Arrows and computation

Ross Paterson

Many programs and libraries involve components that are 'function-like', in that they take inputs and produce outputs, but are not simple functions from inputs to outputs. This chapter explores the features of such 'notions of computation', defining a common interface, called 'arrows'. This allows all these notions of computation to share infrastructure, such as libraries, proofs or language support. Arrows also provide a useful discipline for structuring many programs, and allow one to program at a greater level of generality. Monads, discussed in Chapter 11 of IFPH, serve a similar purpose, but arrows are more general. In particular, they include notions of computation with static components, independent of the input, as well as computations that consume multiple inputs. We also consider some useful subclasses of arrows, one of which turns out to be equivalent to monads. The others bring choice and feedback to the arrow world.

With this machinery, we can give a common structure to programs based on different notions of computation. The generality of arrows tends to force one into a point-free style, which is useful for proving general properties. However it is not to everyone's taste, and can be awkward for programming specific instances. The solution is a point-wise notation for arrows, which is automatically translated to the functional language Haskell. Each notion of computation thus defines a special sublanguage of Haskell.

1 Notions of computation

We shall explore what we mean by a notion of computation using four varied examples. As a point of comparison, we shall consider how the following operator on functions may be generalized to the various types of 'function-like' components.

$$
\begin{aligned}
&add && :: (\beta \to Int) \to (\beta \to Int) \to (\beta \to Int) \\
&add\, f\ g\ b && = f\ b + g\ b
\end{aligned}
$$

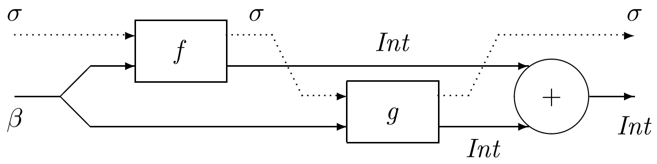
State transformers: Applications that involve the threading of a state through a function might have types described by

$$\textbf{type}\ \mathit{State}\ \sigma\ \iota\ o = (\sigma, \iota) \rightarrow (\sigma, o)$$

A generalization of *add* to state transformers is

$$\begin{aligned}
&\mathit{addST} && :: \mathit{State}\ \sigma\ \beta\ \mathit{Int} \rightarrow \mathit{State}\ \sigma\ \beta\ \mathit{Int} \rightarrow \mathit{State}\ \sigma\ \beta\ \mathit{Int} \\
&\mathit{addST}\ f\ g\ (s, b) && = \textbf{let}\ (s', x)\ = f\ (s, b) \\
& && \qquad\ \ (s'', y) = g\ (s', b) \\
& && \quad\ \textbf{in}\ \ (s'', x + y)
\end{aligned}$$

which we can picture as below:



But for the dotted line, this is the same as *add*. The difference is that the state σ is threaded first through f and then through g to produce the resulting state. This is not the only choice: we could have defined *addST* to thread the state through g and then through f.

Nondeterminism: Many search algorithms may be described as functions that return a list of possible answers (with the empty list indicating failure):

$$\textbf{type}\ \mathit{NonDet}\ \iota\ o = \iota \rightarrow [o]$$

The generalization of *add* is to add each combination:

$$\begin{aligned}
&\mathit{addND} && :: \mathit{NonDet}\ \beta\ \mathit{Int} \rightarrow \mathit{NonDet}\ \beta\ \mathit{Int} \rightarrow \mathit{NonDet}\ \beta\ \mathit{Int} \\
&\mathit{addND}\ f\ g\ b && = [x + y \mid x \leftarrow f\ b,\ y \leftarrow g\ b]
\end{aligned}$$

The effect under lazy evaluation is similar to backtracking. To compute the head of the list, the function gets the first elements from f and g. To compute the next element, it gets the next element from g if any, or otherwise the next element from f and the first from g, and so on. As with state transformers, another version is available: we could have gone through all the results of f first.

More sophisticated variations on this idea are discussed in Chapter **??**.

Map transformers: A data parallel algorithm transforms a family of input values, one for each place in a set σ, into a family of output values at the same places:

$$\textbf{type}\ \mathit{MapTrans}\ \sigma\ \iota\ o = (\sigma \rightarrow \iota) \rightarrow (\sigma \rightarrow o)$$

Another interpretation takes σ as representing time, so that a function $\sigma \rightarrow \alpha$ is a time-varying value, or *behaviour*, and the component is a behaviour transformer.

The generalization of *add* to map transformers adds the functions pointwise:

$$\begin{aligned}
&\mathit{addMT} :: \mathit{MapTrans}\ \sigma\ \beta\ \mathit{Int} \rightarrow \mathit{MapTrans}\ \sigma\ \beta\ \mathit{Int} \rightarrow \mathit{MapTrans}\ \sigma\ \beta\ \mathit{Int} \\
&\mathit{addMT}\ f\ g\ m\ s\ = f\ m\ s\ +\ g\ m\ s
\end{aligned}$$

If σ represents place, this corresponds to evaluating f and g at each place and summing each pair of results. If σ represents time, this operator adds the values of two behaviours at each point in time.

Simple automata: One way to model synchronous circuits is to use simple automata, which map an input to an output and a new version of themselves:

$$\mathbf{newtype}\ \mathit{Auto}\ \iota\ o = A\ (\iota\ \rightarrow\ (o,\ \mathit{Auto}\ \iota\ o))$$

The generalization of *add* to automata runs the two automata in parallel, summing each pair of results:

$$\begin{array}{l}
\mathit{addAuto} :: \mathit{Auto}\ \beta\ \mathit{Int}\ \rightarrow\ \mathit{Auto}\ \beta\ \mathit{Int}\ \rightarrow\ \mathit{Auto}\ \beta\ \mathit{Int} \\
\mathit{addAuto}\ (A\ f)\ (A\ g) = A\ (\lambda\ b\ \rightarrow\ \mathbf{let}\ (x, f') = f\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (y, g') = g\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ (x + y,\ \mathit{addAuto}\ f'\ g'))
\end{array}$$

1.1 Categories and functors

In each of the above examples, and many more like them, there is a type $A \rightsquigarrow B$ of *computations* taking inputs of type A and producing outputs of type B. It is natural to ask what else these examples have in common. We shall define an interface that is sufficiently general to encompass our examples (and many others), but also powerful enough to program general combinators like *add*.

It seems reasonable to expect that we can compose two computations by connecting the output of the first to the input of the second, and that composition is associative with an identity (a 'do-nothing' computation). This is the standard notion of a *category*, with types as objects and computations as arrows (or morphisms). We can represent this in Haskell with the class

$$\begin{array}{l}
\mathbf{class}\ \mathit{Category}\ \alpha\ \mathbf{where} \\
\quad \mathit{idA} \ \ :: \alpha\ \beta\ \beta \\
\quad (\ggg) :: \alpha\ \beta\ \gamma\ \rightarrow\ \alpha\ \gamma\ \delta\ \rightarrow\ \alpha\ \beta\ \delta
\end{array}$$

Haskell does not permit infix type constructors like $\rightsquigarrow$, so the arrow type constructor α is placed before its two type arguments.

In a category, these operations are required to satisfy the axioms

identity	$\mathit{idA} \ggg f$	$=$	$f \ggg \mathit{idA}$	$=$	f
associativity	$(f \ggg g) \ggg h$	$=$	$f \ggg (g \ggg h)$		

Pure functions constitute a category:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Category}\ (\rightarrow)\ \mathbf{where} \\
\quad \mathit{idA} \qquad = \mathit{id} \\
\quad f \ggg g = g \cdot f
\end{array}$$

A category is a very general concept, and it is perhaps no surprise that all of our examples induce categories, as do many other things. However this interface is much too general (or offers too little) for programming purposes. Firstly, we shall also require a combinator to embed ordinary functions as 'pure' computations:

$$pure :: (\beta \rightarrow \gamma) \rightarrow \alpha\,\beta\,\gamma$$

satisfying the following axioms:

functor-identity	$pure\ id$	$=$	idA
functor-composition	$pure\ (g \cdot f)$	$=$	$pure\ f \ggg pure\ g$

Normally functional programmers use only functors, like *map*, from the category of functions to itself, but *pure* is a functor from the category of pure functions to the new category.

