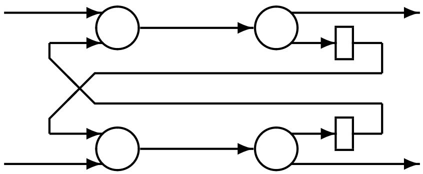
Arrows and Computation

Ross Paterson
City University, London

Example: describing synchronous circuits

Problem: describe clocked hardware circuits



The ideal:

$$\text{single description} \Longrightarrow \begin{cases} \text{simulation} \\ \text{simulation with probes} \\ \text{static properties} \\ \text{representation of the graph} \\ \vdots \end{cases}$$

The plan

- use a combinator library
- or rather, several libraries with a common interface (Haskell classes + axioms)
- much of this interface can be shared with other applications ("arrows"/Freyd-categories as notions of computation)
- additional language support is helpful
- result: an embedded domain-specific language
- this is just one example

Circuits

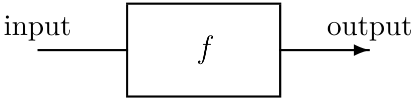
wires through which values of a given type pass.

$$\xrightarrow{\qquad a \qquad}$$

communication is synchronous:

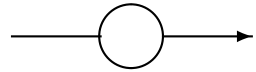
$$\begin{array}{c} \xrightarrow{\qquad a \qquad} \\ \\ \xrightarrow{\qquad b \qquad} \end{array} \quad \cong \quad \xrightarrow{\qquad a \times b \qquad}$$

components (computations) have input and output wires:

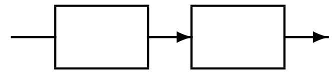


Building circuits

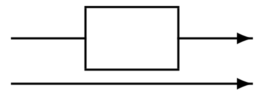
pure functions



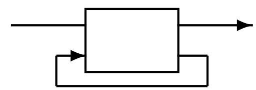
composition



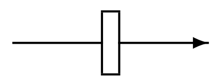
bypass



feedback



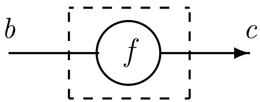
primitive components



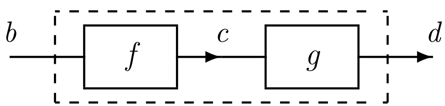
The Arrow class (Hughes, 1997, 2000)

class $\mathit{Arrow}\ \mathrm{a}$ **where**

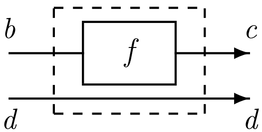
$$\mathrm{pure} :: (\mathrm{b} \to \mathrm{c}) \to \mathrm{a\ b\ c}$$



$$(\ggg) :: \mathrm{a\ b\ c} \to \mathrm{a\ c\ d} \to \mathrm{a\ b\ d}$$

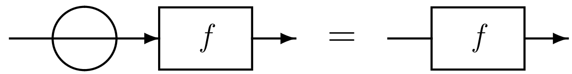


$$\mathrm{first} :: \mathrm{a\ b\ c} \to \mathrm{a\ (b, d)\ (c, d)}$$

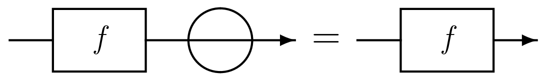


Arrow axioms

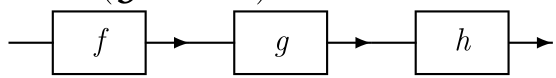
$$\mathrm{pure}\ \mathrm{id} \ggg f = f$$



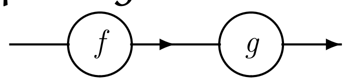
$$f \ggg \mathrm{pure}\ \mathrm{id} = f$$



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

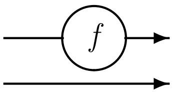


$$\mathrm{pure}\ (g \cdot f) = \mathrm{pure}\ f \ggg \mathrm{pure}\ g$$

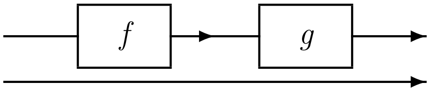


Axioms of first

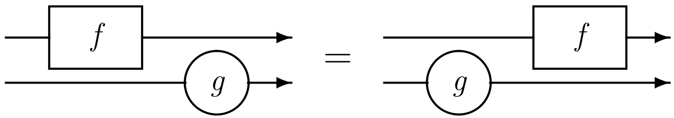
$$\mathrm{first}\ (\mathrm{pure}\ f) \ = \ \mathrm{pure}\ (f \times \mathrm{id})$$



$$\mathrm{first}\ (f \ggg g) \ = \ \mathrm{first}\ f \ggg \mathrm{first}\ g$$



$$\mathrm{first}\ f \ggg \mathrm{pure}\ (\mathrm{id} \times g) \ = \ \mathrm{pure}\ (\mathrm{id} \times g) \ggg \mathrm{first}\ f$$

