we can distinguish between arrows with and without a choice operator.

As an example of using conditionals, let us see how to define a map function for arrows:

```
mapA :: ArrowChoice arr => arr a b -> arr [a] [b]
```

The definition of `mapA` requires choice, because we must choose between the base and recursive cases on the basis of the input list. We shall express `mapA` as *base-case* `|||` *recursive-case*, but first we must convert the input into an `Either` type. We do so using

```
listcase []     = Left ()
listcase (x:xs) = Right (x,xs)
```

and define `mapA` by

```
mapA f = arr listcase >>>
         arr (const []) ||| (f *** mapA f >>> arr (uncurry (:)))
```

where we choose between immediately returning `[]`, and processing the head and tail, then consing them together. We will see examples of using `mapA` once we have shown how to implement `(|||)`.

Notice first that `f ||| g` requires that `f` and `g` have the same output type, which is a little restrictive. Another possibility is to allow for different output types, and combine them into an `Either` type by tagging `f`'s output with `Left`, and `g`'s output with `Right`. We call the operator that does this `(+++)`:

```
class Arrow arr => ArrowChoice arr where
  ...
  (+++) :: arr a b -> arr c d -> arr (Either a c) (Either b d)
```

Now observe that `(+++)` is to `(|||)` as `(***)` is to `(&&&)`: in other words, we can easily define the latter in terms of the former, and the former is (marginally)

simpler to implement. Moreover, it is dual to `(***)` — just replace `Either` types by pairs again, and swap the parameters of `arr`. In this case the definition of `(|||)` becomes

```
f ||| g = f +++ g >>> arr join
  where join (Left b)  = b
        join (Right b) = b
```

Now, just as `(***)` combined two arrows into an arrow on pairs, and could be defined in terms of a simpler combinator which lifted *one* arrow to the first components of pairs, so `(+++)` can be defined in terms of a simpler operator which just lifts an arrow to the *left* summand of an `Either` type. Therefore we introduce

```
class Arrow arr => ArrowChoice arr where
  ...
  left :: arr a b -> arr (Either a c) (Either b c)
```

The idea is that `left f` passes inputs tagged `Left` to `f`, passes inputs tagged `Right` straight through, and tags outputs from `f` with `Left`. Given `left`, we can then define an analogous combinator

```
right f = arr mirror >>> left f >>> arr mirror
  where mirror (Left a)  = Right a
        mirror (Right a) = Left a
```

and combine them to give a definition of `(+++)` in terms of simpler combinators:

```
f +++ g = left f >>> right g
```

Just as in the previous section, the definition of the `ArrowChoice` class in `Control.Arrow` includes all of these combinators (except `ifte`), with the definitions given here as defaults. Thus one can make an arrow an instance of `ArrowChoice` just by implementing `left`, or alternatively give specialised definitions of all the combinators for greater efficiency.

Choice is easy to implement for functions and Kleisli arrows:

```
instance ArrowChoice (->) where
  left f (Left a) = Left (f a)
  left f (Right b) = Right b

instance Monad m => ArrowChoice (Kleisli m) where
  left (Kleisli f) = Kleisli (\x ->
    case x of
      Left a -> do b <- f a
                   return (Left b)
      Right b -> return (Right b))
```

With these definitions, `mapA` behaves like `map` for functions, and `mapM` for Kleisli arrows[1]:

```
StreamFns> mapA (arr (+1)) [1..5]
[2,3,4,5,6]
StreamFns> runKleisli (mapA (Kleisli print) >>> Kleisli print)
                      [1..5]
1
2
3
4
5
[(),(),(),(),()]
```

But what about stream functions?

Implementing `left` for stream functions is a little trickier. First of all, it is clear that the input `xs` is a list of tagged values, from which all those tagged with `Left` should be extracted and passed to the argument stream function, whose outputs should be retagged with `Left`:

```
map Left (f [a | Left a <- xs])
```

Moreover, all the elements of `xs` tagged `Right` should be copied to the output. *But how should the* `Left` *and* `Right` *values be merged into the final output stream?*

There is no single "right answer" to this question. We shall choose to restrict our attention to synchronous stream functions, which produce exactly one output per input[2]. With this assumption, we can implement `left` by including one element of `f`'s output in the combined output stream every time an element tagged `Left` appears in the input. Thus:

```
instance ArrowChoice SF where
  left (SF f) = SF (\xs -> combine xs (f [y | Left y <- xs]))
    where combine (Left y:xs) (z:zs) = Left z: combine xs zs
          combine (Right y:xs) zs = Right y: combine xs zs
          combine [] zs = []
```

In fact, the restriction we have just made, to length-preserving stream functions, turns out to be necessary not only to define `left`, but also to ensure the good behaviour of `first`. The definition of `first` we gave in the previous section does not in general satisfy the "arrow laws" formulated in [10], which means that it occasionally behaves in surprising ways — but the laws *are* satisfied under the restriction to length-preserving functions.

---

[1] Here the second expression to be evaluated is split across several lines for readability, which is of course not allowed by Hugs or GHCi.

[2] The `delay` arrow is clearly problematic, but don't worry! We shall see how to fix this shortly.

The only stream function arrow we have seen so far which does *not* preserve the length of its argument is `delay` — the delayed stream has one more element than the input stream. Recall the definition we saw earlier:

```
delay x = SF (x:)
```

In order to meet our new restriction, we redefine `delay` as

```
delay x = SF (init . (x:))
```

This does not change the behaviour of the examples we saw above.

As an example of using choice for stream functions, let us explore how `mapA` behaves for this arrow type. It is interesting to map the `delay` arrow over a stream of lists:

```
StreamFns> runSF (mapA (delay 0)) [[1,2,3],[4,5,6],[7,8,9]]
[[0,0,0],[1,2,3],[4,5,6]]
```

Even more interesting is a stream of lists of different lengths:

```
StreamFns> runSF (mapA (delay 0))
             [[1,2,3],[4,5],[6],[7,8],[9,10,11],[12,13,14,15]]
[[0,0,0],[1,2],[4],[6,5],[7,8,3],[9,10,11,0]]
```

If we arrange the input and output streams as tables,

| | |
|---|---|
| 1 2 3 | 0 0 0 |
| 4 5 | 1 2 |
| 6 | 4 |
| 7 8 | 6 5 |
| 9 10 11 | 7 8 3 |
| 12 13 14 15 | 9 10 11 0 |

then we can see that the *shape* of the table output correponds to the shape of the input, but the elements in each column form a stream delayed by one step, where the gaps in the columns are ignored.

As another example, consider the following arrow which delays a list by passing the head straight through, and recursively delaying the tail by one more step.

```
delaysA = arr listcase >>>
            arr (const []) |||
            (arr id *** (delaysA >>> delay []) >>>
             arr (uncurry (:)))
```

Running this on an example gives

```
StreamFns> runSF delaysA [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
[[1],[4,2],[7,5,3],[10,8,6]]
```

or, laid out as tables,

| | |
|---|---|
| 1 2 3 | 1 |
| 4 5 6 | 4 2 |
| 7 8 9 | 7 5 3 |
| 10 11 12 | 10 8 6 |

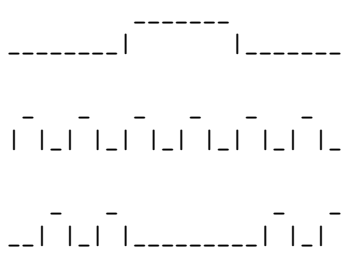
We can see that each column is delayed by a different amount, with missing entries represented by the empty list.

### 2.3 Arrows and feedback

Stream functions are useful for simulating synchronous circuits. For example, we could represent a NOR-gate by the arrow

```
nor :: SF (Bool,Bool) Bool
nor = arr (not.uncurry (||))
```

and simulate it by using `runSF` to apply it to a list of pairs of booleans. In this section we shall visualise such lists by drawing signals as they might appear on an oscilloscope, so a test of `nor` might produce this output:

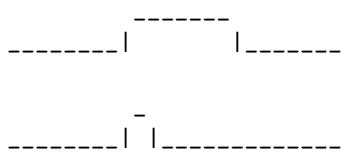


Here the top two signals are the input, and the bottom one the output. As we would expect, the output is high only when both inputs are low. (The ASCII graphics are ugly, but easily produced by portable Haskell code: a function which does so is included in the appendix).

Synchronous circuits contain delays, which we can simulate with the `delay` arrow. For example, a rising edge detector can be modelled by comparing the input with the same signal delayed one step.

```
edge :: SF Bool Bool
edge = arr id &&& delay False >>> arr detect
  where detect (a,b) = a && not b
```

Testing this arrow might produce



where a pulse appears in the output at each rising edge of the input.

Now, by connecting two NOR-gates together, one can build a flip-flop (see Figure 1). A flip-flop takes two inputs, SET and RESET, and produces two outputs, one of which is the negation of the other. As long as both inputs remain low, the outputs remain stable, but when the SET input goes high, then the first output does also, and when the RESET input goes high, then the first output goes low. If SET and RESET are high simultaneously, then the flip-flop becomes unstable. A flip-flop is made by connecting the output of each NOR-gate to one input of the other; the remaining two inputs of the NOR-gates are the inputs of the flip-flop, and their outputs are the outputs of the flip-flop.

**Fig. 1.** A flip-flop built from two NOR-gates.

To represent the flip-flop as an arrow, we need to *feed back* the outputs of the NOR-gates to their inputs. To make this possible, we introduce a new arrow class with a feedback combinator:

```
class Arrow arr => ArrowLoop arr where
  loop :: arr (a,c) (b,c) -> arr a b
```

The intention is that the component of type `c` in the output of the argument arrow is fed back to become the second component of its input. For example, for ordinary functions `loop` just creates a simple recursive definition:

```
instance ArrowLoop (->) where
  loop f a = b
    where (b,c) = f (a,c)
```

Feedback can also be implemented for Kleisli arrows over monads which are instances of the `MonadFix` class, but we omit the details here. Instead, let us implement feedback for stream functions. We might expect to use the following definition:

```
instance ArrowLoop SF where
  loop (SF f) = SF $ \as ->
      let (bs,cs) = unzip (f (zip as cs)) in bs
```

which closely resembles the definition for functions, making a recursive definition of the feedback stream `cs`. However, this is just a little too strict. We would of course expect `loop (arr id)` to behave as `arr id` (with an undefined feedback stream), and the same is true of `loop (arr swap)`, which feeds its input through the feedback stream to its output. But with the definition above, both these loops are undefined. The problem is the recursive definition of `(bs,cs)` above: the functions `unzip` and `zip` are both strict — they must evaluate their arguments before they can return a result — and so are `arr id` and `arr swap`, the two functions we are considering passing as the parameter `f`, with the result that the value to be bound to the pattern `(bs,cs)` cannot be computed until the value of `cs` is known! Another way to see this is to remember that the semantics of a recursive definition is the limit of a sequence of approximations starting with the undefined value, $\bot$, and with each subsequent approximation constructed by evaluating the right hand side of the definition, with the left hand side bound to the previous one. In this case, when we initially bind `(bs,cs)` to $\bot$ then both `bs` and `cs` are bound to $\bot$, but now because `zip` is strict then `zip as` $\bot=\bot$, because `f` is strict then `f` $\bot=\bot$, and because `unzip` is strict then `unzip` $\bot=\bot$. So the second approximation, `unzip (f (zip as` $\bot$ `))`, is also $\bot$, and by the same argument so are all of the others. Thus the limit of the approximations is undefined, and the definition creates a "black hole".