Using *pure*, we can lift functions like $+$ to computations, but this is still not enough to implement a generalization of *add*. So far, the results of a computation must be fed to the immediately following computation; we have no way to save intermediate results. We need a means to apply a computation to part of the input, while preserving the rest. In the category of functions, this is done using the product functor:[1]

$$\begin{array}{ll} (\times) & :: (\alpha \rightarrow \alpha') \rightarrow (\beta \rightarrow \beta') \rightarrow (\alpha, \beta) \rightarrow (\alpha', \beta') \\ (f \times g)\tilde{}(a, b) = (f\ a, g\ b) & \end{array}$$

The operator $\times$ is a functor in two arguments, so that we have

$$(f \cdot g) \times (h \cdot k) \quad = \quad (f \times h) \cdot (g \times k)$$

This does not generalize directly to other notions of computation. For example, in state transformer computations the order of g and h cannot be changed without changing the effect of the computation. However, it suffices to assume a one-sided product, which we shall call *first*. Hence instead of *Category*, we use the following class:

$$\begin{array}{l} \textbf{class}\ Arrow\ \alpha\ \textbf{where} \\ \quad pure\ \ :: (\beta \rightarrow \gamma) \rightarrow \alpha\,\beta\,\gamma \\ \quad (\ggg) :: \alpha\,\beta\,\gamma \rightarrow \alpha\,\gamma\,\delta \rightarrow \alpha\,\beta\,\delta \\ \quad first\ \ :: \alpha\,\beta\,\gamma \rightarrow \alpha\,(\beta, \delta)\,(\gamma, \delta) \end{array}$$

There is no need to assume *idA*, because it is uniquely defined by the functor-identity law for *pure*:

$$\begin{array}{l} idA :: Arrow\ \alpha \Rightarrow \alpha\,\beta\,\beta \\ idA = pure\ id \end{array}$$

We need not assume a mirror image of *first*, as it may be defined as

[1] The tilde marks an irrefutable pattern in Haskell. It is used in this and some later definitions to make the matching of pairs non-strict.

$$
\begin{array}{l}
\mathit{second} \quad :: \mathit{Arrow}\ \alpha \Rightarrow \alpha\ \beta\ \gamma \rightarrow \alpha\ (\delta,\beta)\ (\delta,\gamma) \\
\mathit{second}\ f = \mathit{pure}\ \mathit{swap} \ggg \mathit{first}\ f \ggg \mathit{pure}\ \mathit{swap} \\
\qquad\qquad \textbf{where}\ \ \mathit{swap}\ \tilde{}(x,y) = (y,x)
\end{array}
$$

We assume that *first* satisfies the following axioms:

$$
\begin{array}{lrcl}
\textsf{\textbf{extension}} & \mathit{first}\ (\mathit{pure}\ f) & = & \mathit{pure}\ (f \times id) \\
\textsf{\textbf{functor}} & \mathit{first}\ (f \ggg g) & = & \mathit{first}\ f \ggg \mathit{first}\ g \\
\textsf{\textbf{exchange}} & \mathit{first}\ f \ggg \mathit{pure}\ (id \times g) & = & \mathit{pure}\ (id \times g) \ggg \mathit{first}\ f \\
\textsf{\textbf{unit}} & \mathit{first}\ f \ggg \mathit{pure}\ \mathit{fst} & = & \mathit{pure}\ \mathit{fst} \ggg f \\
\textsf{\textbf{association}} & \mathit{first}\ (\mathit{first}\ f) \ggg \mathit{pure}\ \mathit{assoc} & = & \mathit{pure}\ \mathit{assoc} \ggg \mathit{first}\ f
\end{array}
$$

where *assoc* rearranges pairs:

$$
\begin{array}{l}
\mathit{assoc} \qquad\qquad :: ((\alpha,\beta),\gamma) \rightarrow (\alpha,(\beta,\gamma)) \\
\mathit{assoc}\ \tilde{}(\tilde{}(a,b),c) = (a,(b,c))
\end{array}
$$

Ordinary functions are an instance of the *Arrow* class:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ (\rightarrow)\ \textbf{where} \\
\quad \mathit{pure}\ f \quad = f \\
\quad f \ggg g = g \cdot f \\
\quad \mathit{first}\ f \quad = f \times id
\end{array}
$$

Our other examples are also instances, though in some cases we must alter the definitions slightly, because Haskell does not permit type synonyms as instances:

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{State}\ \sigma\ \iota\ o \qquad = \mathit{ST}\ ((\sigma,\iota) \rightarrow (\sigma,o)) \\
\textbf{newtype}\ \mathit{NonDet}\ \iota\ o \qquad = \mathit{ND}\ (\iota \rightarrow [o]) \\
\textbf{newtype}\ \mathit{MapTrans}\ \sigma\ \iota\ o = \mathit{MT}\ ((\sigma \rightarrow \iota) \rightarrow (\sigma \rightarrow o))
\end{array}
$$

Now we can define an instance for the *State* type. A pure state transformer leaves the state untouched, while *first* routes the state through f:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ (\mathit{State}\ \sigma)\ \textbf{where} \\
\quad \mathit{pure}\ f \qquad\quad = \mathit{ST}\ (id \times f) \\
\quad \mathit{ST}\ f \ggg \mathit{ST}\ g = \mathit{ST}\ (g \cdot f) \\
\quad \mathit{first}\ (\mathit{ST}\ f) \quad = \mathit{ST}\ (\mathit{assoc} \cdot (f \times id) \cdot \mathit{unassoc})
\end{array}
$$

where *unassoc* is the inverse of *assoc*:

$$
\begin{array}{l}
\mathit{unassoc} \qquad\qquad :: (\alpha,(\beta,\gamma)) \rightarrow ((\alpha,\beta),\gamma) \\
\mathit{unassoc}\ \tilde{}(a,\tilde{}(b,c)) = ((a,b),c)
\end{array}
$$

A pure non-deterministic computation is a single-valued, or deterministic, computation, while composition encapsulates backtracking:

$$
\textbf{instance}\ \mathit{Arrow}\ \mathit{NonDet}\ \textbf{where}
$$

$$
\begin{aligned}
&\mathit{pure}\, f &&= \mathit{ND}\, (\lambda\, b \rightarrow [f\ b])\\
&\mathit{ND}\, f \ggg \mathit{ND}\, g &&= \mathit{ND}\, (\lambda\, b \rightarrow [d \mid c \leftarrow f\ b,\ d \leftarrow g\ c])\\
&\mathit{first}\, (\mathit{ND}\, f) &&= \mathit{ND}\, (\lambda\, (b, d) \rightarrow [(c, d) \mid c \leftarrow f\ b])
\end{aligned}
$$

A pure map transformer applies a function to each result of the map, while *first* applies a transformer to only part of a map:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ (\mathit{MapTrans}\ \sigma)\ \textbf{where}\\
&\quad \mathit{pure}\, f &&= \mathit{MT}\, (f\, \cdot)\\
&\quad \mathit{MT}\, f \ggg \mathit{MT}\, g &&= \mathit{MT}\, (g \cdot f)\\
&\quad \mathit{first}\, (\mathit{MT}\, f) &&= \mathit{MT}\, (\mathit{zipMap} \cdot (f \times \mathit{id}) \cdot \mathit{unzipMap})
\end{aligned}
$$

where *zipMap* and its inverse *unzipMap* convert between pairs of maps and maps yielding pairs:

$$
\begin{aligned}
&\mathit{zipMap} &&:: (\sigma \rightarrow \alpha, \sigma \rightarrow \beta) \rightarrow (\sigma \rightarrow (\alpha, \beta))\\
&\mathit{zipMap}\ h\ s &&= (\mathit{fst}\ h\ s, \mathit{snd}\ h\ s)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unzipMap} &&:: (\sigma \rightarrow (\alpha, \beta)) \rightarrow (\sigma \rightarrow \alpha, \sigma \rightarrow \beta)\\
&\mathit{unzipMap}\ h &&= (\mathit{fst} \cdot h, \mathit{snd} \cdot h)
\end{aligned}
$$

The instance for automata is a bit more complicated:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathit{Auto}\ \textbf{where}\\
&\quad \mathit{pure}\, f &&= A\, (\lambda\, b \rightarrow (f\ b, \mathit{pure}\, f))\\
&\quad A\, f \ggg A\, g &&= A\, (\lambda\, b \rightarrow \textbf{let}\ (c, f') = f\ b\\
&&&\qquad\qquad\qquad\quad\ \ (d, g') = g\ c\\
&&&\qquad\qquad\qquad\ \textbf{in}\ \ (d, f' \ggg g'))\\
&\quad \mathit{first}\, (A\, f) &&= A\, (\lambda\, (b, d) \rightarrow \textbf{let}\ (c, f') = f\ b\\
&&&\qquad\qquad\qquad\qquad\ \textbf{in}\ \ ((c, d), \mathit{first}\, f'))
\end{aligned}
$$

Remember that an automaton maps an input to an output and a new version of itself to be used on the next input. In the case of a pure automaton, the new version is the same as the old one: the automaton is stateless, with each output a function of the corresponding input. A pair of automata are composed by running them side by side, with each output of the first used as an input to the second.