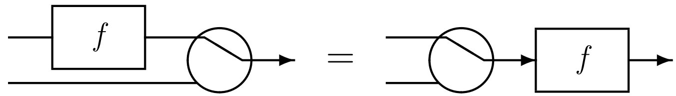


for the functor

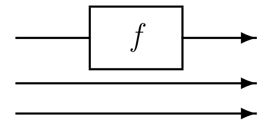
$$(\times) :: (a \to a') \to (b \to b') \to (a, b) \to (a', b')$$
$$(f \times g)\ (a, b) = (f\ a, g\ b)$$

Elimination and associativity

$$\mathrm{first}\ f \ggg \mathrm{pure}\ \mathrm{fst} \;=\; \mathrm{pure}\ \mathrm{fst} \ggg f$$



$$\mathrm{first}\ (\mathrm{first}\ f) \ggg \mathrm{pure}\ \mathrm{assoc} \;=\; \mathrm{pure}\ \mathrm{assoc} \ggg \mathrm{first}\ f$$



for the function

$$\mathrm{assoc} :: ((a, b), c) \to (a, (b, c))$$
$$\mathrm{assoc}\ ((a, b), c) = (a, (b, c))$$

Freyd-categories (Power & Robinson, 1977)

Categories $\mathcal{V}$ (values) and $\mathcal{C}$ (computations) with the same objects and

$$\begin{tikzcd}
\mathcal{V} \times \mathcal{V} \arrow[r, "\mathrm{pure} \times \mathcal{V}"] \arrow[d, "\times"'] & \mathcal{C} \times \mathcal{V} \arrow[d, "\ltimes"] \\
\mathcal{V} \arrow[r, "\mathrm{pure}"'] & \mathcal{C}
\end{tikzcd}$$

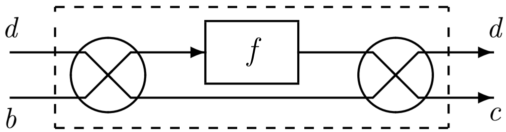
$\boxed{\begin{array}{l} \text{action of } \langle \mathcal{V}, \times, 1 \rangle \text{ on } \mathcal{C} \\ \mathrm{f} \ltimes \mathrm{g} = \mathrm{first\ f} \ggg \mathrm{pure}\ (\mathrm{id} \times \mathrm{g}) \\ \qquad\quad = \mathrm{pure}\ (\mathrm{id} \times \mathrm{g}) \ggg \mathrm{first\ f} \end{array}}$

$\boxed{\text{strict transformation of actions}}$

General case: *premonoidal categories*.

Derived combinators

$$\mathrm{second} :: \mathit{Arrow}\ a \Rightarrow a\ b\ c \to a\ (d, b)\ (d, c)$$
$$\mathrm{second}\ f = \mathrm{pure}\ \mathit{swap} \ggg \mathrm{first}\ f \ggg \mathrm{pure}\ \mathit{swap}$$
$$\textbf{where}\ \mathit{swap}\tilde{\ }(x, y) = (y, x)$$



$$\mathrm{id}A :: \mathit{Arrow}\ a \Rightarrow a\ b\ b$$
$$\mathrm{id}A = \mathrm{pure}\ \mathrm{id}$$

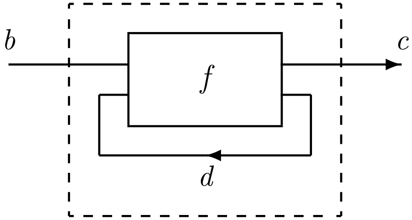
Examples of arrow types

ordinary functions	$b \to c$
Kleisli arrows	$b \to M\ c$, for any monad M
dual Kleisli arrows	$W\ b \to c$, for any comonad W
state/behaviour transformers	$(S \to a) \to (S \to b)$, for any set S
stream transformers	$\mathit{Stream}\ b \to \mathit{Stream}\ c$
static arrows	$F\ (b \to c)$, for suitable functors F
automata	$\nu\, x.\ (b \to (c, x))$
hyperfunctions	$\nu\, x.\ ((x \to b) \to c)$ (KLP, FICS'2001)
etc	

Recursion: A feedback operator

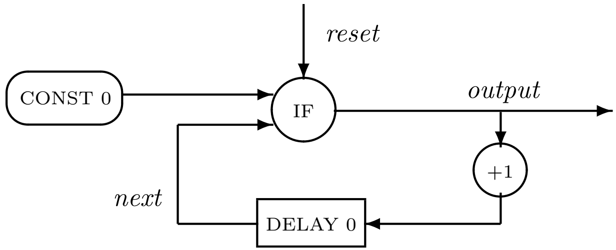
Many (but not all) arrows have an operator

$$\textbf{class}\ Arrow\ a \Rightarrow ArrowLoop\ a\ \textbf{where}$$
$$\quad loop :: a\ (b, d)\ (c, d) \to a\ b\ c$$



Generalizes traces (Joyal, Street and Verity, 1996) and recursive monads (Erkök and Launchbury, 2000).