To avoid this, we must ensure that `cs` is *not* undefined, although it may be a stream of undefined elements. We modify the definition as follows:

```
instance ArrowLoop SF where
  loop (SF f) = SF $ \as ->
      let (bs,cs) = unzip (f (zip as (stream cs))) in bs
    where stream ~(x:xs) = x:stream xs
```

The `~` in the definition of `stream` indicates Haskell's *lazy pattern matching* — it delays matching the argument of `stream` against the pattern `(x:xs)` until the bound variables `x` and `xs` are actually used. Thus `stream` returns an infinite list *without* evaluating its argument — it is only when the *elements* of the result are needed that the argument is evaluated. Semantically, `stream` $\bot=\bot:\bot:\bot:\ldots$. As a result, provided `as` is defined, then so is `zip as (stream` $\bot$`)` — it is a list of pairs with undefined second components. Since neither `f` nor `unzip` needs these components to deliver a defined result, we now obtain defined values for `bs` and `cs` in the second approximation, and indeed the limit of the approximations is the result we expect. The reader who finds this argument difficult should work out the sequence of approximations in the call `runSF (loop (arr swap)) [1,2,3]` — it is quite instructive to do so.

Note that `stream` itself is not a length-preserving stream function: its result is always infinite, no matter what its argument is. But `loop` respects our restriction to synchronous stream functions, because `zip` always returns a list as long as

its *shorter* argument, which in this case is `as`, so the lists bound to `bs` and `cs` always have the same length as `as`.

Returning to the flip-flop, we must pair two NOR-gates, take their outputs and duplicate them, feeding back one copy, and supplying each NOR-gate with one input and the output of the other NOR-gate as inputs. Here is a first attempt:

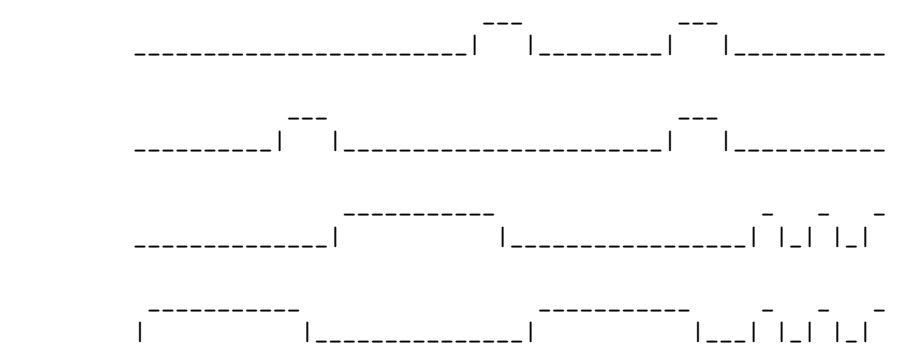
```
flipflop =
  loop (arr (\((reset,set),(c,d)) -> ((set,d),(reset,c))) >>>
        nor *** nor >>>
        arr id &&& arr id)
```

The first line takes the external inputs and fed-back outputs and constructs the inputs for each NOR-gate. The second line invokes the two NOR-gates, and the third line duplicates their outputs.

Unfortunately, this definition is circular: the $i$th output depends on itself. To make a working model of a flip-flop, we must add a delay. We do so as follows:

```
flipflop =
  loop (arr (\((reset,set),~(c,d)) -> ((set,d),(reset,c))) >>>
        nor *** nor >>>
        delay (False,True) >>>
        arr id &&& arr id)
```

which initialises the flip-flop with the first output low. We must also ensure that the loop body is not strict in the loop state, which explains the lazy pattern matching on the first line. Note that the `delay` in the code delays a *pair* of bits, and so corresponds to *two* single-bit delays in the hardware, and the feedback path in this example passes through both of them (refer to Figure 1). This makes the behaviour a little less responsive, and we must now trigger the flip-flop with pulses lasting at least two cycles. For example, one test of the flip-flop produces this output:



Here the first two traces are the RESET and SET inputs, and the bottom two are the outputs from the flip-flop. Initially the first output is low, but when the SET input goes high then so does the output. It goes low again when the RESET input goes high, then when both inputs go high, the flip-flop becomes unstable.

The `ArrowLoop` class, together with instances for functions and Kleisli arrows, is included in `Control.Arrow`. Ross Paterson has also suggested overloading `delay`, and placing it in an `ArrowCircuit` class, but this has not (yet) found its way into the standard hierarchical libraries.

### 2.4 Higher-order arrows

What about higher-order programming with arrows? Can we construct arrows which receive other arrows in their input, and invoke them? We cannot, using the combinators we have already seen, but we can, of course, add a new class to make this possible. We introduce an arrow analogue of the "apply" function:

```
class Arrow arr => ArrowApply arr where
  app :: arr (arr a b, a) b
```

Instances for functions and Kleisli arrows are easy to define:

```
instance ArrowApply (->) where
  app (f,x) = f x

instance Monad m => ArrowApply (Kleisli m) where
  app = Kleisli (\(Kleisli f,x) -> f x)
```

but there is no reasonable implementation for stream functions. We shall see why shortly.

First of all, note that both `first` and `left` are easy to implement in terms of `app` (the details are left as an exercise). So `app` is a strictly more powerful operator to provide. We have also seen that we can base a Kleisli arrow on *any* monad, and we can implement all of `first`, `left` and `app` for such a type. In fact, `app` is so powerful that we can reconstruct a monad from any arrow type which supports it! We represent a computation of an `a` as an arrow from the empty tuple to `a`:

```
newtype ArrowMonad arr a = ArrowMonad (arr () a)
```

We can now define `return` and `(>>=)` as follows:

```
instance ArrowApply a => Monad (ArrowMonad a) where
  return x = ArrowMonad (arr (const x))
  ArrowMonad m >>= f =
    ArrowMonad (m >>>
                arr (\x-> let ArrowMonad h = f x in (h, ())) >>>
                app)
```

The second argument of `(>>=)` is a function returning an arrow, which we turn into an arrow outputting an arrow (`h`) using `arr`; we then need `app` to invoke the result.

Thus we have finally fulfilled our promise at the beginning of this section, to extend the arrow interface until it is as powerful as — indeed, equivalent to —

the monadic one. We can now do everything with arrows that can be done with monads — if need be, by converting our arrow type into a monad. Yet this is only a Pyrrhic victory. If we want to do "monadic" things, it is much simpler to define a monad directly, than to first define all the arrow operations, and then build a monad on top of them.

The conclusion we draw is that arrows that support `app` are of relatively little interest! Apart from the benefits of point-free notation, we might as well use a monad instead. The truly *interesting* arrow types are those which do *not* correspond to a monad, because it is here that arrows give us real extra generality. Since we know that stream functions cannot be represented as a monad, then they are one of these "interesting" arrow types. So are the arrows used for functional reactive programming, for building GUIs, and the arrows for discrete event simulation we present in Section 5. And since these arrows cannot be represented by a monad, we know that they cannot support a sensible definition of `app` either.

### 2.5 Exercises

1. **Filtering.** Define

   ```
   filterA :: ArrowChoice arr => arr a Bool -> arr [a] [a]
   ```

   to behave as `filter` on functions, and like `filterM` on Kleisli arrows. Experiment with running

   ```
   filterA (arr even >>> delay True)
   ```

   on streams of lists of varying lengths, and understand its behaviour.
2. **Stream processors.** Another way to represent stream processors is using the datatype

   ```
   data SP a b = Put b (SP a b) | Get (a -> SP a b)
   ```

   where `Put b f` represents a stream processor that is ready to output `b` and continue with `f`, and `Get k` represents a stream processor waiting for an input `a`, which will continue by passing it to `k`. Stream processors can be interpreted as stream functions by the function

   ```
   runSP (Put b s) as = b:runSP s as
   runSP (Get k) (a:as) = runSP (k a) as
   runSP (Get k) [] = []
   ```

   Construct instances of the classes `Arrow`, `ArrowChoice`, `ArrowLoop`, and `ArrowCircuit` for the type `SP`.
   - You are provided with the module `Circuits`, which defines the class `ArrowCircuit`.
   - You should find that you can drop the restriction we imposed on stream functions, that one output is produced per input — so `SP` arrows can represent *asynchronous* processes.

– On the other hand, you will encounter a tricky point in defining `first`. How will you resolve it?
– Check that your implementation of `loop` has the property that the arrows `loop (arr id)` and `loop (arr swap)` behave as `arr id`:

```
SP> runSP (loop (arr id)) [1..10]
[1,2,3,4,5,6,7,8,9,10]
SP> runSP (loop (arr swap)) [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

Module `Circuits` also exports the definition `flipflop` above, together with sample input `flipflopInput` and a function `showSignal` which visualises tuples of lists of booleans as the "oscilloscope traces" we saw above.

```
SP> putStr$ showSignal$ flipflopInput
                           ___           ___
_________________________|   |_________|   |___________

           ___                           ___
__________|   |_________________________|   |___________
```

Use these functions to test a flipflop using your stream processors as the underlying arrow type. The behaviour should be the same as we saw above.

## 3 Pointed arrow programming

We began these notes by arguing the merits of a point-free programming style. Yet although point-free programming is often concise and readable, it is not *always* appropriate. In many cases it is clearer to give names to the values being manipulated, and in ordinary Haskell programs we do not hesitate to do so. This "pointed" style is also well-supported in monadic programs, by the `do` notation. But what if we want to use a pointed style in an arrow-based program? With the combinators we have seen so far, it is quite impossible to do so.

In fact, an extension to the language is needed to make pointed arrow programming possible. Ross Paterson designed such an extension [17], and implemented it using a preprocessor, and it was built into GHC in version 6.2. Paterson's extension is comparable to the **do** notation for monads — it is quite possible to program without it, but some programs are much nicer with it. But just as arrows have a more complex interface than do monads, so Paterson's notation is more complex than the **do**, and its translation much more complicated. The complexity of the translation, of course, makes the notation all the more valuable. In this section we will explain how to use this extension.

### 3.1 Arrow abstractions

Paterson's extension adds a new form of expression to Haskell: the *arrow abstraction*, of the form `proc pat -> body`. Arrow abstractions are analogous to

$\lambda$-expressions: they denote an arrow, and bind a name (or, in general, a pattern) to the arrow input, which may then be used in the body of the abstraction.

However, the body of an arrow abstraction is not an expression: it is of a new syntactic category called a *command*. Commands are designed to help us construct just those arrows that can be built using the arrow combinators — but in a sometimes more convenient notation.

The simplest form of command, `f -< exp`, can be thought of as a form of "arrow application" — it feeds the value of the expression `exp` to the arrow `f`. (The choice of notation will make more sense shortly). Thus, for example, an AND-gate with a delay of one step could be expressed by

```
proc (x,y) -> delay False -< x && y
```

This is equivalent to `arr (\(x,y) -> x&&y) >>> delay False`. Arrow abstractions with a simple command as their body are translated as follows,

$$\texttt{proc pat -> a -< e} \longrightarrow \texttt{arr (\pat -> e) >>> a}$$

and as this translation suggests, arrow-bound variables (such as `x` and `y` in the AND-gate example above) are *not in scope* to the left of the `-<`. This is an easy way to see that `proc` could not possibly be implemented by a combinator taking a $\lambda$-expression as an argument: the scopes of arrow-bound variables do not correspond to the scopes of $\lambda$-bound variables.

This scope rule rules out arrow abstractions such as