Now we can write some general combinators for arrows. For example, we can define something that looks like a product:

$$
\begin{aligned}
&(\ast\!\!\ast\!\!\ast) &&:: \mathit{Arrow}\ \alpha \Rightarrow \alpha\, \beta\, \gamma \rightarrow \alpha\, \beta'\, \gamma' \rightarrow \alpha\, (\beta, \beta')\, (\gamma, \gamma')\\
&f \ast\!\!\ast\!\!\ast g &&= \mathit{first}\, f \ggg \mathit{second}\ g
\end{aligned}
$$

Note that we have arbitrarily chosen an order for the computations f and g. For many arrows (*e.g.* state transformers), the other order has a different meaning. As a consequence, $\ast\!\!\ast\!\!\ast$ is not in general a functor.

A convenient variant duplicates the input first:

$$
\begin{aligned}
&(\&\&\&) &&:: \mathit{Arrow}\ \alpha \Rightarrow \alpha\, \beta\, \gamma \rightarrow \alpha\, \beta\, \gamma' \rightarrow \alpha\, \beta\, (\gamma, \gamma')\\
&f\ \&\&\&\ g &&= \mathit{pure}\ \mathit{dup} \ggg (f \ast\!\!\ast\!\!\ast g)
\end{aligned}
$$

$$\textbf{where}\ \ dup\ b = (b, b)$$

A general combinator corresponding to *add* is:

$$\begin{aligned} &addA && :: Arrow\ \alpha \Rightarrow \alpha\,\beta\,Int \rightarrow \alpha\,\beta\,Int \rightarrow \alpha\,\beta\,Int \\ &addA\,f\,g = f\ \&\&\&\ g \ggg pure\,(uncurry\,(+)) \end{aligned}$$

Each of the versions of *add* defined above may be obtained by specializing *addA* with the appropriate arrow type.

Exercise 1 Write *Arrow* instances for the following types:

$$\begin{aligned} &\textbf{newtype}\ Reader\ \sigma\ \iota\ o = R\ ((\sigma, \iota) \rightarrow o) \\ &\textbf{newtype}\ Writer\ \iota\ o \quad = W\ (\iota \rightarrow (String, o)) \end{aligned}$$

In the latter case, any monoid could be used in place of *String*. □

Exercise 2 The following is almost an arrow type:

$$\textbf{newtype}\ ListMap\ \iota\ o = LM\ ([\iota] \rightarrow [o])$$

What goes wrong? □

Exercise 3 Define the following as an arrow type:

$$\begin{aligned} &\textbf{data}\ Stream\ \alpha && = Cons\ \alpha\ (Stream\ \alpha) \\ &\textbf{newtype}\ StreamMap\ \iota\ o && = SM\ (Stream\ \iota \rightarrow Stream\ o) \end{aligned}$$

This enables us to mimic dataflow languages, in which an infinite list represents all the values of a variable, or Kahn networks, in which a list represents all the values that pass through a channel. □

Exercise 4 Show that the following is a functor:

$$\begin{aligned} &(\ltimes) && :: Arrow\ \alpha \Rightarrow \alpha\,\beta\,\gamma \rightarrow (\beta' \rightarrow \gamma') \rightarrow \alpha\,(\beta, \beta')\,(\gamma, \gamma') \\ &f \ltimes g = \mathit{first}\,f \ggg (id \times g) \end{aligned}$$

□

2 Special cases

We consider here a number of interfaces that are less general than arrows, but which nevertheless offer useful additional facilities when available.

2.1 Arrows and monads

The *Arrow* combinators operate on computations, rather than the values that pass through them. They are *point-free*, in contrast to normal functional programming, where we use named values (variables) bound by λ or **let**. We could name inputs to arrows in the same way if the arrow type α satisfied

currying $\quad \alpha\,(\beta,\gamma)\,\delta \quad \cong \quad \beta \rightarrow \alpha\,\gamma\,\delta$

(This is an example of what category theorists call an *adjunction*.) There is a obvious function from the left side to the right:

$$\begin{aligned} &\mathit{curryA} && :: \mathit{Arrow}\,\alpha \Rightarrow \alpha\,(\beta,\gamma)\,\delta \rightarrow \beta \rightarrow \alpha\,\gamma\,\delta \\ &\mathit{curryA}\,f\;b && = \mathit{mkPair}\;b \ggg f \end{aligned}$$

$$\begin{aligned} &\mathit{mkPair} && :: \mathit{Arrow}\,\alpha \Rightarrow \beta \rightarrow \alpha\,\gamma\,(\beta,\gamma) \\ &\mathit{mkPair}\;b && = \mathit{pure}\,(\lambda\,c \rightarrow (b,c)) \end{aligned}$$

But does *curryA* have an inverse? Such an inverse would map $\mathit{id} :: \alpha\,\gamma\,\delta \rightarrow \alpha\,\gamma\,\delta$ to a combinator

$$\begin{aligned} &\textbf{class}\;\mathit{Arrow}\,\alpha \Rightarrow \mathit{ArrowApply}\,\alpha\;\textbf{where} \\ &\quad \mathit{app} :: \alpha\,(\alpha\,\gamma\,\delta,\gamma)\,\delta \end{aligned}$$

Indeed, it can be shown that a natural currying isomorphism exists if and only if such an arrow *app* exists and satisfies the axioms:

composition	$\mathit{pure}\,((\ggg h) \times \mathit{id}) \ggg \mathit{app}$	$=$	$\mathit{app} \ggg h$
reduction	$\mathit{pure}\,(\mathit{mkPair} \times \mathit{id}) \ggg \mathit{app}$	$=$	$\mathit{pure}\;\mathit{id}$
extensionality	$\mathit{mkPair}\,f \ggg \mathit{app}$	$=$	f

Pure functions are a trivial instance of this class:

$$\begin{aligned} &\textbf{instance}\;\mathit{ArrowApply}\,(\rightarrow)\;\textbf{where} \\ &\quad \mathit{app}\tilde{}(f,c) = f\;c \end{aligned}$$

Of the other arrows we have considered, *State* and *NonDet* are instances, but *MapTrans* and *Auto* are not. Indeed, the currying isomorphism implies

$$\alpha\,\beta\,\delta \quad \cong \quad \alpha\,(\beta,())\,\delta \quad \cong \quad \beta \rightarrow \alpha\,()\,\delta$$

and any such arrow may be factored as $\beta \rightarrow M\;\delta$, where M is a monad. Conversely any monad gives rise to an arrow of this form, called its Kleisli arrow, which satisfies the currying isomorphism. Thus the *ArrowApply* class is merely a less convenient version of the *Monad* class. It describes computations that always take a single input. However, arrows also include computations that consume multiple input values (like *MapTrans* and *Auto*), as well as computations that are partially static, *i.e.* independent of the input. We shall see examples of both kinds in Section 4.