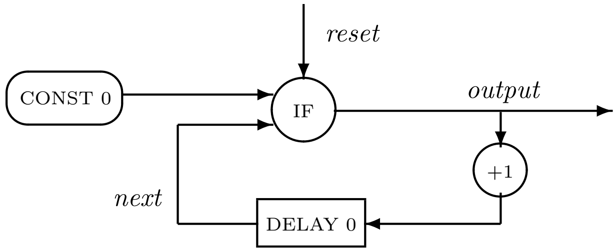
Example: a resettable counter



$$\textbf{class}\ \mathrm{ArrowLoop}\ a \Rightarrow \mathrm{ArrowCircuit}\ a\ \textbf{where}$$
$$\quad \mathrm{delay} :: b \to a\ b\ b$$

$$\mathrm{counter} :: \mathrm{ArrowCircuit}\ a \Rightarrow a\ \mathrm{Bool}\ \mathrm{Int}$$
$$\mathrm{counter} = \mathrm{loop}\ (\mathrm{pure\ cond} \ggg \mathrm{pure\ dup} \ggg$$
$$\qquad \mathrm{second}\ (\mathrm{pure}\ (+1) \ggg \mathrm{delay}\ 0))$$
$$\quad \textbf{where}\ \mathrm{cond}\ (\mathrm{reset}, \mathrm{next}) = \textbf{if}\ \mathrm{reset}\ \textbf{then}\ 0\ \textbf{else}\ \mathrm{next}$$
$$\qquad \mathrm{dup}\ x = (x, x)$$

Resettable counter in arrow notation



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

New syntax: arrow expressions

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

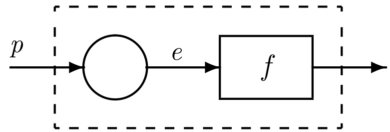
with semantics by translation into Haskell.

(implemented using a preprocessor)

Arrow application

$$\boxed{\textbf{proc}\ p \rightarrow \textbf{do}\ \{\ f \mathrel{-\!\!\prec} e\ \} \triangleq \mathrm{pure}\ (\lambda\ p \rightarrow e) \ggg f}$$

(variables of p not free in f — relaxed for Kliesli arrows)

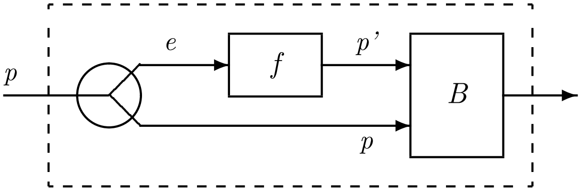


A special case:

$$\textbf{proc}\ p \rightarrow \textbf{do}\ \mathrm{id}A \mathrel{-\!\!\prec} e \ = \ \mathrm{pure}\ (\lambda\ p \rightarrow e)$$

Sequencing and binding

$$\boxed{\begin{aligned}\textbf{proc}\ p \to \textbf{do}\ \{\ p' \leftarrow f \prec e; B\ \} \triangleq\ & \mathtt{pure}\ (\lambda\, p \to (e, p)) \ggg \\ & \mathtt{first}\ f \ggg \\ & \textbf{proc}\ (p', p) \to \textbf{do}\ \{\ B\ \}\end{aligned}}$$



A special case:

$$\boxed{\textbf{proc}\ p \to \textbf{do}\ \{\ f \prec a; B\ \} \triangleq \textbf{proc}\ p \to \textbf{do}\ \{\ \_ \leftarrow f \prec a; B\ \}}$$