```
proc (f,x) -> f -< x
```

which is rejected, because it translates to `arr (\(f,x)->x) >>> f`, in which `f` is used outside its scope. As usual, if we want to apply an arrow received as input, we must use `app`:

```
proc (f,x) -> app -< (f,x)
```

The arrow notation does offer syntactic sugar for this as well:

```
proc (f,x) -> f -<< x
```

However, this is of little importance, since there is little reason to use arrows with `app` — one might as well use the equivalent monad instead.

Pointed arrow notation really comes into its own when we start to form *compound commands* from simpler ones. For example, suppose we want to feed inputs to either the arrow `f` or `g`, depending on the value of a predicate `p`. Using the arrow combinators, we would need to encode the choice using `Left` and `Right`, in order to use the choice combinator `(|||)`:

```
arr (\x -> if p x then Left x else Right x) >>> f ||| g
```

Using the pointed arrow notation, we can simply write

```
proc x -> if p x then f -< x else g -< x
```

Here the `if...then...else...` is a *conditional command*, which is translated as follows:

```
proc pat -> if e then c1 else c2
⟶
arr (\pat -> if e then Left pat else Right pat) >>>
(proc pat -> c1 ||| proc pat -> c2)}
```

Note that the scope of `x` in the example now has *two* holes: the arrow-valued expressions before each arrow application.

Even in the simple example above, the pointed notation is more concise than the point-free. When we extend this idea to `case` commands, its advantage is even greater. Recall the definition of `mapA` from section 2.2:

```
mapA f = arr listcase >>>
         arr (const []) ||| (f *** mapA f >>> arr (uncurry (:)))
  where listcase [] = Left ()
        listcase (x:xs) = Right (x,xs)
```

We were obliged to introduce an encoding function `listcase` to convert the case analysis into a choice between `Left` and `Right`. Clearly, a case analysis with more cases would require an encoding into nested `Either` types, which would be tedious in the extreme to program. But all these encodings are generated automatically from `case`-commands in the pointed arrow notation. We can reexpress `mapA` as

```
mapA f = proc xs ->
  case xs of
    [] -> returnA -< []
    x:xs' -> (f *** mapA f >>> uncurry (:)) -< (x,xs')
```

which is certainly more readable.

Just as in the monadic notation, we need a way to express just delivering a result without any effects: this is the rôle of `returnA -< []` above, which corresponds to `arr (const [])` in the point-free code. In fact, `returnA` *is* just an arrow, but a trivial arrow with no effects: it is defined to be `arr id`. We could have written this `case` branch as `arr (const []) -< xs`, which would correspond exactly to the point-free code, but it is clearer to introduce `returnA` instead.

### 3.2 Naming intermediate values

Just as in monadic programs, it is convenient to be able to name intermediate values in arrow programs. This is the primary function of Haskell's `do` notation for monads, and it is provided for arrows by a similar `do` notation. However, while a monadic `do` block is an expression, an arrow `do` block is a command, and can thus only appear inside an arrow abstraction. Likewise, while the statements `x <- e` in a monadic `do` bind a name to the result of an expression `e`, the arrow

form binds a name to the output of a command. As a simple example, we can reexpress the `printFile` arrow of the introduction as

```
printFile = proc name ->
  do s <- Kleisli readFile -< name
     Kleisli print -< s
```

in which we name the string read by `readFile` as `s`. And now at last the choice of `-<` as the arrow application operator makes sense — it is the tail feathers of an arrow! A binding of the form `x <- f -< e` looks suggestively as though `e` is being fed through an arrow labelled with `f` to be bound to `x`!

As another example, recall the rising edge detector from section 2.3:

```
edge :: SF Bool Bool
edge = arr id &&& delay False >>> arr detect
  where detect (a,b) = a && not b
```

We can give a name to the intermediate, delayed value by using a `do` block:

```
edge = proc a -> do
         b <- delay False -< a
         returnA -< a && not b
```

Notice that both `a` and `b` are in scope after the binding of `b`, although they are bound at different places. Thus a binding is translated into an arrow that *extends the environment*, by pairing the bound value with the environment received as input. The translation rule is

```
proc pat -> do x <- c1          ⟶  (arr id &&& proc pat -> c1) >>>
               c2                   proc (pat,x) -> c2
```

where we see clearly which variables are in scope in each command. Applying the rule to this example, the translation of the pointed definition of `edge` is

```
edge = (arr id &&& (arr (\a->a) >>> delay False)) >>>
       (arr (\(a,b) -> a && not b) >>> returnA)
```

which can be simplified by the arrow laws to the point-free definition we started with, bearing in mind that `arr (\a->a)` and `returnA` are both the identity arrow, and thus can be dropped from compositions. In practice, GHC can and does optimise these translations, discarding unused variables from environments, for example. But the principle is the one illustrated here.

Note that the same variable occupies different positions in the environment in different commands, and so different occurrences must be translated differently. The arrow notation lets us use the *same* name for the *same value*, no matter where it occurs, which is a major advantage.

We can use the `do` notation to rewrite `mapA` in an even more pointed form. Recall that in the last section we redefined it as

```
mapA f = proc xs ->
  case xs of
    [] -> returnA -< []
    x:xs' -> (f *** mapA f >>> uncurry (:)) -< (x,xs')
```

Here the second branch of the case is still expressed in a point-free form. Let us use `do` to name the intermediate results:

```
mapA f = proc xs ->
  case xs of
    [] -> returnA -< []
    x:xs' -> do y <- f -< x
                ys' <- mapA f -< xs'
                returnA -< y:ys
```

We are left with a definition in a style which closely resembles ordinary monadic programming.

**Fig. 2.** A full adder built from half-adders.

When used with the stream functions arrow, the pointed notation can be used to express circuit diagrams with named signals very directly. For example, suppose that a half-adder block is available, simulated by

```
halfAdd :: Arrow arr => arr (Bool,Bool) (Bool,Bool)
halfAdd = proc (x,y) -> returnA -< (x&&y, x/=y)
```

A full adder can be constructed from a half adder using the circuit diagram in Figure 2. From the diagram, we can read off how each component maps input signals to output signals, and simply write this down in a `do` block.

```
fullAdd :: Arrow arr => arr (Bool,Bool,Bool) (Bool,Bool)
fullAdd = proc (x,y,c) -> do
            (c1,s1) <- halfAdd -< (x,y)
            (c2,s2) <- halfAdd -< (s1,c)
            returnA -< (c1||c2,s2)
```

The arrow code is essentially a net-list for the circuit. Without the pointed arrow notation, we would have needed to pass `c` past the first half adder, and `c1` past the second, explicitly, which would have made the dataflow much less obvious.

### 3.3 Recursive arrow bindings

Of course, this simple scheme doesn't work if the circuit involves feedback. We have already seen an example of such a circuit: the flipflop of section 2.3. We repeat its circuit diagram again in Figure 3. In section 2.3 we represented this diagram as an arrow as follows:

**Fig. 3.** The flip-flop, again.

```
flipflop =
  loop (arr (\((reset,set),~(c,d)) -> ((set,d),(reset,c))) >>>
        nor *** nor >>>
        delay (False,True) >>>
        arr id &&& arr id)
```

The arrow `do` syntax provides syntactic sugar for an application of `loop`: a group of bindings can be preceded by `rec` to make them recursive using `loop`. In this example, we can define `flipflop` instead by

```
flipflop :: ArrowCircuit arr => arr (Bool,Bool) (Bool,Bool)
flipflop = proc (reset,set) -> do
             rec c <- delay False -< nor reset d
                 d <- delay True -< nor set c
             returnA -< (c,d)
  where nor a b = not (a || b)
```

As always with stream functions, we must insert enough `delay`s to ensure that each stream element is well-defined (in this example, one `delay` would actually suffice). In this case also, the pointed definition is more straightforward than the point-free one. Its relationship to the circuit diagram is much more obvious.

### 3.4 Command combinators

Of course, once we have introduced commands, it is natural to want to define *command combinators*. But since commands are not first-class expressions, it appears at first that we cannot define combinators that take commands as arguments. However, commands *do* denote arrows from environments to outputs, and these are first-class values. The pointed arrow notation therefore provides a mechanism for using *arrow combinators* as command combinators.

However, we cannot use just any arrow combinator as a command combinator. The commands that appear in pointed arrow notation denote arrows with types of the form `arr env a`, where `env` is the type of the environment that the command appears in, and `a` is the type of its output. Now, when we apply a command combinator, then all of the commands we pass to it naturally occur in the *same* environment, which is moreover the same environment that the combinator itself appears in. Thus the type of a command combinator should have the form

```
arr env a -> arr env b -> ... -> arr env c
```

That is, all the arrows passed to it as arguments should have the *same* input type, which moreover should be the same input type as the arrow produced.

For example, the pairing combinator

```
(&&&) :: Arrow arr => arr a b -> arr a c -> arr a (b,c)
```

has just such a type (where `a` is the type of the environment), while in contrast

```
(|||) :: Arrow arr => arr a c -> arr b c -> arr (Either a b) c
```

does not. We can indeed use `(&&&)` as an operator on commands:

```
example = proc x ->
       do returnA -< x
   &&& do delay 0 -< x
```

is equivalent to `returnA &&& delay 0`. Running the example gives

```
Main> runSF example [1..5]
[(1,0),(2,1),(3,2),(4,3),(5,4)]
```

(One trap for the unwary is that this syntax is ambiguous: in the example above, `&&&` could be misinterpreted as a part of the expression following `returnA -<`. The `do`s in the example are there to prevent this. Because of the layout rule, it is clear in the example above that `&&&` is not a part of the preceding command.)

When a command combinator is not an infix operator, then applications are enclosed in banana brackets to distinguish them from ordinary function calls. Thus (since `(&&&)` is not an infix operator), we could also write the example above as

```
example = proc x -> (| (&&&) (returnA -< x) (delay 0 -< x) |)
```

The translation of banana brackets is just

`proc pat -> (| e c1...cn |)` $\longrightarrow$ `e (proc pat->c1)...(proc pat->cn)`

Lovers of higher-order programming (such as ourselves!) will of course wish to define combinators which not only accept and return commands, but *parameterised* commands, or "functions returning commands", if you will. The pointed arrow notation supports this too. Parameterised commands are represented as arrows with types of the form `arr (env, a) b`, where `env` is the type of the environment and the `b` the type of the result, as usual, but `a` is the type of the parameter. Such parameterised commands can be constructed using lambda-notation: the command `\x -> cmd` is translated into an arrow taking a pair of the current environment and a value for `x`, to the output of `cmd`. The translation rule is (roughly)

`proc pat -> \x -> c` $\longrightarrow$ `proc (pat,x) -> c`

Likewise the command `cmd e` can be thought of as supplying such a parameter: `cmd` is expected to be a command taking a pair of an environment and value as input, and `cmd e` is a command which just takes the environment as input, and extends the environment with the output of `e` before passing it to `cmd`. The translation rule is (roughly)

`proc pat -> c e` $\longrightarrow$ `arr (\pat -> (pat,e)) >>> proc pat -> c`

See the GHC documentation for more details on the translation.

As an example, let us define a command combinator corresponding to `map`. The arrow map we have already defined,