Exercise 5 Verify the *ArrowApply* axioms for pure functions. □

Exercise 6 The following instance has the correct type:

$$\begin{array}{l}\textbf{instance } \mathit{ArrowApply\ Auto} \textbf{ where}\\ \quad \mathit{app} = \mathit{pure}\ (\lambda\ (A\ f, x) \to \mathit{fst}\ (f\ x))\end{array}$$

Show that the extensionality axiom fails for this definition. □

2.2 Conditionals

In many situations we wish to perform different computations for different inputs. This would be trivial if we could refer to the input directly, as with monadic computations (or, equivalently, *ArrowApply*). Although this is not available for arrows in general, more arrows admit conditional computation than currying. We begin with a Haskell sum type:

$$\textbf{data } \mathit{Either}\ \alpha\ \beta = \mathit{Left}\ \alpha \mid \mathit{Right}\ \beta$$

Just as with the product type, there is a corresponding functor:

$$\begin{array}{ll}(\oplus) & :: (\alpha \to \alpha') \to (\beta \to \beta') \to \mathit{Either}\ \alpha\ \beta \to \mathit{Either}\ \alpha'\ \beta'\\ (f \oplus g)\ (\mathit{Left}\ a) & = \mathit{Left}\ (f\ a)\\ (f \oplus g)\ (\mathit{Right}\ b) & = \mathit{Right}\ (g\ b)\end{array}$$

By analogy with products, we postulate a one-sided sum functor on arrows:

$$\begin{array}{l}\textbf{class } \mathit{Arrow}\ \alpha \Rightarrow \mathit{ArrowChoice}\ \alpha \textbf{ where}\\ \quad \mathit{left} :: \alpha\ \beta\ \gamma \to \alpha\ (\mathit{Either}\ \beta\ \delta)\ (\mathit{Either}\ \gamma\ \delta)\end{array}$$

As with products, the mirror image may be defined:

$$\begin{array}{l}\mathit{right} \quad :: \mathit{ArrowChoice}\ \alpha \Rightarrow \alpha\ \beta\ \gamma \to \alpha\ (\mathit{Either}\ \delta\ \beta)\ (\mathit{Either}\ \delta\ \gamma)\\ \mathit{right}\ f = \mathit{pure\ mirror} \ggg \mathit{left}\ f \ggg \mathit{pure\ mirror}\\ \qquad\quad \textbf{where } \begin{array}{ll}\mathit{mirror}\ (\mathit{Left}\ x) & = \mathit{Right}\ x\\ \mathit{mirror}\ (\mathit{Right}\ y) & = \mathit{Left}\ y\end{array}\end{array}$$

The axioms for *left* mirror those for *first*:

extension	$\mathit{left}\ (\mathit{pure}\ f)$	$=$	$\mathit{pure}\ (f \oplus \mathit{id})$
functor	$\mathit{left}\ (f \ggg g)$	$=$	$\mathit{left}\ f \ggg \mathit{left}\ g$
exchange	$\mathit{left}\ f \ggg \mathit{pure}\ (\mathit{id} \oplus g)$	$=$	$\mathit{pure}\ (\mathit{id} \oplus g) \ggg \mathit{left}\ f$
unit	$\mathit{pure\ Left} \ggg \mathit{left}\ f$	$=$	$f \ggg \mathit{pure\ Left}$
association	$\mathit{left}\ (\mathit{left}\ f) \ggg \mathit{pure\ assocsum}$	$=$	$\mathit{pure\ assocsum} \ggg \mathit{left}\ f$

for the function

$$\begin{array}{ll}\mathit{assocsum} & :: \mathit{Either}\ (\mathit{Either}\ \alpha\ \beta)\ \gamma \to \mathit{Either}\ \alpha\ (\mathit{Either}\ \beta\ \gamma)\\ \mathit{assocsum}\ (\mathit{Left}\ (\mathit{Left}\ a)) & = \mathit{Left}\ a\\ \mathit{assocsum}\ (\mathit{Left}\ (\mathit{Right}\ b)) & = \mathit{Right}\ (\mathit{Left}\ b)\\ \mathit{assocsum}\ (\mathit{Right}\ c) & = \mathit{Right}\ (\mathit{Right}\ c)\end{array}$$

In the world of pure functions, sums and products are related by the function

$$
\begin{aligned}
&\mathit{distr} && :: (\mathit{Either}\ \alpha\ \beta, \gamma) \rightarrow \mathit{Either}\ (\alpha, \gamma)\ (\beta, \gamma) \\
&\mathit{distr}\ (\mathit{Left}\ a, c) && = \mathit{Left}\ (a, c) \\
&\mathit{distr}\ (\mathit{Right}\ b, c) && = \mathit{Right}\ (b, c)
\end{aligned}
$$

We also postulate an additional axiom for arrows:

distribution $\mathit{first}\ (\mathit{left}\ f) \ggg \mathit{pure}\ \mathit{distr} \quad = \quad \mathit{pure}\ \mathit{distr} \ggg \mathit{left}\ (\mathit{first}\ f)$

We can define derived combinators corresponding to those for products:

$$
\begin{aligned}
&(\lhd\!\!\!\rhd) && :: \mathit{ArrowChoice}\ \alpha \Rightarrow \\
& && \quad \alpha\ \beta\ \gamma \rightarrow \alpha\ \beta'\ \gamma' \rightarrow \alpha\ (\mathit{Either}\ \beta\ \beta')\ (\mathit{Either}\ \gamma\ \gamma') \\
&f \lhd\!\!\!\rhd g && = \mathit{left}\ f \ggg \mathit{right}\ g
\end{aligned}
$$

$$
\begin{aligned}
&(|||) && :: \mathit{ArrowChoice}\ \alpha \Rightarrow \alpha\ \beta\ \delta \rightarrow \alpha\ \gamma\ \delta \rightarrow \alpha\ (\mathit{Either}\ \beta\ \gamma)\ \delta \\
&f\ |||\ g && = f \lhd\!\!\!\rhd g \ggg \mathit{pure}\ \mathit{untag} \\
& && \quad \textbf{where}\ \ \mathit{untag}\ (\mathit{Left}\ x) \ = x \\
& && \qquad\qquad\ \ \mathit{untag}\ (\mathit{Right}\ y) = y
\end{aligned}
$$

Naturally pure functions are an instance of *ArrowChoice*:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{ArrowChoice}\ (\rightarrow)\ \textbf{where} \\
&\quad \mathit{left}\ f = f \oplus \mathit{id}
\end{aligned}
$$

Indeed any instance of *ArrowApply*, including *State* and *NonDet*, is also an instance of *ArrowChoice*. More interestingly, simple automata allow choice, even though they do not permit application:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{ArrowChoice}\ \mathit{Auto}\ \textbf{where} \\
&\quad \mathit{left}\ (A\ f) = A\ \mathit{lf} \\
&\qquad\qquad \textbf{where}\ \ \mathit{lf}\ (\mathit{Left}\ b) \ \ = \textbf{let}\ (c, f') = f\ b \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ (\mathit{Left}\ c, \mathit{left}\ f') \\
&\qquad\qquad\qquad\quad \mathit{lf}\ (\mathit{Right}\ d) = (\mathit{Right}\ d, \mathit{left}\ (A\ f))
\end{aligned}
$$

Only inputs marked *Left* are run through the automaton, while others are copied to the output.

There is no *ArrowChoice* instance for map transformers.