An example translation

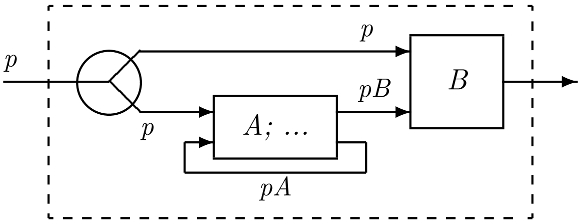
$$
\begin{aligned}
& \textbf{proc}\ x \to \textbf{do} \\
= \ & \mathrm{pure}\ (\lambda x \to (x, x)) \ggg \mathrm{first}\ f \ggg \\
& \textbf{proc}\ (y, x) \to \textbf{do} \\
= \ & \mathrm{pure}\ (\lambda x \to (x, x)) \ggg \mathrm{first}\ f \ggg \\
& \mathrm{pure}\ (\lambda(y, x) \to (x, (y, x))) \ggg \mathrm{first}\ g \ggg \\
& \textbf{proc}\ (z, (y, x)) \to \textbf{do} \\
= \ & \mathrm{pure}\ (\lambda x \to (x, x)) \ggg \mathrm{first}\ f \ggg \\
& \mathrm{pure}\ (\lambda(y, x) \to (x, (y, x))) \ggg \mathrm{first}\ g \ggg \\
& \mathrm{pure}\ (\lambda(z, (y, x)) \to y + z) \\
= \ & \text{(simplify)} \\
& \mathrm{pure}\ \mathrm{dup} \ggg \mathrm{first}\ f \ggg \mathrm{second}\ g \ggg \mathrm{pure}\ (\lambda(y, z) \to y + z)
\end{aligned}
$$

Recursive definitions

Translation of

$$\textbf{proc}\ p \to \textbf{do}\ \{\ \textbf{rec}\ \{\ A\ \}; B\}$$

uses loop:



Interpretations: Stream processors

$$\textbf{data}\ \mathrm{Stream}\ b = \mathrm{Cons}\ b\ (\mathrm{Stream}\ b)$$

$$\mathit{zipStream} :: (\mathrm{Stream}\ a, \mathrm{Stream}\ b) \to \mathrm{Stream}\ (a, b)$$
$$\mathit{zipStream}^{-1} :: \mathrm{Stream}\ (a, b) \to (\mathrm{Stream}\ a, \mathrm{Stream}\ b)$$

$$\textbf{newtype}\ \mathrm{StreamProc}\ b\ c = \mathrm{SP}\ (\mathrm{Stream}\ b \to \mathrm{Stream}\ c)$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{Arrow}\ \mathrm{StreamProc}\ \textbf{where}\\
&\qquad \mathrm{pure}\ f = \mathrm{SP}\ (\mathrm{fmap}\ f)\\
&\qquad \mathrm{SP}\ f \ggg \mathrm{SP}\ g = \mathrm{SP}\ (g \cdot f)\\
&\qquad \mathrm{first}\ (\mathrm{SP}\ f) = \mathrm{SP}\ (\mathit{zipStream} \cdot (f \times \mathrm{id}) \cdot \mathit{zipStream}^{-1})
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{ArrowLoop}\ \mathrm{StreamProc}\ \textbf{where}\\
&\qquad \mathrm{loop}\ (\mathrm{SP}\ f) = \mathrm{SP}\ (\mathrm{loop}\ (\mathit{zipStream}^{-1} \cdot f \cdot \mathit{zipStream}))
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ \mathit{ArrowCircuit}\ \mathrm{StreamProc}\ \textbf{where}\\
&\qquad \mathrm{delay}\ b = \mathrm{SP}\ (\mathrm{Cons}\ b)
\end{aligned}$$

$$\textbf{newtype}\ \mathit{Auto}\ b\ c = A\ (b \to (c, \mathit{Auto}\ b\ c))$$

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

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

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

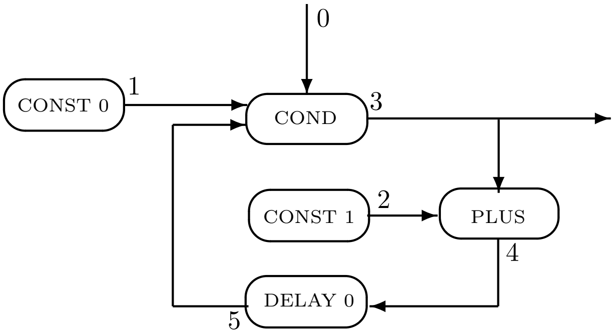
More interpretations: Arrow transformers

It $\leadsto$ is an arrow, so are the following:

$b \leadsto \mathit{Either}\ ex\ c$	exceptions
$b \leadsto (M, c)$	writer (M a monoid, e.g. String)
$(s, b) \leadsto c$	reader
$(s, b) \leadsto (s, c)$	state transformer
$(s \to b) \leadsto (s \to c)$	map transformer
$\mathit{Stream}\ b \leadsto \mathit{Stream}\ c$	stream transformers
$F\ (b \leadsto c)$	static properties (appropriate F)
$\nu\, x.\ (b \leadsto (c, x))$	simple automata