```
mapA :: ArrowChoice arr => arr a b -> arr [a] [b]
```

is not suitable, because it changes the type of the input in a way which is not allowable. Let us instead pair the input of both the argument and result arrow with the environment:

```
mapC :: ArrowChoice arr => arr (env,a) b -> arr (env,[a]) [b]
mapC c = proc (env,xs) ->
  case xs of
    [] -> returnA -< []
    x:xs' -> do y <- c -< (env,x)
                ys <- mapC c -< (env,xs')
                returnA -< y:ys
```

With this type, `mapC` can be used as a command combinator. To apply it, we have to first apply the command combinator `mapC` to a command abstraction (inside banana brackets), and then apply the resulting command to a suitable list (no banana brackets). For example,

```
example2 = proc (n,xs) ->
  (| mapC (\x-> do delay 0 -< n
                   &&& do returnA -< x) |) xs
```

is a stream function whose input stream contains pairs of numbers and lists, and which pairs each element of such a list (using `mapC`) with a *delayed* number from the foregoing input stream element. It can be run as follows:

```
Main> runSF example2 [(1,[1,2]),(3,[4]),(5,[6,7])]
[[(0,1),(0,2)],[(1,4)],[(3,6),(3,7)]]
```

The example may seem a little contrived, but its purpose is to illustrate the behaviour when the argument of `mapC` refers *both* to its parameter and a free variable (`n`). A much more useful example of this can be found in Exercise 4d below.

Taken together, these extensions provide us with a comprehensive notation for pointed programming with arrows, which leads to programs which, superficially at least, resemble monadic programs very closely.

### 3.5 Exercises

1. **Adder.** An $n$-bit adder can be built out of full adders using the design shown in Figure 4, which adds two numbers represented by lists of booleans, and delivers a sum represented in the same way, and a carry bit. Simulate this design by defining

```
adder :: Arrow arr => Int ->
           arr ([Bool],[Bool]) ([Bool],Bool)
```

Represent the inputs and the sum with the *most* significant bit first in the list.

**Fig. 4.** A 3-bit adder built from full adders.

**Fig. 5.** A bit-serial adder.

2. **Bit serial adder.** A bit-serial adder can be constructed from a full adder using feedback, by the circuit in Figure 5. The inputs are supplied to such an adder one bit at a time, starting with the least significant bit, and the sum is output in the same way. Model this circuit with an arrow

   ```
   bsadd :: ArrowCircuit arr => arr (Bool,Bool) Bool
   ```

   Use the `rec` arrow notation to obtain feedback. The `showSignal` function from module `Circuits` may be useful again for visualising the inputs and outputs.
3. **Filter.**
   (a) Define
   ```
   filterA :: ArrowChoice arr => arr a Bool -> arr [a] [a]
   ```
   again (as in exercise 1 in section 2.5), but this time use the pointed arrow notation.
   (b) Now define a *command combinator* `filterC`:
   ```
   filterC :: ArrowChoice arr =>
              arr (env,a) Bool -> arr (env,[a]) [a]
   ```
   and test it using the following example:
   ```
   test :: Show a => Kleisli IO [a] [a]
   test = proc xs -> (|filterC (\x->Kleisli keep-<x)|) xs
     where keep x = do putStr (show x++"? ")
                       s <- getLine
                       return (take 1 s == "y")
   ```
   Running this example might yield:
   ```
   Main> runKleisli (test3 >>> Kleisli print) [1..3]
   1? y
   2? n
   3? y
   [1,3]
   ```

**Fig. 6.** A row of f.

4. **Counters.**
   (a) One of the useful circuit combinators used in the Lava hardware description environment [15] is `row`, illustrated in Figure 6. Let us represent `f` by an arrow from pairs to pairs,
   ```
   f :: arr (a,b) (c,d)
   ```
   with the components representing inputs and outputs as shown:

   Define a command combinator
   ```
   rowC :: Arrow arr =>
             Int ->
               arr (env,(a,b)) (c,a) ->
                 arr (env,(a,[b])) ([c],a)
   ```
   to implement the connection pattern in the Figure.

**Fig. 7.** A 1-bit counter.

   (b) A one-bit counter can be constructed using the circuit diagram in Figure 7. Implement this as an arrow using the pointed arrow notation:
   ```
   counter1bit :: ArrowCircuit arr => arr Bool (Bool,Bool)
   ```
   (c) An $n$-bit counter can be built by connecting $n$ 1-bit counters in a row, with the carry output of each counter connected to the input of the next. (Note that, in this case, the vertical input of the row is unused). Implement an $n$-bit counter using your `rowC` command combinator.
   (d) In practice, it must be possible to *reset* a counter to zero. Modify your one bit counter so that its state can be reset, changing its type to
   ```
   counter1bit :: ArrowCircuit arr =>
                    arr (Bool,Bool) (Bool,Bool)
   ```
   to accomodate a reset input. Now modify your $n$-bit counter to be resettable also. This kind of modification, which requires feeding a new signal to every part of a circuit description, would be rather more difficult without the arrow notation.

## 4 Implementing arrows

We began section 2 by comparing the types of the sequencing operators for monads and arrows respectively:

```
class Monad m where
  (>>=) :: m b -> (b -> m c) -> m c
  ...
class Arrow arr where
  (>>>) :: arr a b -> arr b c -> arr a c
  ...
```

As we observed then, the fact that the second argument of `(>>=)` is a Haskell function gives the user a great deal of expressivity "for free" — to obtain similar expressivity with the arrows interface, we have to introduce a large number of further operations on the arrow types. However, there is another side to this coin. When we *implement* a monad, the `(>>=)` can do nothing with its second argument except apply it — the `(>>=)` operator is forced to treat its second operand as abstract. When we implement an arrow, on the other hand, the `(>>>)` can inspect the representation of its second operand and make choices based on what it finds. Arrows can carry *static information*, which is available to the arrow combinators, and can be used to implement them more efficiently.

This was, in fact, the original motivation for developing arrows. Swierstra and Duponcheel had developed a parsing library, in which the representation of parsers included a list of starting symbols, and the choice combinator made use of this information to avoid backtracking (which can incur a heavy space penalty by saving previous states in case it is necessary to backtrack to them) [24]. Their parsing library, in contrast to many others, did not — indeed, could not — support a monadic interface, precisely because the starting symbols of `f >>= g` cannot be determined in general without knowing the starting symbols of both `f` *and* `g`, and the latter are not available to `(>>=)`. It was the search for an interface with similar generality to the monadic one, which *could* be supported by Swierstra and Duponcheel's library, which led to my proposal to use arrows. However, since parsing arrows have been discussed in several papers already, we will take other examples in these notes.

### 4.1 Optimising composition

For some arrow types (such as stream transformers), composition is quite expensive. Yet in some cases it can be avoided, by applying arrow laws. For example, one law states that

```
arr f >>> arr g = arr (f >>> g)
```

but whereas the left hand side involves composition of arrows, the right hand side involves only composition of functions. Replacing the left hand side by the right may thus be an optimisation — in the case of stream functions, it corresponds to optimising `map f.map g` to `map (f.g)`.

Suppose, now, that we represent arrows constructed by `arr` specially, so that the implementation of `(>>>)` can recognise them. Then we can apply this optimisation dynamically, every time a composition of two such pure arrows is constructed!

To implement this optimisation in as general a form as possible, we define a new arrow type `Opt arrow`, based on an underlying arrow type `arrow`, which represents arrows of the form `arr f` specially.

```
data Opt arrow a b = Arr (a->b)
                   | Lift (arrow a b)
```

Thus an `Opt arrow` arrow is either the special case `arr f`, or just contains an arrow of the underlying type. We can recover an underlying arrow from an `Opt arrow` arrow using the function

```
runOpt (Arr f)  = arr f
runOpt (Lift f) = f
```

Now we implement the `Arrow` operations as follows:

```
instance Arrow arrow => Arrow (Opt arrow) where
  arr = Arr
  Arr f >>> Arr g = Arr (f>>>g)
  f >>> g = Lift (runOpt f>>>runOpt g)
  first (Arr f) = Arr (first f)
  first (Lift f) = Lift (first f)
```

We implement `arr` just by building the special case representation, and each of the other operations by first testing for the special case and optimising if possible, then falling through into a general case which just converts to the underlying arrow type and applies the same operation there. The other arrow classes can be implemented in a similar way. Yampa [8], perhaps the most extensive application of arrows, makes use of just such an optimisation — with the addition of another special form for constant arrows (of the form `arr (const k)`).

This is an example of an *arrow transformer* or *functor*, analogous to a monad transformer. In his experimental arrow library [16], Paterson defines a class of arrow transformers:

```
class (Arrow a, Arrow (f a)) => ArrowTransformer f a where
  lift :: a b c -> f a b c
```

(The library contains a number of useful transformers, corresponding to many of the well-known monad transformers, plus several which make sense only for arrows). We can declare `Opt` to be an arrow transformer very simply:

```
instance Arrow arrow => ArrowTransformer Opt arrow where
  lift = Lift
```

This idea can be taken further, of course. As it stands, the optimisation of composition is applied only if compositions are bracketed correctly — a term such as `(Lift f >>> arr g) >>> arr h` would not be optimised, because the leftmost composition constructs the arrow `Lift (f >>> arr g)`, which cannot be optimised by the rightmost composition.

Let us see how to improve this, by taking advantage of the associativity of composition. Rather than representing optimised arrows as one of the forms `arr f` or `lift g`, let us use the forms `arr f` and `arr f >>> lift g >>> arr h`. The advantage of this latter form is that we can keep the "pure part" of each arrow separate on both the left and the right, thus making it possible to optimise compositions with pure arrows on both sides.

We need existential types to represent these forms in a data structure, because in the form `arr f >>> lift g >>> arr h` then the arrow `g` may have any type at all. The `Opt` type becomes

```
data Opt arrow a b = Arr (a->b)
                   | forall c d. ALA (a->c) (arrow c d) (d->b)
```

(where `ALA` stands for `arr-lift-arr`: a value of the form `ALA f g h` represents `arr f >>> lift g >>> arr h`). We lift underlying arrows to this type by pre- and post-composing with the identity arrow:

```
instance ArrowTransformer Opt arrow where
  lift f = ALA id f id
```

We convert back to the underlying arrow type in the obvious way:

```
runOpt :: Arrow arrow => Opt arrow a b -> arrow a b
runOpt (Arr f) = arr f
runOpt (ALA pre f post) = arr pre >>> f >>> arr post
```

The implementations of the arrow operations just take advantage of the known pure parts to construct compositions of *functions*, rather than the underlying arrow type, wherever possible.