Exercise 7 Define *ArrowChoice* instances for *NonDet*, *State* and the *StreamMap* type from Exercise 3. □

Exercise 8 Show that the equation

$$
(f\ |||\ g) \ggg h \quad = \quad (f \ggg h)\ |||\ (g \ggg h)
$$

fails for the *Auto* and *StreamMap* arrows. □

Exercise 9 Given the following definition, which adds string-valued exceptions to an arrow:

$$\textbf{newtype}\ \mathit{Except}\ \alpha\ \beta\ \gamma = E\ (\alpha\ \beta\ (\mathit{Either}\ \mathit{String}\ \gamma))$$

Define the following instance

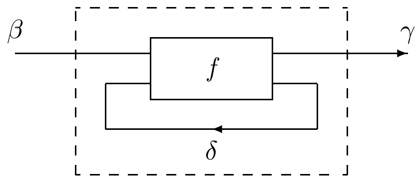
$$\textbf{instance}\ \mathit{ArrowChoice}\ \alpha \Rightarrow \mathit{Arrow}\ (\mathit{Except}\ \alpha)$$

□

Exercise 10 Prove the functor axiom for *first* in the arrow defined in the previous exercise. This requires the distributivity axiom. □

2.3 Feedback

Since arrows are Haskell values, they may be recursively defined in the usual way. Sometimes we want a different form of recursion, where values input to an arrow are defined in terms of its outputs, as in the following diagram:



For ordinary functions, we can define a function category theorists call a *trace*:

$$\begin{aligned} &\mathit{trace} :: ((\beta,\delta) \to (\gamma,\delta)) \to \beta \to \gamma \\ &\mathit{trace}\ f\ b = \textbf{let}\ (c,d) = f\ (b,d)\ \textbf{in}\ c \end{aligned}$$

Here the second component of the output of f is fed back as the second component of its input. Some arrows permit a generalization of *trace*, characterized by the class

$$\begin{aligned} &\textbf{class}\ \mathit{Arrow}\ \alpha \Rightarrow \mathit{ArrowLoop}\ \alpha\ \textbf{where} \\ &\quad \mathit{loop} :: \alpha\ (\beta,\delta)\ (\gamma,\delta) \to \alpha\ \beta\ \gamma \end{aligned}$$

The intention is that while the value is fed back, any effect of the computation occurs once only, and this is reflected by the axioms of *loop*:

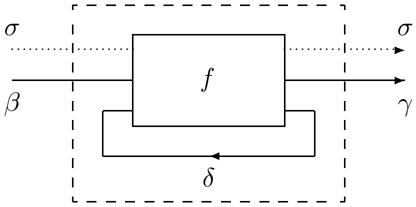
extension	$\mathit{loop}\ (\mathit{pure}\ f)$	$=$	$\mathit{pure}\ (\mathit{trace}\ f)$
left tightening	$\mathit{loop}\ (\mathit{first}\ h \ggg f)$	$=$	$h \ggg \mathit{loop}\ f$
right tightening	$\mathit{loop}\ (f \ggg \mathit{first}\ h)$	$=$	$\mathit{loop}\ f \ggg h$
sliding	$\mathit{loop}\ (f \ggg \mathit{pure}\ (\mathit{id} \times k))$	$=$	$\mathit{loop}\ (\mathit{pure}\ (\mathit{id} \times k) \ggg f)$
vanishing	$\mathit{loop}\ (\mathit{loop}\ f)$	$=$	$\mathit{loop}\ (\mathit{pure}\ \mathit{unassoc} \ggg f \ggg \mathit{pure}\ \mathit{assoc})$
superposing	$\mathit{second}\ (\mathit{loop}\ f)$	$=$	$\mathit{loop}\ (\mathit{pure}\ \mathit{assoc} \ggg \mathit{second}\ f \ggg \mathit{pure}\ \mathit{unassoc})$

For example, a computation that is independent of the feedback value may be moved out of the *loop*, as in the 'tightening' axioms.

As we have already seen, ordinary functions support such an operator:

$$\begin{array}{l}\textbf{instance}\ \mathit{ArrowLoop}\ (\to)\ \textbf{where}\\ \quad \mathit{loop} = \mathit{trace}\end{array}$$

In the case of state transformers, we want a function that transforms the state once, while feeding back part of the output:



We can implement this directly, with a little manipulation of the pairs:

$$\begin{array}{l}\textbf{instance}\ \mathit{ArrowLoop}\ (\mathit{State}\ \sigma)\ \textbf{where}\\ \quad \mathit{loop}\ (\mathit{ST}\ f) = \mathit{ST}\ (\mathit{trace}\ (\mathit{unassoc} \cdot f \cdot \mathit{assoc}))\end{array}$$

This definition is almost an inside-out version of the definition of *first* for state transformers. The instance for map transformers is similar:

$$\begin{array}{l}\textbf{instance}\ \mathit{ArrowLoop}\ (\mathit{MapTrans}\ \sigma)\ \textbf{where}\\ \quad \mathit{loop}\ (\mathit{MT}\ f) = \mathit{MT}\ (\mathit{trace}\ (\mathit{unzipMap} \cdot f \cdot \mathit{zipMap}))\end{array}$$

The instance for automata is more subtle. At each stage, part of the output is fed back to the input:

$$\begin{array}{l}\textbf{instance}\ \mathit{ArrowLoop}\ \mathit{Auto}\ \textbf{where}\\ \quad \mathit{loop}\ (A\ f) = A\ (\lambda\ b\ \to\ \textbf{let}\ (\tilde{}(c, d), f') = f\ (b, d)\\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ (c, \mathit{loop}\ f'))\end{array}$$

However there is no *loop* operator for non-deterministic functions, unless we are prepared to relax our axioms.

Exercise 11 Define an *ArrowLoop* instance for the *StreamMap* type from Exercise 3. □

Exercise 12 Prove that $\mathit{loop}\ (\mathit{first}\ f) = f$. □

3 Arrow notation

Arrows present a usefully general interface to computation, but we have seen that they force a point-free style. This may be convenient for general definitions and proofs, but it can be cumbersome for specific programming. For example, suppose

Minor rephrasing to improve pagebreaks

we have the following operations for the state transformer arrow:

$$
\begin{aligned}
&\mathit{fetch} :: \mathit{State}\,\sigma\,()\,\sigma\\
&\mathit{fetch} = \mathit{ST}\,(\lambda\,(s,())\ \rightarrow\ (s,s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{store} :: \mathit{State}\,\sigma\,\sigma\,()\\
&\mathit{store} = \mathit{ST}\,(\lambda\,(s,s')\ \rightarrow\ (s',()))
\end{aligned}
$$

The following arrow uses an integer state to generate a fresh number:

$$
\begin{aligned}
&\mathit{genSym} :: \mathit{State}\ \mathit{Int}\,()\,\mathit{Int}\\
&\mathit{genSym} = \mathit{fetch} \ggg \mathit{pure}\ \mathit{incr} \ggg \mathit{first}\ \mathit{store} \ggg \mathit{pure}\ \mathit{snd}\\
&\qquad\qquad\ \ \mathbf{where}\ \ \mathit{incr}\ n =\ (n+1,n)
\end{aligned}
$$

Note that we must explicitly describe the plumbing, duplicating the n, passing the new value to *store* and then discarding its result.

To facilitate programming with arrows, we extend the Haskell syntax with a new sort of expression, the **proc** expression for defining arrows. Here is a version of *genSym* in the new notation:

$$
\begin{aligned}
&\mathit{genSym} :: \mathit{State}\ \mathit{Int}\,()\,\mathit{Int}\\
&\mathit{genSym} = \mathbf{proc}\,()\ \rightarrow\ \mathbf{do}\\
&\qquad\qquad\qquad n\ \leftarrow\ \mathit{fetch}\ \prec\ ()\\
&\qquad\qquad\qquad \mathit{store}\ \prec\ n+1\\
&\qquad\qquad\qquad \mathit{idA}\ \prec\ n
\end{aligned}
$$

The right-hand side is a **proc** expression, a variation on a lambda expression that defines an arrow. In this case the input has the form (). The next line passes a value () through the arrow *fetch*, naming the result n. (The $\prec$ symbol is meant to signify an arrow tail.) The value $n+1$ is then sent to *store*. Finally n is sent to the identity arrow *idA* to produce the output of the whole arrow.

The new expressions are defined by the grammar of Figure 1. A *command* occurs as the body of a **proc** binding, and elsewhere. It returns a value, but may also have some effect. The simplest sort of command is built using the arrow tail $\prec$ which acts as a sort of application of an arrow to an input expression. The above example includes sub-commands $\mathit{fetch}\ \prec\ ()$ and $\mathit{store}\ \prec\ n+1$ that are of this form. The **do** command consists of a series of *statements*, which may define variables or simply execute commands for effect, followed by a final command, in the above example $\mathit{idA}\ \prec\ n$.