(sometimes with restrictions on $\leadsto$)

Netlists



$$
\begin{aligned}
[&(1, \mathsf{Const}\ 0),\\
&(2, \mathsf{Const}\ 1),\\
&(3, \mathsf{Cond}\ 0\ 1\ 5),\\
&(4, \mathsf{Plus}\ 3\ 2),\\
&(5, \mathsf{Delay}\ 0\ 4)]
\end{aligned}
$$

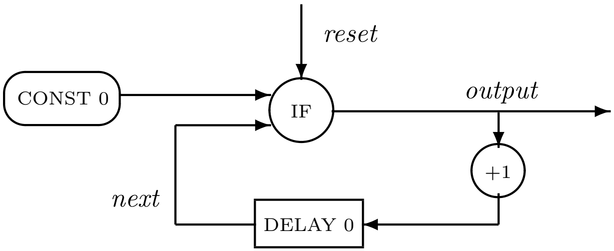
Abstracting the value types

$$
\begin{array}{l}
\textbf{class}\ \mathit{ArrowLoop}\ a \Rightarrow \mathit{ArrowBoolCircuit}\ a\ b\ \textbf{where} \\
\qquad \mathit{true} :: a\ ()\ b \\
\qquad \mathit{false} :: a\ ()\ b
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{ArrowLoop}\ a \Rightarrow \mathit{ArrowIntCircuit}\ a\ i\ \textbf{where} \\
\qquad \mathit{delay} :: \mathit{Int} \to a\ i\ i \\
\qquad \mathit{plus} :: a\ (i, i)\ i \\
\qquad \mathit{constant} :: \mathit{Int} \to a\ ()\ i
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ (\mathit{ArrowBoolCircuit}\ a\ b, \mathit{ArrowIntCircuit}\ a\ i) \Rightarrow \\
\qquad\qquad\qquad \mathit{ArrowCircuit}\ a\ b\ i\ \textbf{where} \\
\qquad \mathit{cond} :: a\ (b, i, i)\ i
\end{array}
$$

The counter again



$$
\begin{aligned}
&\mathrm{counter} :: \mathrm{ArrowCircuit\ a\ b\ i} \Rightarrow \mathrm{a\ b\ i} \\
&\mathrm{counter} = \textbf{proc}\ \mathrm{reset} \rightarrow \textbf{do} \\
&\qquad \mathrm{zero} \leftarrow \mathrm{constant}\ 0 \prec () \\
&\qquad \mathrm{one} \leftarrow \mathrm{constant}\ 1 \prec () \\
&\qquad \textbf{rec}\ \mathrm{output} \leftarrow \mathrm{cond} \prec (\mathrm{reset}, \mathrm{zero}, \mathrm{next}) \\
&\qquad\quad \mathrm{incr} \leftarrow \mathrm{plus} \prec (\mathrm{output}, \mathrm{one}) \\
&\qquad\quad \mathrm{next} \leftarrow \mathrm{delay}\ 0 \prec \mathrm{incr} \\
&\qquad \mathrm{idA} \prec \mathrm{output}
\end{aligned}
$$

A netlist interpretation

$$\textbf{type}\ \mathit{Label} = \mathit{Int}$$

$$\begin{aligned}\textbf{data}\ \mathit{Node} &= \mathit{Delay}\ \mathit{Int}\ \mathit{Label} \\ &\mid \mathit{Cond}\ \mathit{Label}\ \mathit{Label}\ \mathit{Label} \\ &\mid \ldots\end{aligned}$$

$$\textbf{type}\ \mathit{NetList} = [(\mathit{Label}, \mathit{Node})]$$

$$\boxed{\textbf{data}\ \mathit{NLArrow}\ b\ c = \mathit{NL}\ ((\mathit{Label}, b) \to (\mathit{Label}, (\mathit{NetList}, c)))}$$

$$\textbf{instance}\ \mathit{ArrowCircuit}\ \mathit{NLArrow}\ \mathit{Label}\ \mathit{Label}$$

Recall that

$$\mathit{counter} :: \mathit{ArrowCircuit}\ a\ b\ i \Rightarrow a\ b\ i$$

so pass Labels through the wires.

Conclusion

- Arrows are a useful generalization of monads
- Arrow notation makes arrows more convenient, yielding embedded domain-specific languages:
  - synchronous circuits (dataflow)
  - Functional Reactive Programming (Elliott, Hudak)
  - data-parallel algorithms
  - self-optimizing parsers (Swierstra)
- For more details, see `http://www.haskell.org/arrows/`