```
instance Arrow arrow => Arrow (Opt arrow) where
  arr = Arr

  Arr f >>> Arr g = Arr (f>>>g)
  Arr f >>> ALA pre g post = ALA (f >>> pre) g post
  ALA pre f post >>> Arr g = ALA pre f (post >>> g)
  ALA pre f post >>> ALA pre' g post' =
    ALA pre (f >>> arr (post>>>pre') >>> g) post'

  first (Arr f) = Arr (first f)
  first (ALA pre f post) =
    ALA (first pre) (first f) (first post)
```

In the code above, the *only* composition at the underlying arrow type takes place in the last equation defining `(>>>)`, when two impure arrows are composed. Writing out the left and right hand sides as their interpretations, this equation reads

```
arr pre >>> f >>> arr post >>> arr pre' >>> g >>> arr post'
=
arr pre >>> (f >>> arr (post >>> pre') >>> g) >>> arr post'
```

and we see that, even in this case, one arrow composition is converted into a composition of functions (`post` and `pre'`).

However, as it stands this implementation is not a good one, because every time we lift an underlying arrow to the `Opt` type, we insert two identity functions, which must later be composed with the underlying arrow. Lifting an arrow `f` and then running it actually introduces two new arrow compositions:

```
runOpt (lift f) = runOpt (ALA id f id)
                = arr id >>> f >>> arr id
```

Thus lifting computations into the "optimised" arrow type might well make them run slower. But we can avoid this overhead, in either of two different ways:

- We can introduce new constructors in the `Opt` type, to represent arrows of the form `lift f`, `lift f>>>arr g`, and `arr f>>>lift g`. With these additional constructors, no identity functions need be introduced, and so no spurious compositions are necessary.
- We can change the representation of the `pre` and `post` functions to make the identity function recognisable, so we can treat it as a special case.

The first solution is straightforward but slightly inelegant, since it leads to rather long and repetetive code. The second solution gives us an opportunity to define another arrow transformer, which optimises composition with the identity, using the same approach as in this section. One can go on to build optimising arrow transformers that implement more and more algebraic laws; we skip the details here. We remark only that the generalised algebraic data types recently added to GHC [20] are invaluable here, allowing us to define constructors such as

```
First :: Opt arrow a b -> Opt arrow (a,c) (b,c)
```

whose result type is not of the usual form, so that we can pattern match on, and thus optimise, arrows built using the `first` combinator.

## 5 Arrows for simulation

In the last section of these notes, we will carry through an extended example of defining a prototype arrow library with richer functionality. Staying with the application area of circuit simulation, we will construct a library which can perform more accurate timing simulations. The stream functions we used earlier perform a synchronous, or "cycle based" simulation, which proceeds in discrete steps, and computes the value of every signal at every step. This is why, when we defined the arrow operations, we could assume that each stream functions produced one output for each input. This kind of simulation is costly if the time represented by each step is short (because there must be many steps), and

inaccurate if it is long (because potentially many changes in a signal *during one step* must be approximated by a single value). In contrast, our new library will track *every* change in a signal's value, no matter how often they occur, but incur costs only when signals actually do change. It will implement a form of *discrete event* simulation.

To give a flavour of the library, here is how we might use it to simulate a flip-flop, in which the two NOR-gates have slightly different gate delays. The description of the flip-flop itself is not unlike those we have already seen — we shall see the details later. When we run the simulation, we provide a list of all the changes in the value of the input, with the times at which they occur:

```
Sim> runSim (printA "input " >>>
             cutoff 6.5 flipflop >>>
             printA "output")
       (False,False)
       [Event 1 (False,True), Event 2 (False,False), ...]
```

The arrow we are simulating contains "probes" of the form `printA s`, which behave as identity arrows but also print the values which pass through them. We also specify a time at which simulation should end. The output of the simulation tells us how the probed signals changed over time:

```
input : (False,False)@init
output: (False,True)@init
input : (False,True)@1.0
output: (False,False)@1.1
output: (True,False)@1.2100000000000002
input : (False,False)@2.0
input : (True,False)@3.0
output: (False,False)@3.11
output: (False,True)@3.21
input : (False,False)@4.0
input : (True,True)@5.0
output: (False,False)@5.1
input : (False,False)@6.0
output: (False,True)@6.1
output: (True,True)@6.109999999999999
output: (False,True)@6.209999999999999
output: (False,False)@6.2099999999999999
output: (False,True)@6.309999999999999
output: (True,True)@6.3199999999999985
output: (False,True)@6.419999999999998
output: (False,False)@6.4199999999999998
```

We can see that when the set or reset input goes high, the flip-flop responds by quickly setting the appropriate output high, after a brief moment in which both outputs are low. When both inputs go low, the output does not change, and no

event appears in the trace above. If both inputs are high simultaneously, and then drop, then the flip-flop becomes unstable and begins to oscillate, generating many output events although no further events appear in the input.

One application for this more accurate kind of simulation is power estimation for integrated circuits. Although the *behaviour* of such a circuit can be simulated by a synchronous simulation which just computes the *final* value of each signal on each clock cycle, the power consumption depends also on how many times each signal changes its value *during* a clock cycle. This is because much of the power in such a circuit is consumed charging and discharging wires, and if this happens several times during one clock cycle, the power consumed is correspondingly higher.

Like the stream functions arrow, our new simulation arrows represent a kind of process, transforming input events to output events. They are thus a rather typical kind of application for arrows, and useful to study for that reason. Moreover, this library (although much smaller) is closely related to the Yampa simulation library [8] — many design decisions are resolved in the same way, and the reader who also studies the internals of Yampa will find much that is familiar. However, there is also much that is different, since Yampa is not a discrete event simulator: Yampa simulations proceed in steps in which every signal is calculated, although the "sampling interval" between steps can vary.

### 5.1 Signals and Events

Abstractly, we think of the inputs and outputs of simulation arrows as *signals*, which are piecewise constant functions from time to values. Times may be any real number:

```
type Time = Double
```

and we explicitly allow negative times. We refer to a change in a signal's value as an *event*, and require the event times for each signal to form a (possibly infinite) increasing sequence. That is, each signal must have a first event before which it has a constant value no matter how far back in time we go, and after each event there must be a *next* event. We can thus represent a signal by its initial value, and the events at which its value changes[3]. We shall represent events by the type

```
data Event a = Event {time::Time, value::a}
```

The value of a signal at time `t` will be the initial value, if `t` is before the time of the first event, or the value of the last event at a time less than or equal to `t`.

[3] The reader might be tempted to treat the initial value as an event at time "minus infinity", to avoid introducing a special case. It is tempting, but it does not work well: at a number of places in the simulation code, the initial value of a signal *must* be treated differently from the later ones. Exactly the same design choice is made in Yampa, so there is safety in numbers!

This abstract view of signals fits our intended application domain well, but it is worth noting that we are ruling out at least one useful kind of signal: those which take a different value at a single point only. Consider an edge detector for example: when the input signal changes from `False` to `True`, we might expect the output signal to be `True` at the time of the change, but `False` immediately before and after. We cannot represent this: if the output signal takes the value `True` at all, then it must remain `True` throughout some non-zero interval — it would be non-sensical to say that the output signal has *both* a rising and a falling event at the same time.

Of course, a hardware edge detector must *also* keep its output high for a non-zero period, so our abstraction is realistic, but nevertheless in some types of simulation we might find it useful to allow instantaneously different signal values. For example, if we were simulating a car wash, we might want to represent the arrival of a car as something that takes place at a particular instant, rather than something that extends over a short period. What Yampa calls "events" are in fact precisely such signals: they are signals of a `Maybe` type whose value is `Nothing` except at the single instant where an event occurs. We could incorporate such signals into our model by associating *two* values with each event, the value at the instant of the event itself, and the value at later times, but for simplicity we have not pursued this approach.

### 5.2 Simulation arrows

We might expect a simulation arrow just to be a function from a list of input events to a list of output events, rather like the stream function arrows of Section 1.2. Unfortunately, this simple approach does not work at all. To see why not, consider a simulated adder, which combines two input integer signals to an output integer signal. Clearly, whenever an event occurs on *either* input, the adder must produce an event on its output with the new sum. So, to decide which output event to produce next, the adder must choose the *earliest* event from its next two input events. If these are supplied as lists of events, then this cannot be done without inspecting, and thus computing, *both* of them — and one of them may lie far in the simulated future. In the presence of feedback, it is disastrous if we cannot compute present simulated events without knowing future ones, since then events may depend on themselves and simulation may be impossible. Hence this approach fails. We will use an approach related to the *stream processors* of Exercise 2 instead.

Abstractly, though, we will think of a simulation arrow as a function from an input signal to an output signal. (In practice, we shall parameterise simulation arrows on a monad, but we ignore that for the time being). But we place two restrictions on these functions: they should respect *causality*, and be *time-invariant*.

Causality means that the output of an arrow at time `t` should not be affected by the value of its input at later times: the future may not affect the past. Causality is a vital property: it makes it possible to run a simulation from earlier

times to later ones, without the need to return to earlier times and revise the values of prescient signals when later events are discovered.

Time-invariance means that shifting the input signal of an arrow backwards or forwards in time should shift the output signal in the same way: the behaviour should not depend on the *absolute* time at which events occur. One important consequence is that the output of an arrow cannot change from its initial value until the input has (since any such event can depend only on the constant part of the input by causality, and so can by shifted arbitrarily later in time by time invariance). This relieves the simulator of the need to simulate all of history, from the beginning of time until the first input event, since we know that all signals must retain their initial values until that point.

We shall represent simulation arrows by functions from the initial input value, to the initial output value and a simulation state:

```
newtype Sim m a b = Sim (a -> m (b, State m a b))
```

This state will then evolve as the simulation proceeds, and (in general) it depends on the initial input value. We parameterise simulation arrows on a monad `m` (in our examples, the `IO` monad), so that it is possible to define probes such as `printA`, which we saw in the introduction to this section.

A running simulation can be in one of three states, which we represent using the following type:

```
data State m a b =
    Ready (Event b) (State m a b)
  | Lift (m (State m a b))
  | Wait Time (State m a b) (Event a -> State m a b)
```

Here

- `Ready e s` represents a simulation that is ready to output `e`, then behave like `s`,
- `Lift m` represents a simulation that is ready to perform a computation `m` in the underlying monad, then continue in the state that `m` delivers,
- `Wait t s k` represents a simulation that is waiting for an input event until time `t`: if an input arrives *before* time `t`, then it is passed to the continuation `k`, and if no input arrives before simulated time reaches `t` then the simulation changes to the "timeout" state `s`.

Conveniently, Haskell's `Double` type includes the value infinity,

```
infinity = 1/0
```

which behaves as a real number greater than all others. We can thus represent a simulation state which is waiting for ever using `Wait infinity`.

Given an initial input value and a list of input events, we run a simulation simply by supplying the inputs at the right simulated times and performing the underlying monadic actions, in simulated time order. We can discard the outputs, since simulations are observed by inserting probes. Thus:

```
runSim (Sim f) a as = do
  (b,r) <- f a
  runState r as

runState (Ready b s) as = runState s as
runState (Lift m) as = do s <- m
                          runState s as
runState (Wait t s k) []             -- no further inputs
  | t==infinity = return ()          -- infinity never comes
  | otherwise   = runState s []      -- timeout
runState (Wait t s k) (a:as)         -- inputs waiting
  | t <= time a = runState s (a:as)  -- timeout
  | otherwise   = runState (k a) as  -- supply event
```

Simulation states should satisfy a number of invariants, of course. Since they represent signal functions, no output events should be generated before the first input, and output events should not depend on later inputs. To ensure the latter property, we require that output events generated after an input is received carry the same or a later time. Moreover, output events must be generated in time order.

Since simulation states are in general infinite, we cannot *check* whether or not these invariants are satisfied, but we can guarantee them by construction. We therefore define "smart constructors" for the `Sim` and `State` types, which check that the values they are constructing satisfy the invariants, and raise an exception if an invariant would be broken. We ensure that simulation states are initially quiescent by constructing them with

```
sim f = Sim $ \a -> do
  (b,s) <- f a
  return (b,quiescent s)

quiescent (Lift m) = Lift (liftM quiescent m)
quiescent (Wait t s k) = wait t (quiescent s) k
```

which fails if the constructed state is ready to output before the first input, and we ensure the other invariants hold by constructing simulation states using