These new forms are given meaning by translation to ordinary Haskell, using the rules of Figure 2. Note in particular the two translations for arrow application ($\prec$). The first is syntactically restricted, but is valid for any arrow. In the special case where f is *idA*, we have

$$\mathbf{proc}\,p\ \rightarrow\ \mathbf{do}\,\mathit{idA}\ \prec\ e \quad = \quad \mathit{pure}\,(\lambda\,p\ \rightarrow\ e)$$

The second possibility has no such syntactic restriction, but because its translation uses *app*, the arrow in the type of the arrow expression must be an instance of the

$$
\begin{array}{lcl}
exp & = & \ldots \\
 & | & \mathbf{proc}\ pat \rightarrow cmd \\
cmd & = & exp \prec exp \\
 & | & \mathbf{do}\ \{\ stmt; \ldots; stmt; cmd\ \} \\
stmt & = & pat \leftarrow cmd \\
 & | & cmd \\
 & | & \mathbf{rec}\ \{\ stmt; \ldots; stmt\ \}
\end{array}
$$

Figure 1: Grammar for arrow expressions

$$
\mathbf{proc}\ p \rightarrow f \prec a \triangleq \begin{cases} pure\ (\lambda p \rightarrow a) \ggg f & \text{if } FV(p) \cap FV(f) = \varnothing \\ pure\ (\lambda p \rightarrow (f, a)) \ggg app & \text{otherwise} \end{cases}
$$

$$
\mathbf{proc}\ p \rightarrow \mathbf{do}\ \{\ c\ \} \triangleq \mathbf{proc}\ p \rightarrow c
$$

$$
\begin{aligned}
\mathbf{proc}\ p \rightarrow \mathbf{do}\ \{\ p' \leftarrow c; B\ \} \triangleq{} & ((\mathbf{proc}\ p \rightarrow c)\ \&\&\&\ idA) \ggg \\
& \mathbf{proc}\ (p', p) \rightarrow \mathbf{do}\ \{\ B\ \}
\end{aligned}
$$

$$
\mathbf{proc}\ p \rightarrow \mathbf{do}\ \{\ c; B\ \} \triangleq \mathbf{proc}\ p \rightarrow \mathbf{do}\ \{\ \_ \leftarrow c; B\ \}
$$

$$
\begin{aligned}
& \mathbf{proc}\ p \rightarrow \mathbf{do}\ \{\ \mathbf{rec}\ \{\ A\ \}; B\} \triangleq \\
& \qquad idA\ \&\&\&\ loop\ (\mathbf{proc}\ (p, p_A) \rightarrow \mathbf{do}\ \{\ A; idA \prec (p_B, p_A)\ \}) \ggg \\
& \qquad \mathbf{proc}\ (p, p_B) \rightarrow \mathbf{do}\ \{\ B\ \}
\end{aligned}
$$

Figure 2: Translation rules for arrow expressions

ArrowApply class. Thus neither form is more general than the other, and both are useful.

The third rule deals with binding of new variables. As the translation makes clear, a copy of the original input (described by p) must be routed around the command c for use in the rest of the **do** command, which also has access to the output of c, bound by p'. The next rule is the special case where we wish to discard the output of a command, as with *store* in the *genSym* example.

The last rule, dealing with recursion, is more involved. The right side may be visualized as follows:

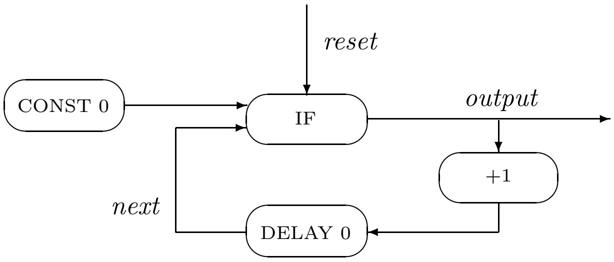
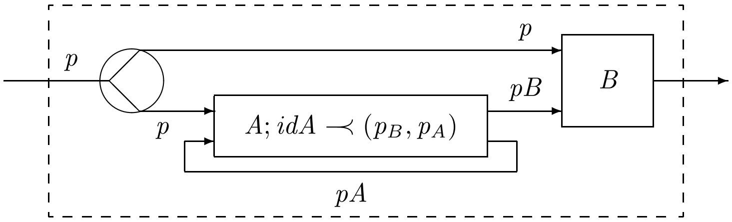


Figure 3: A resettable counter circuit



Here pA is a pattern containing all the variables that are both defined and used in A, while the pattern pB contains the variables defined in A that are used in B. Note that if recursion is used, the arrow involved must be an instance of *ArrowLoop*.

Exercise 13 Use the rules of Figure 2 to translate the above definition of *genSym* into ordinary Haskell, and use the axioms to simplify the result. Compare your answer with the point-free version. □

Exercise 14 Prove that when both translations of **proc** $p \rightarrow f \prec a$ are possible, they are equal. □

Exercise 15 Suppose the syntax is extended with a new form of command

if *exp* **then** *cmd* **else** *cmd*

Suggest a translation for the new form, assuming the arrow concerned belongs to *ArrowChoice*. A **case** command could be defined similarly. □

4 Examples

4.1 Synchronous circuits

A synchronous circuit receives an input and produces an output on each tick of some global clock. The output for a given tick may depend on the input for that tick, as well as previous inputs. Consider the simple circuit of Figure 3 (taken from [13]). This circuit represents a resettable counter, taking a Boolean input and producing an integer output, which will be the number of clock ticks since the input was last *True*. To achieve this, the output is incremented and fed back, delayed by one clock cycle. The first output of the DELAY component is its argument, here 0; its second output is its first input, and so on.

We propose to treat circuits as arrows. It suffices to consider circuits with a single input and output, because multiple inputs may be treated as input of a tuple, and similarly for output.

- The *pure* operation defines circuits where each output is a pure function of the corresponding input (*e.g.* IF and +1 in the above circuit).
- Composition connects the output of the first circuit to the input of the second.
- The *first* operation channels part of the input to a subcircuit, with the rest copied directly to the output.

As with any arrow instance, we shall require additional operations. We define circuits as arrows that support cycles and a delay arrow:

$$\begin{array}{l}\textbf{class}\ \mathit{ArrowLoop}\ \alpha \Rightarrow \mathit{ArrowCircuit}\ \alpha\ \textbf{where}\\ \quad \mathit{delay} :: \beta \to \alpha\,\beta\,\beta\end{array}$$

The argument supplies the initial output; subsequent outputs are copied from the input of the previous tick. A circuit built with *loop* will not work unless it includes a *delay* somewhere on its second input before using it, as in the example above. One could enforce this by combining the two in a single construct, but the present formulation is better suited to algebraic manipulation.

Here is the resettable counter circuit in arrow notation:

$$\begin{array}{l}\mathit{counter} :: \mathit{ArrowCircuit}\ \alpha \Rightarrow \alpha\ \mathit{Bool}\ \mathit{Int}\\ \mathit{counter} = \textbf{proc}\ \mathit{reset} \to \textbf{do}\\ \qquad\qquad\qquad \textbf{rec}\ \mathit{output} \leftarrow \mathit{idA} \prec \textbf{if}\ \mathit{reset}\ \textbf{then}\ 0\ \textbf{else}\ \mathit{next}\\ \qquad\qquad\qquad\qquad \mathit{next} \leftarrow \mathit{delay}\ 0 \prec \mathit{output} + 1\\ \qquad\qquad\qquad \mathit{idA} \prec \mathit{output}\end{array}$$

This corresponds rather directly to the graphical presentation of Figure 3. The variables denote the values passing through wires on a particular clock tick.

Several implementations of the *ArrowCircuit* class are possible, allowing many different interpretations of circuit descriptions. For example, we already have an *ArrowLoop* instance for simple automata, so we need only implement *delay*:

$$\begin{array}{l}\textbf{instance}\ \mathit{ArrowCircuit}\ \mathit{Auto}\ \textbf{where}\\ \quad \mathit{delay}\ b = A\ (\lambda\, b' \to (b, \mathit{delay}\ b'))\end{array}$$

To simulate a circuit, we feed it a list of inputs, extracting a list of the same number of outputs:

$$\begin{array}{ll}\mathit{runAuto} & :: \mathit{Auto}\,\beta\,\gamma \to [\beta] \to [\gamma]\\ \mathit{runAuto}\ (A\,f)\ [\,] & = [\,]\\ \mathit{runAuto}\ (A\,f)\ (b : bs) & = \textbf{let}\ (c, f') = f\ b\ \textbf{in}\ (c : \mathit{runAuto}\ f'\ bs)\end{array}$$

However many other other arrows satisfy the *ArrowCircuit* interface, permitting a range of interpretations of a single circuit description. An interpretation could add probes that show intermediate values, or calculate static properties of a circuit, such as a wiring description.

Exercise 16 Define an *ArrowCircuit* instance for the *StreamMap* type from Exercise 3. □

4.2 Homogeneous functions

Many parallel algorithms and circuit designs operate on collections of 2^n elements, with behaviour defined by induction on n. We can model these collections in Haskell using the following type:

$$\begin{array}{ll} \mathbf{data}\ \mathit{BalTree}\ \alpha & = \mathit{Zero}\ \alpha \mid \mathit{Succ}\ (\mathit{BalTree}\ (\mathit{Pair}\ \alpha)) \\ \mathbf{type}\ \mathit{Pair}\ \alpha & = (\alpha, \alpha) \end{array}$$

Here are some example elements:

$$\begin{array}{l} \mathit{tree}0 = \mathit{Zero}\ 1 \\ \mathit{tree}1 = \mathit{Succ}\ (\mathit{Zero}\ (1,2)) \\ \mathit{tree}2 = \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ ((1,2),(3,4)))) \\ \mathit{tree}3 = \mathit{Succ}\ (\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (((1,2),(3,4)),((5,6),(7,8)))))) \end{array}$$

The elements of this type have a string of constructors expressing a depth n as a Peano numeral, enclosing a nested pair tree of 2^n elements. However few algorithms can be expressed in terms of the balanced tree type. Typically one wants to split a tree into two subtrees, do some processing on the subtrees and combine the results. But the type system cannot discover that the two results are of the same depth (and thus combinable).

The solution is to define a type we call *homogeneous functions*, namely families of functions mapping trees of size 2^n to trees of the same size 2^n:

$$\mathit{data}\ \mathit{Hom}\ \alpha\ \beta\ =\ (\alpha\ \to\ \beta)\ :\!\&\!:\ \mathit{Hom}\ (\mathit{Pair}\ \alpha)\ (\mathit{Pair}\ \beta)$$

Elements of this type have the form

$$f_0 :\!\&\!: f_1 :\!\&\!: f_2 :\!\&\!: \ldots$$

where $f_n :: \mathit{Pair}^n\ \alpha \to \mathit{Pair}^n\ \beta$.

The following function applies a homogeneous function to a perfectly balanced tree, yielding another perfectly balanced tree of the same depth:

$$\begin{array}{ll} \mathit{apply} & :: \mathit{Hom}\ \alpha\ \beta\ \to\ \mathit{BalTree}\ \alpha\ \to\ \mathit{BalTree}\ \beta \\ \mathit{apply}\ (f\ :\!\&\!:\ \mathit{fs})\ (\mathit{Zero}\ x) & = \mathit{Zero}\ (f\ x) \\ \mathit{apply}\ (f\ :\!\&\!:\ \mathit{fs})\ (\mathit{Succ}\ t) & = \mathit{Succ}\ (\mathit{apply}\ \mathit{fs}\ t) \end{array}$$

Having defined *apply*, we can program with the *Hom* type and forget about *BalTree*. Firstly, *Hom* is an arrow:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arrow\ Hom}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{pure}\, f & = f\ \text{:\&:}\ \mathit{pure}\,(f \times f) \\
(f\ \text{:\&:}\ \mathit{fs}) \ggg (g\ \text{:\&:}\ \mathit{gs}) & = (g \cdot f)\ \text{:\&:}\ (\mathit{fs} \ggg \mathit{gs}) \\
\mathit{first}\,(f\ \text{:\&:}\ \mathit{fs}) & = \\
\multicolumn{2}{r}{(f \times \mathit{id})\ \text{:\&:}\ (\mathit{pure\ transpose} \ggg \mathit{first\ fs} \ggg \mathit{pure\ transpose})}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{transpose} & :: ((\alpha,\beta),(\gamma,\delta)) \rightarrow ((\alpha,\gamma),(\beta,\delta)) \\
\mathit{transpose}\,((a,b),(c,d)) & = ((a,c),(b,d))
\end{array}
$$

The function *pure* maps a function over the leaves of the tree. The composition $\ggg$ composes sequences of functions pairwise. The $***$ operator unriffles a tree of pairs (α, β) into a tree of αs and a tree of βs, applies the appropriate function to each tree and riffles the results.

When describing algorithms, one often provides a pure function for the base case (trees of one element) and an expression for trees of pairs, usually invoking the same algorithm on smaller trees.

Parallel Prefix: This operation (also called *scan*) maps the sequence

$$x_1, x_2, x_3, \ldots, x_{2^n}$$

to the sequence

$$x_1,\ x_1 + x_2,\ x_1 + x_2 + x_3,\ \ldots,\ x_1 + x_2 + \cdots + x_{2^n}$$

(All this generalizes easily from + to any associative operation.)

If there is only one element (*i.e.* the tree has zero depth) then obviously the scan should be the identity function. Otherwise, we need to deal with a tree of pairs, so the general *scan* operation will have the form

$$
\begin{array}{l}
\mathit{scan} :: \mathit{Num}\ \beta \Rightarrow \mathit{Hom}\ \beta\ \beta \\
\mathit{scan} = \mathit{id}\ \text{:\&:}\ \textbf{proc}\,(x, y) \rightarrow \langle \text{something using } \mathit{scan} \text{ on smaller trees} \rangle
\end{array}
$$

An efficient scheme, devised by Ladner and Fischer [12], is first to sum the elements pairwise:

$$x_1 + x_2, x_3 + x_4, x_5 + x_6, \ldots$$

and then to compute the scan of this list (which is half the length of the original), yielding

$$x_1 + x_2,\ x_1 + x_2 + x_3 + x_4,\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6,\ \ldots$$

This list is half of the desired answer; the other elements are

$$x_1,\ x_1 + x_2 + x_3,\ x_1 + x_2 + x_3 + x_4 + x_5,\ \ldots$$

which can be obtained by shifting our partial answer one place to the right and adding $x_1, x_3, x_5, \ldots$. We can express this idea directly in our notation:

$$
\begin{aligned}
&\mathit{scan} :: \mathit{Num}\,\beta \Rightarrow \mathit{Hom}\,\beta\,\beta \\
&\mathit{scan} = \mathit{id} \mathbin{:\&:} \mathit{proc}\,(o, e) \rightarrow \mathbf{do} \\
&\qquad\qquad e' \leftarrow \mathit{scan} \prec o + e \\
&\qquad\qquad \mathit{el} \leftarrow \mathit{rsh}\,0 \prec e' \\
&\qquad\qquad \mathit{idA} \prec (\mathit{el} + o, e')
\end{aligned}
$$

The auxiliary arrow $\mathit{rsh}\ b$ shifts each element in the tree one place to the right, placing b in the now-vacant leftmost position, and discarding the old rightmost element. This could be supplied as a primitive, but it is also possible to code it directly:

$$
\begin{aligned}
&\mathit{rsh} \;\; :: \beta \rightarrow \mathit{Hom}\,\beta\,\beta \\
&\mathit{rsh}\ v = \mathit{const}\ v \mathbin{:\&:} \mathit{proc}\,(o, e) \rightarrow \mathbf{do} \\
&\qquad\qquad o' \leftarrow \mathit{rsh}\ v \prec e \\
&\qquad\qquad \mathit{idA} \prec (o', o)
\end{aligned}
$$

Butterfly Circuits: In many divide-and-conquer schemes, one recursive call processes the odd-numbered elements and the other the even ones [9]:

deleted second 'processes' to avoid bad linebreak

$$
\begin{aligned}
&\mathit{butterfly} \;\; :: (\mathit{Pair}\,\beta \rightarrow \mathit{Pair}\,\beta) \rightarrow \mathit{Hom}\,\beta\,\beta \\
&\mathit{butterfly}\,f = \mathit{id} \mathbin{:\&:} \mathit{proc}\,(o, e) \rightarrow \mathbf{do} \\
&\qquad\qquad o' \leftarrow \mathit{butterfly}\,f \prec o \\
&\qquad\qquad e' \leftarrow \mathit{butterfly}\,f \prec e \\
&\qquad\qquad \mathit{idA} \prec f\,(o', e')
\end{aligned}
$$