```
ready e r = Ready e (causal (time e) r)
lift m = Lift m
wait t f k = Wait t (causal t f)
               (\e -> causal (time e) (k e))

causal t (Ready e f) | t <= time e = Ready e f
causal t (Lift m) = Lift (liftM (causal t) m)
causal t (Wait t' s k) = Wait t' (causal t s) (causal t.k)
```

Here `causal t s` raises an exception if the *first* event output by `s` precedes time `t`; provided `s` itself satisfied the invariants then this is enough to ensure

that the states constructed using `causal` also do so. It is used in `ready` to ensure that output events occur in increasing time order, and in `wait` to ensure that output events do not precede inputs which have been received, or timeouts which have passed. The alert reader will notice that `causal` does not prevent successive output events occurring at the *same* time, and this is because such "glitches" do seem occasionally to be unavoidable (we will return to this point).

Using these smart constructors, we can now define primitive arrows with confidence that mistakes that break the invariants will not go undiscovered, at least not if they would affect the results of simulation. For example, we can define the `printA` arrow that we saw above as follows:

```
printA name = sim $ \a -> do
  message (show a++"@init")
  return (a,s)
  where s = waitInput $ \a -> lift $ do
              message (show a)
              return (ready a s)
        message a = putStrLn (name++": "++a)
```

(The `show` method for events is defined to display the value along with the time, in the format we saw earlier). Typically the simulation state is defined recursively, as we see here, in order to respond to any number of input events. The `waitInput` function is just a useful shorthand for a `wait` with an infinite timeout:

```
waitInput k = wait infinity undefined k
```

Of course, it is not necessary to define smart constructors as we have done: we could simply replace them with the real ones, and provided we make no mistakes, our simulations would behave identically. However, the smart constructors have proven to be invaluable while debugging the library. Indeed, the real code contains a more elaborate version of `causal`, which collects and reports a trace of the events leading up to a violation, which is invaluable information when a bug is discovered.

Note, however, that these smart constructors do not ensure time invariance in the arrows we write, because we write code which manipulates absolute times. It would be possible to build a further layer of abstraction on top of them, in which we would program with relative times only (except in monadic actions which do not affect future output, such as printing the simulated time in `printA`). For this prototype library, however, this degree of safety seems like overkill: we rely on writing correct code instead.

### 5.3 Implementing the arrow operations

In this section, we shall see how to implement the arrow operations for this type. Of course, we must require the underlying monad to be a monad:

```
instance Monad m => Arrow (Sim m) where
  ...
```

The `arr` operation is rather simple to define: `arr f` just applies `f` to the initial input value, and to the value in every subsequent input event. Computing a new output takes no simulated time.

```
arr f = sim $ \a -> return (f a, s)
  where s = waitInput (\a ->
              ready (Event (time a) (f (value a))) s)
```

Note, however, that the output events generated may well be redundant. For example, if we simulate an AND-gate as follows:

```
Main> runSim (arr (uncurry (&&)) >>> printA "out") (False,False)
        [Event 1 (False,True),
         Event 2 (True,False),
         Event 3 (True,True)]
out: False@init
out: False@1.0
out: False@2.0
out: True@3.0
```

then we see that the output signal carries three events, even though the value only changes once.

Redundant events are undesirable, even if they seem semantically harmless, because they cause unnecessary computation as later parts of the simulation react to the "change" of value. Our approach to avoiding the problem is to define an arrow `nubA` which represents the identity function on signals, but drops redundant events. Inserting `nubA` into the example above, we now see the output we would expect:

```
Main> runSim (arr (uncurry (&&)) >>> nubA >>> printA "out")
        (False,False)
        [Event 1 (False,True),
         Event 2 (True,False),
         Event 3 (True,True)]
out: False@init
out: True@3.0
```

Defining `nubA` just involves some simple programming with our smart constructors:

```
nubA :: (Eq a, Monad m) => Sim m a a
nubA = sim $ \a -> return (a,loop a)
  where loop a = waitInput $ \(Event t a') ->
                   if a==a' then loop a
                   else ready (Event t a') (loop a')
```

Why not just include this in the definition of `arr`? Or why not rename the old definition of `arr` to `protoArr` and redefine `arr` as follows?

```
arr f = protoArr f >>> nubA
```

The reason is *types*: we are constrained by the type stated for `arr` in the `Arrow` class. The definition above does not have that type — it has the type

```
arr :: (Arrow arr, Eq b) => (a->b) -> arr a b
```

with the extra constraint that the output type must support equality, so that `nubA` can discard events with values equal to previous one. Of course, we could just define `arr'` as above, and use that instead, but any overloaded arrow functions which we use with simulation arrows, and the code generated from the pointed arrow notation, will still use the class method `arr`. For this reason we make `nubA` explicit as an arrow, and simply expect to use it often[4].

Composing simulation arrows is easy — we just have to compute the initial output — but the real work is in composing simulation states.

```
Sim f >>> Sim g = sim $ \a -> do
  (b,sf) <- f a
  (c,sg) <- g b
  return (c,sf `stateComp` sg)
```

When we compose simulation states, we must be careful to respect our invariants. Since all the outputs of a composition are generated by a single arrow (the second operand), which should itself fulfill the invariants, we can expect them to be correctly ordered at least. However, we must see to it that no output is delayed until after a *later* input is received by the first operand, which would violate causality. To ensure this, we always produce outputs (and perform monadic actions) as early as possible. Thus, if the second operand of a composition is ready to output an event or perform a monadic computation, then so is the composition.

```
sf `stateComp` Ready c sg = ready c (sf `stateComp` sg)
sf `stateComp` Lift m = lift (liftM (sf`stateComp`) m)
```

If neither of these equations applies, then the second operand must be of the form `Wait ...`. We assume this in the equations that follow — thanks to Haskell's top-to-bottom strategy, the equations above take precedence over (apparently overlapping) equations below.

Given that the second operand is waiting, then if the first is ready to output or perform a computation, then we allow it to proceed, in the hope that it will wake

[4] It is interesting to note that the same problem arose in a different context in the first arrow paper [10]. I have proposed a language extension that would solve the problem, by allowing parameterised types to restrict their parameters to instances of certain classes only [9]. With this extension, simulation arrows could require that their output type be in class `Eq`, making equality usable on the output even though no such constraint appears in the `Arrow` class itself.

up the second operand, and enable us to generate an event from the composition without needing to wait for an input. When the first operand outputs an event, then we must of course feed it into the second. This is the purpose of the operator `` `after` ``, which computes the state of `sg` *after* it has received the event `b`. Thus `` `after` `` is the state transition function for simulation states.

```
Ready b sf `stateComp` sg = sf `stateComp` (sg `after` b)
Lift m `stateComp` sg = lift (liftM (`stateComp` sg) m)
```

When *both* operands of a composition are waiting, then the composition must also wait — but only until the earlier of the deadlines of its operands. When that simulated time is reached, the operand with the earlier deadline times out and may continue computing. If an input event is received in the meantime, it is sent (of course) to the first operand of the composition.

```
Wait tf sf kf `stateComp` Wait tg sg kg =
  wait (min tf tg)
       timeout
       (\a -> kf a `stateComp` Wait tg sg kg)
  where timeout | tf<tg  = sf `stateComp` Wait tg sg kg
                | tf==tg = sf `stateComp` sg
                | tf>tg  = Wait tf sf kf `stateComp` sg
```

Note that this code behaves correctly even if one or both deadlines is `infinity`.

To complete the definition of composition, we must implement the state transition function `after`. But this is easy: an input event is received by the first `Wait` whose deadline is after the event itself:

```
Ready b s  `after` a = ready b (s `after` a)
Lift m     `after` a = lift (liftM (`after` a) m)
Wait t s k `after` a
  | t <= time a = s `after` a
  | otherwise   = k a
```

Moving on to the `first` combinator, once again computing the initial output is easy, but adapting the simulation state much more complex.

```
simFirst (Sim f) = sim $ \(a,c) -> do
  (b,s) <- f a
  return ((b,c), stateFirst b c s)
```

When does the output of `first f` change? Well, clearly, if the output of `f` changes, then so does the output of `first f`. But the output of `first f` may also change if its *input* changes, since the change may affect the second component of the pair, which is fed directly through to the output. Moreover, when the output of `f` changes, then we need to know the current value of the second component of the input, so we can construct the new pair of output values. Likewise, when the input to `first f` changes, and we have a new second component for the output pair, then we need to know the current value of the first component of

the output to construct the new output pair. For this reason, the `stateFirst` function is parameterised not only on the state of `f`, but also on the current values of the components of the output.

In the light of this discussion, we can see that if `f` is waiting, then `first f` should also wait: no change to the output can occur until either the deadline is reached, or an input is received. If an input is received before the deadline, then `first f` is immediately ready to output a new pair containing an updated second component.

```
stateFirst b c (Wait t s k) =
  wait t (stateFirst b c s) $ \(Event t' (a,c')) ->
    ready (Event t' (b,c')) (stateFirst b c' (k (Event t' a)))
```

As before, if `f` is ready to perform a monadic action, then so is `first f`:

```
stateFirst b c (Lift m) = lift (liftM (stateFirst b c) m)
```

The trickiest case is when `f` is ready to output an event. Before we can actually output the corresponding event from `first f`, we must ensure that there are no remaining inputs at earlier times, which would cause changes to the output of `first f` that should precede the event we are about to generate. The only way to ensure this is to wait until the simulated time at which the event should occur, and see whether we timeout or receive an input. Thus we define:

```
stateFirst b c (Ready b' s) =
  wait (time b')
       (ready (Event (time b') (value b',c))
              (stateFirst (value b') c s))
       (\(Event t' (a,c')) ->
         ready (Event t' (b,c'))
                (stateFirst b c'
                    (ready b' (s `after` (Event t' a)))))
```

After waiting without seeing an input until the time of the new event, we can generate a new output immediately. If an input is received in the meantime, we generate a corresponding output event and continue waiting, but in the state we reach by feeding the first component of the input just received into the state of `f`.

This definition seems very natural, but it does exhibit a surprising behaviour which we can illustrate with the following example:

```
Sim> runSim (first (arr (+1)) >>> printA "out") (0,0)
       [Event 1 (1,1)]
out: (1,0)@init
out: (1,1)@1.0
out: (2,1)@1.0
```

Although there is only one input event, `first` generates *two* output events, one generated by the change of the input, and the other by the change in the output of `arr (+1)`. Moreover, both output events occur *at the same simulated time* — and how should we interpret that? Our solution is to declare that when several events appear at the same simulated time, then the last one to be generated gives the true value of the output signal at that time. The previous events we call "glitches", and they represent steps towards the correct value. Glitches arise, as in this example, when two parts of the simulation produce output events at exactly the same time, and they are then combined into the same signal. A glitch results because we make no attempt to identify such simultaneous events and combine them into a single one.

We put up with glitches, because of our design decision to produce output events as soon as possible. To eliminate them, we would, among other things, need to change the semantics of `Wait`. At present, `Wait` times out if no input event arrives *before* the deadline; we would need to change this to time out only if the next input event is *after* the deadline, thus allowing inputs that arrive exactly on the deadline to be received, possibly affecting the output at the same simulated time. The danger would be that some arrows would then be unable to produce their output at time `t`, without seeing a *later* input event — which would make feedback impossible to implement. However, it is not obvious that this would be inevitable, and a glitch-free version of this library would be worth investigating. For the purpose of these notes, though, we stick with the glitches.

We will, however, introduce an arrow to filter them out: an arrow which copies its input to its output, but whose output is glitch-free even when its input contains glitches. To filter out glitches in the input at time `t`, we must wait until we can be certain that all inputs at time `t` have been received — which we can only be *after* time `t` has passed! It follows that a glitch-remover *must* introduce a delay: it must wait until some time `t'` later than `t`, before it can output a copy of the input at time `t`. We therefore build glitch removal into our delay arrow, which we call `delay1`, because there is at most one event at each simulated time in its output.

In contrast to the `delay` arrow we saw earlier, in this case we do not need to supply an initial value for the output. The stream function `delay`, for use in synchronous simulation, delays its output by one step with respect to its input, and needs an initial value to insert in the output at the first simulation step. In contrast, our signals all have an initial value "since time immemorial", and the output of `delay1` just has the same initial value as its input.

We define `delay1` as follows:

```
delay1 d = sim (\a -> return (a,r))
  where r = waitInput loop
        loop (Event t a) =
          wait (t+d) (ready (Event (t+d) a) r) $
            \(Event t' a') ->
              if t==t'
                then loop (Event t' a')
```

```
                    else ready (Event (t+d) a) (loop (Event t' a'))
```

It delays its input signal by time `d`, which permits it to wait until time `t+d` to be sure that the input at time `t` is stable.

### 5.4 Implementing feedback

Just as with the other arrow combinators, implementing feedback requires two stages: first we define `loop` for simulation arrows, constructing the initial values, then we implement looping for simulation states. Now, recall that the argument of `loop` is an arrow of type `arr (a,c) (b,c)`, where `c` is the type of the loop state. Clearly the initial output of this arrow will depend on its initial input, which in turn depends on its initial output! Thus the initial output of a loop must be recursively defined. Since simulation arrows are based on an underlying monad, we need a fixpoint operator for that monad. Such a fixpoint operator is provided by the class

```
class (Monad m) => MonadFix m where
  mfix :: (a -> m a) -> m a
```

and there are instances for `IO` and many other monads.

Using this operator, we can now define `loop` for simulation arrows:

```
instance MonadFix m => ArrowLoop (Sim m) where
  loop (Sim f) = sim $ \a-> do
    ((b,c),s) <- mfix (\(~((b,c),s)) -> f (a,c))
    return (b,stateLoop a c [] s)
```

We just recursively construct the initial output from the parameter arrow `f`, and return its first component as the initial output of the loop. Obviously, the function we pass to `mfix` must not be strict, and so the lazy pattern matching (indicated by `~`) in the definition above is essential.

As simulation proceeds, the loop body `f` will periodically produce new outputs, whose second components must be fed back as changes to the input. But those input changes must be *merged* with changes to the input of the loop as a whole. We handle this by passing `stateLoop` a *queue* of pending changes to the loop state, to be merged with changes to the loop input as those arrive. We must also track the current values of the loop input and the loop state, so as to be able to construct a new input pair for the loop body when either one changes.

After this discussion, we can present the code of `stateLoop`. If the loop body is ready to output, the loop as a whole does so, and the change to the loop state is added to the pending queue.

```
stateLoop a c q (Ready (Event t (b,c')) s) =
  ready (Event t b) (stateLoop a c (q++[(t,c')]) s)
```

If the loop body is ready to perform a monadic action, then we do so.

```
stateLoop a c q (Lift m) = lift $ liftM (stateLoop a c q) m
```

If the loop body is waiting, and there are *no* pending state changes, then the loop as a whole waits. If a new input is received, then it is passed together with the current loop state to the loop body, and the current input value is updated.

```
stateLoop a c [] (Wait t s k) =
  wait t (stateLoop a c [] s) $ \(Event t' a') ->
    stateLoop a' c [] (k (Event t' (a',c)))
```

Finally, if the loop body is waiting and there *are* pending state changes, then we must wait and see whether any input event arrives before the first of them. If so, we process the input event, and if not, we make the state change.

```
stateLoop a c ((t',c'):q) (Wait t s k) =
  wait (min t t') timeout $ \(Event t'' a') ->
    stateLoop a' c ((t',c'):q) (k (Event t'' (a',c)))
  where timeout
          | t'<t = stateLoop a c' q (k (Event t' (a,c')))
          | t'>t = stateLoop a c ((t',c'):q) s
          | t'==t= stateLoop a c' q (s `after` Event t (a,c'))
```

As a simple example, let us simulate a loop which simply copies its input to its output — a loop in which the feedback is irrelevant.

```
Sim> runSim (loop (arr id) >>> printA "out") 0
         [Event 1 1, Event 2 2]
out: 0@init
out: 1@1.0
out: 1@1.0
out: 1@1.0
out: 1@1.0
out: 1@1.0
...
```

It doesn't work! The simulation produces an infinite number of glitch events, as soon as the input changes. The reason is that the first input change generates an output from the loop body including a "new" (but unchanged) loop state. This state change is fed back to the loop body, and generates another new (and also unchanged) state, and so on. To avoid this, we must *discard* state "changes" which do not actually change the state, by inserting a `nubA` arrow into the loop body.

```
Sim> runSim (loop nubA >>> printA "out")
            0
            [Event 1 1, Event 2 2]
out: 0@init
out: 1@1.0
*** Exception: <<loop>>
```

This does not work either! In this case, the exception "`<<loop>>`" is generated when a value depends on itself, and it is fairly clear which value that is — it is, of course, the loop state, whose initial value is undefined[5]. Indeed, since the initial value of the loop state cannot be determined from the input, there is no alternative to specifying it explicitly. We therefore define a new combinator, which provides the initial output value for an arrow explicitly:

```
initially x (Sim f) = Sim $ \a -> do (_,s) <- f a
                                     return (x,s)
```

Now we can revisit our example and initialise the loop state as follows:

```
Sim> runSim (loop (initially (0,0) nubA) >>> printA "out") 0
         [Event 1 1, Event 2 2]
out: 0@init
out: 1@1.0
out: 2@2.0
```

At last, it works as expected.

Now, although this example was very trivial, the difficulties that arose will be with us every time we use `loop`: the loop state must (almost) always be initialised, and we must always discard redundant state changes, to avoid generating an infinite number of events. This means that we will always need to include `nubA` in a loop body. This is not an artefact of our particular library, but a fundamental property of simulation with feedback: after an input change, a number of state changes may result, but eventually (we hope) the state will stabilise. A simulator *must* continue simulating until the state reaches a fixed point, and that is exactly what `loop` with `nubA` does.

It would be natural, now, to include `nubA` in the definition of `loop`, since it will always be required, but once again the type stated for `loop` in the class `ArrowLoop` prevents us from doing so. Instead, we define a new looping combinator just for simulations, which combines `loop` with `nubA`. We give it a type analogous to `mfix`:

```
afix :: (MonadFix m, Eq b) => Sim m (a,b) b -> Sim m a b
afix f = loop (f >>> nubA >>> arr id &&& arr id) >>> nubA
```

Because we cannot define `loop` to discard redundant state changes, we will not be able to use the `rec` syntax in the pointed arrow notation — it is simply impossible to insert `nubA` in the source code, so that it appears in the correct place in the translation. But `afix` is a good alternative: it can be used as a command combinator to good effect, as we will see in the next section.

[5] The generation of loop exceptions is somewhat variable between one GHC version and another. The output shown here was generated using version 6.2.1, but other versions might actually loop infinitely rather than raise an exception.

### 5.5 Examples

In this section we will give just a few simple examples to show how the simulation arrows can be used to simulate small circuits. First let us revisit the `nor` gate: we can now make our simulation more realistic, by including a gate delay.

```
nor = proc (a,b) -> do
        (a',b') <- delay1 0.1 -< (a,b)
        returnA -< not (a'||b')
```

A `nor` gate can be used to construct an oscillator, which generates an oscillating signal as long as its input is low:

```
oscillator = proc disable ->
  (|afix (\x -> nor -< (disable,x))|)
```

Here the output of the `nor` gate is fed back, using `afix`, to one of its inputs. While `disable` is low, the `nor` gate simply inverts its other input, and so the circuit acts as an inverter with its output coupled to its input, and oscillates. When `disable` is high, then the output of the `nor` gate is always held low. Running a simulation, we see that the oscillator behaves as expected:

```
Sim> runSim (oscillator >>> printA "out") True
         [Event 1 False, Event 2 True]
out: False@init
out: True@1.1
out: False@1.2000000000000002
out: True@1.3000000000000003
out: False@1.4000000000000004
out: True@1.5000000000000004
out: False@1.6000000000000005
out: True@1.7000000000000006
out: False@1.8000000000000007
out: True@1.9000000000000008
out: False@2.000000000000001
```

Of course, it is important to initialise the input signal to `True`, since otherwise the oscillator should oscillate "since time immemorial", and we cannot represent that. If we try, we find that the output of the oscillator is undefined.

It is interesting that in this example, we did *not* need to initialise the oscillator state. This is because the initial state is the solution to the equation

```
x = not (True || x)
```

and this is equal to `False`, because Haskell's "or" operator `(||)` is not strict in its second input, when the first one is `True`.

Finally, let us see how we can use `afix` to define a flip-flop:

```
flipflop = proc (reset,set) ->
  (|afix (\ ~(x,y)->do
     x' <- initially False nor -< (reset,y)
     y' <- initially True nor -< (set,x)
     returnA -< (x',y'))|)
```

Although this is not quite as notationally elegant as the `rec` syntax, we can see that `afix` does let us emulate recursive definitions rather nicely, and that we are able to describe the flip-flop in a natural way. As usual, the argument of a fix-point combinator cannot be strict, so we must match on the argument pair lazily. Simulating this flip-flop, we obtain the results presented in the introduction to this section.

### 5.6 Exercises

1. **Circuit simulation.** Revisit exercises 1, 2, and 4 of Section 3.5, and simulate the adder, bit serial adder, and $n$-bit counter using the new simulation arrows.
2. **Choice.** In this section, we implemented the operations in class `Arrow`, but we did not implement those in `ArrowChoice`. Can you construct an implementation of `left` for simulation arrows?
   The input signal to `left` is of type `Either a c`, which we can think of as *two* signals, one of type `a`, and one of type `c`, multiplexed onto one channel. It is the signal of type `a` that must be provided as the input to `left`'s argument, but `left f` receives this signal only incompletely — at times when the input signal is carrying a `Right` value, then the value of the input to `f` is unknown. You will need to *complete* this partially known input signal to construct an input signal for `f`, which can be done by assuming that the signal remains constant during the periods when it is unknown.
   If the initial value of the input signal is a `Right` value, then we must initialise `left f` without knowing the initial value of `f`'s input! Fortunately, we *do* know the initial value of `left f`'s *output* — it is just the same as the input in this case. We are left with the problem of initialising the arrow `f`. This cannot be done at this time, because its initial input is unknown. However, if we assume that the initial value of the `Left` input is the *same* as the first value we see, then we can initialise `f` when the first event of the form `Left a` is received.
   The output from `left f` can *also* be thought of as two signals multiplexed onto one channel, but in this case these signals are the `Right` input signal to `left f`, and the output signal from `f` itself. How should these be multiplexed? That is, when should the output signal be taken from the `Right` input, and when should it be taken from `f`? It seems natural to take the `Right` input when it is present, and the output from `f` when it is not, with the result that the multiplexing of the output channel is the same as the multiplexing of the input.

Implement `left` according to the ideas in this discussion, and experiment with the `if-then-else` pointed arrow notation (which uses it) to investigate its usefulness.

## 6 Arrows in perspective

Arrows in Haskell are directly inspired by category theory, which in turn is just the theory of arrows — a category is no more than a collection of arrows with certain properties. Thus every time we define a new instance of class `Arrow`, we construct a category! However, categorical arrows are more general than their Haskell namesakes, in two important ways. Firstly, the "source" and "target" of Haskell arrows, that is the types of their inputs and outputs, are just Haskell types. The source and target of a categorical arrow can be anything at all — natural numbers, for example, or pet poodles. In the general case, most of the operations we have considered make no sense — what would the target of `f &&& g` be, in a category where the targets of arrows are natural numbers? Secondly, Haskell arrows support many more operations than categorical arrows do. In fact, categorical arrows need only support composition and identity arrows (which we constructed as `arr id`). In general, categories have no equivalent even of our `arr` operator, let alone all the others we have considered. Thus even Haskell arrows which are only instances of class `Arrow` have much more structure than categorical arrows do in general.

However, as we saw in the introduction, there is little interesting that can be done without more operations on arrows than just composition. The same is true in category theory, and mathematicians have explored an extensive flora of additional operations that some categories have. Of particular interest to programming language semanticists are *cartesian closed* categories, which have just the right structure to model $\lambda$-calculus. In such models, the meaning of a $\lambda$-expression is an arrow of the category, from the types of its free variables (the context), to the type of its value — compare with the translations of Paterson's arrow notation. The advantage of working categorically is that one can study the properties of *all* semantic models at once, without cluttering one's work with the details of any particular one. Pierce's book is an excellent introduction to category theory from the programming language semantics perspective [21].

One may wonder, then, whether the structure provided by Haskell arrows has been studied by theoreticians? The answer turns out to be "yes". Monads, which were invented by category theorists for quite different purposes, were first connected with computational effects by Moggi [14], who used them to structure denotational semantic descriptions, and his work was the direct inspiration for Wadler to introduce monads in Haskell [26]. But Power and Robinson were dissatisfied with Moggi's approach to modelling effects, because the semantics of terms was no longer a simple arrow in a category, but rather a combination of an arrow and a monad. They asked the question: what properties should a category have for its arrows to *directly* model computations with effects? Their answer was to introduce "premonoidal" categories [22], now called Freyd categories,

which correspond closely to instances of class `Arrow`. Later, Power and Thielecke studied the command abstractions and applications that we saw in Paterson's arrow notation, under the name *closed $\kappa$-categories* [23]. Feedback operators, which are called *trace* operators by category theorists, have been studied in this setting by Benton and Hyland [1]. This latter paper moreover makes the connection back to Haskell programs, which can otherwise be somewhat obscured by notational and terminological differences.

The use of arrows in programming was introduced in my paper from the year 2000 [10]. That paper introduced the arrow classes presented here (with the exception of `ArrowLoop`), arrow transformers, and a number of applications. The direct inspiration was Swierstra and Duponcheel's non-monadic parser library [24], which collected static information about parsers to optimise them during their construction. My paper showed that their library *could* be given an arrow interface. While doing so, I also introduced two classes for arrows that can fail: `ArrowZero` and `ArrowPlus`, which provide operations for failure and failure-handling respectively. My paper also discussed stream processors, and showed that the Fudgets GUI library [3], which is based on a kind of abstract stream processor, can be given an arrow interface. Finally, I presented a small library for CGI programming.

CGI scripts are small programs that run on web servers to generate dynamic web pages. Typically, when a user fills in an HTML form in a browser, the form data is sent to a CGI script on the server which generates the next web page the user sees. CGI programming is awkward, because the script which generates a form is not usually the script that processes the user's reply. This leads to all sorts of complication, software engineering problems, and bugs.

The key idea behind my library was for a CGI script generating a form to *suspend its own state*, embed that state in a hidden field of the form (where it is sent to the client browser and returned with the other form data once the form is filled in), and then *restart* from the same state when the client's reply is received. That permits CGI programs to be written like ordinary interactive programs, where communication with the client appears as a simple function call, delivering the client's answer as a result. My implementation was based on an arrow type with a suspend operator. On suspension, the arrow combinators constructed a kind of "program counter" for the arrow, where the program counter for a composition, for example, recorded whether the suspension point was in the first or second operand, and the program counter within that operand. The program counter could then be shipped to and from the client browser, and used to restart in the same state. This idea turns out to be useful outside the world of arrows: Peter Thiemann realised that the same behaviour can be achieved elegantly using a monad, and this insight lies behind his Wash/CGI library [25].

Ross Paterson developed the pointed arrow notation presented in these notes [17], collaborated with Peyton-Jones on its implementation in GHC, and is one of the people behind the arrows web page [6]. Paterson introduced the `ArrowLoop` class, and has developed an extensive experimental arrow library containing many arrow transformers, and classes to make arrows constructed using many

transformers easy to use. Paterson applied arrows to circuit simulation, and made an arrowized version of Launchbury et als. architecture description language Hawk [12]. A good description of this work can be found in Paterson's excellent tutoral on arrows [18].

Patrik Jansson and Johan Jeuring used arrows to develop polytypic data conversion algorithms [11]. Their development is by equational reasoning, and the advantage of arrows in this setting is just the point-free notation — Jansson and Jeuring could have worked with monads instead, but their proofs would have been much clumsier.

Joe English uses arrows in his library for parsing and manipulating XML [7]. Inspired by Wallace and Runciman's HaxML [27], XML is manipulated by composing *filters*, which are almost, but not quite, functions from an `a` to a list of `b`s. Filters are defined as an arrow type, and the advantage of using arrows rather than a monad in this case is that the composition operator can be very slightly stricter, which improves memory use when the filters are run.

Courney and Elliott developed an arrow-based GUI library called Fruit [5], based on functional reactive programming. Fruit considers a GUI to be a mapping between the entire history of the user's input (mouse motion, button presses, etc), and the history of the appearance of the screen — from a user input signal to a screen output signal. The GUIs are implemented as arrows, which leads to a very attractive programming style.

Indeed, arrows have been adopted comprehensively in recent work on functional reactive programming, now using a system called Yampa [8]. Yampa programs define arrows from input signals to output signals, where a signal, just as in these notes, is a function from time to a value. Functional reactive programming is older than arrows, of course, and in its original version programmers wrote real functions from input signals to output signals. The disadvantage of doing so is that signals become real Haskell values, and are passed around in FRP programs. Since a signal represents the entire history of a value, and in principle a program might ask for the signal's value at any time, it is difficult for the garbage collector to recover any memory. In the arrowized version, in contrast, signals are not first-class values, and the arrow combinators can be implemented to make garbage collection possible. As a result, Yampa has much better memory behaviour than the original versions of FRP.

Finally, arrows have been used recently by the Clean group to develop graphical editor components [19]. Here a GUI is seen as a kind of editor for an underlying data structure — but the data structure is subject to constraints. Whenever the user interacts with the interface, thus editing the underlying data, the editor reacts by modifying other parts, and possibly performing actions on the real world, to reestablish the constraints. Editors are constructed as arrows from the underlying datatype to itself: invoking the arrow maps the data modified by the user to data in which the constraints are reestablished. This work will be presented at this very summer school.

If these applications have something in common, it is perhaps that arrows are used to combine an attractive programming style with optimisations that

would be hard or impossible to implement under a monadic interface. Arrows have certainly proved to be very useful, in applications I never suspected. I hope that these notes will help you, the reader, to use them too.

## References


1. Nick Benton and Martin Hyland. Traced premonoidal categories. *ITA*, 37(4):273–299, 2003.
2. R. S. Bird. A calculus of functions for program derivation. In D. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.
3. M. Carlsson and T. Hallgren. FUDGETS - A graphical user interface in a lazy functional language. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, Copenhagen, 1993. ACM.
4. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.
5. Anthony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, Firenze, Italy, 2001.
6. Antony Courtney, Henrik Nilsson, and Ross Paterson. Arrows: A general interface to computation. http://www.haskell.org/arrows/.
7. Joe English. Hxml. http://www.flightlab.com/∼joe/hxml/.
8. Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.
9. J. Hughes. Restricted Datatypes in Haskell. In *Third Haskell Workshop*. Utrecht University technical report, 1999.
10. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.
11. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
12. John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. In *ICFP*, pages 60–69, Paris, 1999. ACM Press.
13. Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, San Francisco, January 1995. ACM SIGPLAN-SIGACT.
14. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.
15. M. Sheeran P. Bjesse, K. Claessen and S. Singh. Lava: Hardware design in Haskell. In *ICFP*. ACM Press, 1998.
16. Ross Paterson. Arrow transformer library. http://www.haskell.org/arrows/download.html.
17. Ross Paterson. A new notation for arrows. In *ICFP*, Firenze, Italy, 2001. ACM.
18. Ross Paterson. Arrows and computation. In Jeremy Gibbons and Oege De Moor, editors, *The Fun of Programming*. Palgrave, 2003.

19. Rinus Plasmijer Peter Achten, Marko van Eekelen. Arrows for generic graphical editor components. Technical Report NIII-R0416, Nijmegen Institute for Computing and Information Sciences, University of Nijmegen, 2004.
20. Simon Peyton-Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types . http://research.microsoft.com/Users/simonpj/papers/gadt/index.htm, July 2004.
21. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
22. John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, 1997.
23. John Power and Hayo Thielecke. Closed Freyd- and $\kappa$-categories. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings 26th Int. Coll. on Automata, Languages and Programming, ICALP'99, Prague, Czech Rep., 11–15 July 1999*, volume 1644, pages 625–634. Springer-Verlag, Berlin, 1999.
24. D. S. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.
25. Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.
26. P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.
27. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, pages 148–159. ACM Press, 1999.

## Appendix: the module `Circuits.hs`

```
module Circuits where

import Control.Arrow
import List

class ArrowLoop a => ArrowCircuit a where
  delay :: b -> a b b

nor :: Arrow a => a (Bool,Bool) Bool
nor = arr (not.uncurry (||))

flipflop :: ArrowCircuit a => a (Bool,Bool) (Bool,Bool)
flipflop = loop (arr (\((a,b),~(c,d)) -> ((a,d),(b,c))) >>>
 nor *** nor >>>
 delay (False,True) >>>
 arr id &&& arr id)

class Signal a where
  showSignal :: [a] -> String

instance Signal Bool where
  showSignal bs = concat top++"\n"++concat bot++"\n"
    where (top,bot) = unzip (zipWith sh (False:bs) bs)
          sh True True = ("__","  ")
  sh True False = ("  ","|_")
  sh False True = (" _","| ")
  sh False False = ("  ","__")

instance (Signal a,Signal b) => Signal (a,b) where
  showSignal xys = showSignal (map fst xys) ++
                   showSignal (map snd xys)

instance Signal a => Signal [a] where
  showSignal = concat . map showSignal . transpose

sig = concat . map (uncurry replicate)

flipflopInput = sig
[(5,(False,False)),(2,(False,True)),(5,(False,False)),
 (2,(True,False)),(5,(False,False)),(2,(True,True)),
 (6,(False,False))]
```