The recursive calls operate on halves of the original tree, so the recursion is well-defined. (The Fast Fourier Transform has a similar structure. See also Chapter **??** for another view.) Here are some examples of butterflies:

$$
\begin{aligned}
&\mathit{rev} \;\; :: \mathit{Hom}\,\beta\,\beta \\
&\mathit{rev} \;\; = \mathit{butterfly}\ \mathit{swap} \\[4pt]
&\mathit{unriffle} :: \mathit{Hom}\,(\mathit{Pair}\,\beta)\,(\mathit{Pair}\,\beta) \\
&\mathit{unriffle} = \mathit{butterfly}\ \mathit{transpose}
\end{aligned}
$$

Batcher's ingenious sorter for bitonic sequences [1] is another example of a butterfly circuit:

$$
\begin{aligned}
&\mathit{bisort} :: \mathit{Ord}\,\beta \Rightarrow \mathit{Hom}\,\beta\,\beta \\
&\mathit{bisort} = \mathit{butterfly}\ \mathit{cmp} \\
&\qquad \mathbf{where}\ \ \mathit{cmp}\,(x, y) = (\mathit{min}\ x\ y, \mathit{max}\ x\ y)
\end{aligned}
$$

Exercise 17 Use the functions defined in this section to define an arrow

$$\mathit{sort} :: \mathit{Ord}\,\beta \Rightarrow \mathit{Hom}\,\beta\,\beta$$

using a merge based on Batcher's bitonic sorter, combined with rev. □

4.3 Combining arrows

We have seen that each arrow type embodies a notion of computation. Sometimes we want to combine two such notions, *e.g.* dataflow with state. The trick is to generalize one of the arrows to an *arrow transformer*, and then apply it to the other one. For example, state transformers were defined using functions, but we can generalize to any arrow:

$$\textbf{newtype}\ \mathit{StateT}\ \sigma\ \alpha\ \iota\ o = \mathit{ST}\ (\alpha\ (\sigma, \iota)\ (\sigma, o))$$

Now we can define a new arrow for each arrow α, as follows:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ \alpha \Rightarrow \mathit{Arrow}\ (\mathit{StateT}\ \sigma\ \alpha)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{pure}\ f & = \mathit{ST}\ (\textbf{proc}\ (s, b) \rightarrow \mathit{idA} \prec (s, f\ b)) \\
\mathit{ST}\ f \ggg \mathit{ST}\ g & = \mathit{ST}\ (f \ggg g) \\
\mathit{first}\ (\mathit{ST}\ f) & = \mathit{ST}\ (\textbf{proc}\ (s, (b, d)) \rightarrow \textbf{do} \\
& \qquad\quad (s', c) \leftarrow f \prec (s, b) \\
& \qquad\quad \mathit{idA} \prec (s', (c, d)))
\end{array}
\end{array}$$

The arrow notation is useful here, especially in the definition of *first*. Now *State* may be defined as a special case:

$$\textbf{type}\ \mathit{State}\ \sigma = \mathit{StateT}\ \sigma\ (\rightarrow)$$

The transformer *StateT* could be also applied to *Auto*, yielding an automaton that also transforms a state on each iteration.

Arrows can also model static data, which is independent of the input. For example, the following arrow transformer augments an arrow with an integer count:

$$\textbf{data}\ \mathit{Count}\ \alpha\ \iota\ o = \mathit{Count}\ \mathit{Int}\ (\alpha\ \iota\ o)$$

The arrow combinators of the base arrow lift directly to the new arrow:

$$\begin{array}{l}
\textbf{instance}\ \mathit{Arrow}\ \alpha \Rightarrow \mathit{Arrow}\ (\mathit{Count}\ \alpha)\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{pure}\ f & = \mathit{Count}\ 0\ (\mathit{pure}\ f) \\
\mathit{Count}\ n1\ f1 \ggg \mathit{Count}\ n2\ f2 & = \mathit{Count}\ (n1 + n2)\ (f1 \ggg f2) \\
\mathit{first}\ (\mathit{Count}\ n\ f) & = \mathit{Count}\ n\ (\mathit{first}\ f)
\end{array}
\end{array}$$

Other arrow combinators lift similarly:

$$\begin{array}{l}
\textbf{instance}\ \mathit{ArrowChoice}\ \alpha \Rightarrow \mathit{ArrowChoice}\ (\mathit{Count}\ \alpha)\ \textbf{where} \\
\quad \mathit{left}\ (\mathit{Count}\ n\ f) = \mathit{Count}\ n\ (\mathit{left}\ f)
\end{array}$$

This is a very simple example, but the ability to handle such static information enables arrows to express many efficient algorithms. An example is parser combinators that compute properties of the grammar, independent of the input being parsed [17].

Exercise 18 Rewrite the definition of $StateT$ without arrow notation. $\square$

Exercise 19 Given the following definition,

$$\textbf{newtype}\ AutoFunctor\ \alpha\ \iota\ o = A\ (\alpha\ \iota\ (o, AutoFunctor\ \alpha\ \iota\ o))$$

write the following instance, using arrow notation (or not):

$$\textbf{instance}\ Arrow\ \alpha\ \Rightarrow\ Arrow\ (AutoFunctor\ \alpha)$$

$\square$

5 Chapter notes

Arrows were invented as a programming abstraction by John Hughes [8], to deal with libraries that did not fit the monad model. Unknown to Hughes, workers in denotational semantics had also defined generalizations of monads, based on *premonoidal categories* [16]. A special case of these structures, later called a *Freyd-category*, turns out to be identical to Hughes's definition.

Hughes also introduced the $ArrowApply$ and $ArrowChoice$ classes, though the distribution axiom is new here. The $ArrowLoop$ class [15] generalizes traces, defined by Joyal, Street and Verity [11], and the recursive monads of Erkök and Launchbury [7]. A more primitive and powerful version of the arrow notation sketched here may also be found in [15]. More information about arrows, including a preprocessor implementation this notation, may be found on the web page `http://www.haskell.org/arrows/`.

The embedded language FRP [6], used to model reactive situations such as robotics and graphical interfaces, has recently been reformulated in terms of arrows [5]. Map transformers are used as the abstract semantics, though implementations use variants of stream maps or automata.

Several dataflow languages have been used to model circuits [2, 4, 18]. The microarchitecture design language Hawk [13] abstracts over the type of values that may pass through wires. Low-level descriptions deal with bits ($Bool$), but any Haskell type may be used, allowing Hawk to scale to much more abstract descriptions, and also allowing the same circuit description to be simulated or symbolically executed. Related ideas are discussed in Chapter **??**. There, circuits are functions, but in another version of the hardware description language Lava [3], circuits have the form $Value \rightarrow Monad\ Value'$ where both value and monad types are parameters described by Haskell classes. By selecting appropriate instances, a single description may be simulated, symbolically executed or presented in a variety of styles.

Formalisms related to our homogeneous functions include Ruby [10], used in circuit design, and Misra's powerlists [14].

Hughes [8] also introduced arrow transformers, including the state arrow transformer.

References

[1] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Conference*, pages 307–314, 1968.

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

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*. ACM, 1998.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[5] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, pages 41–69, 2001.

[6] C. Elliott and P. Hudak. Functional reactive programming. In *International Conference on Functional Programming*, pages 163–173, 1997.

[7] L. Erkök and J. Launchbury. Recursive monadic bindings. In *International Conference on Functional Programming*, pages 174–185, 2000.

[8] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[9] G. Jones and M. Sheeran. Collecting butterflies. Technical Monograph PRG-91, Oxford University Computing Laboratory, Programming Research Group, Feb. 1991.

[10] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 208–232. Springer, 1993.

[11] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society*, 119(3):447–468, 1996.

[12] R. Ladner and M. Fischer. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.

[13] J. Launchbury, J. Lewis, and B. Cook. On embedding a microarchitecture design language within Haskell. In *International Conference on Functional Programming*. ACM, 1999.

[14] J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. Prog. Lang. Syst.*, 16(6):1737–1767, Nov. 1994.

[15] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[16] J. Power and E. Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5):453–468, Oct. 1997.

[17] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer, 1996.

[18] W. Wadge and E. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.