A Brief Introduction to Functional Reactive Programming and Yampa

FoP Away Day 17 January 2007

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

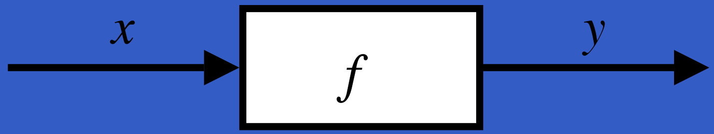
Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
- ***Yampa:*** An FRP implementation in the form of a Haskell ***combinator library***, a.k.a. ***Domain-Specific Embedded Language*** (DSEL).

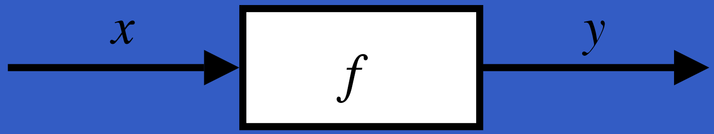
Signal functions

Key concept: ***functions on signals***.



Signal functions

Key concept: ***functions on signals***.

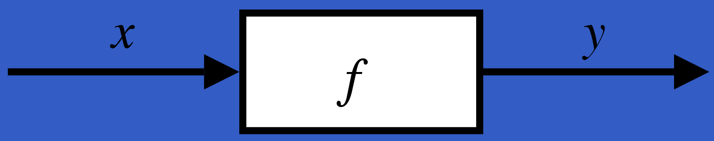


Intuition:

$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$
x :: Signal T1
y :: Signal T2
$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$
f :: SF T1 T2

Signal functions

Key concept: ***functions on signals***.



Intuition:

$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$
x :: `Signal T1`
y :: `Signal T2`
$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$
f :: `SF T1 T2`

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

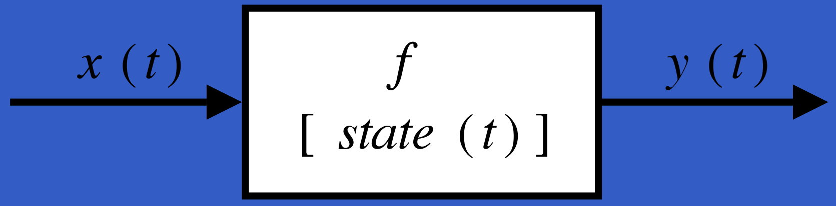
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.

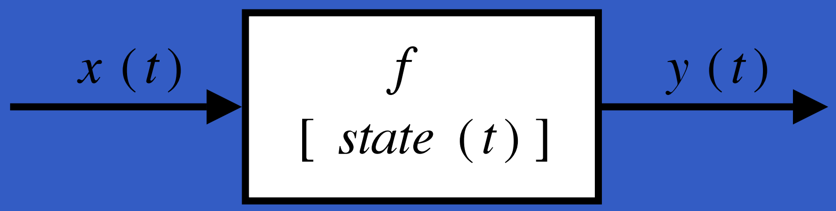


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.



$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

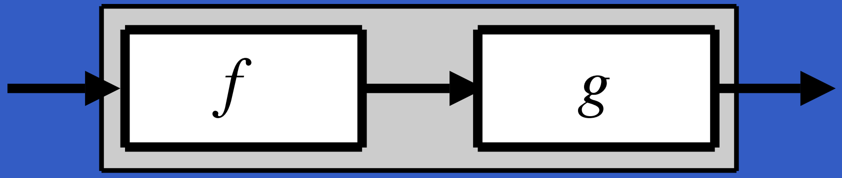
Programming with signal functions

In Yampa, systems are described by combining signal functions (forming new signal functions).

Programming with signal functions

In Yampa, systems are described by combining signal functions (forming new signal functions).

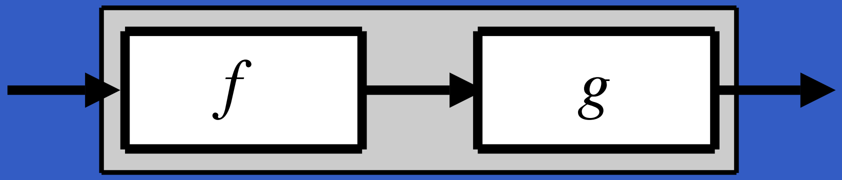
For example, serial composition:



Programming with signal functions

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

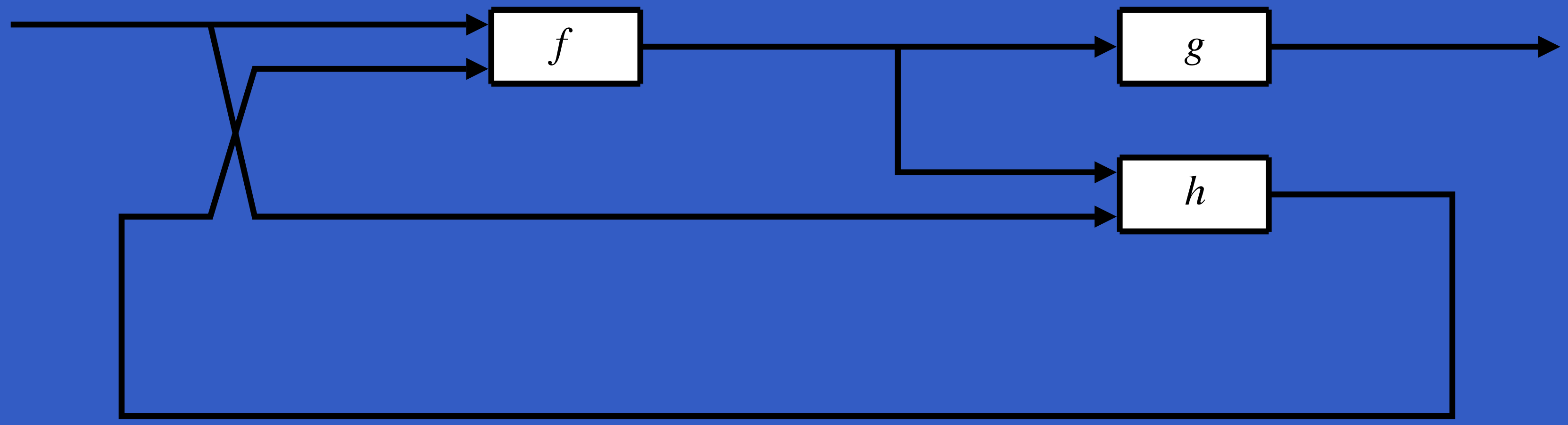


A *combinator* can be defined that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

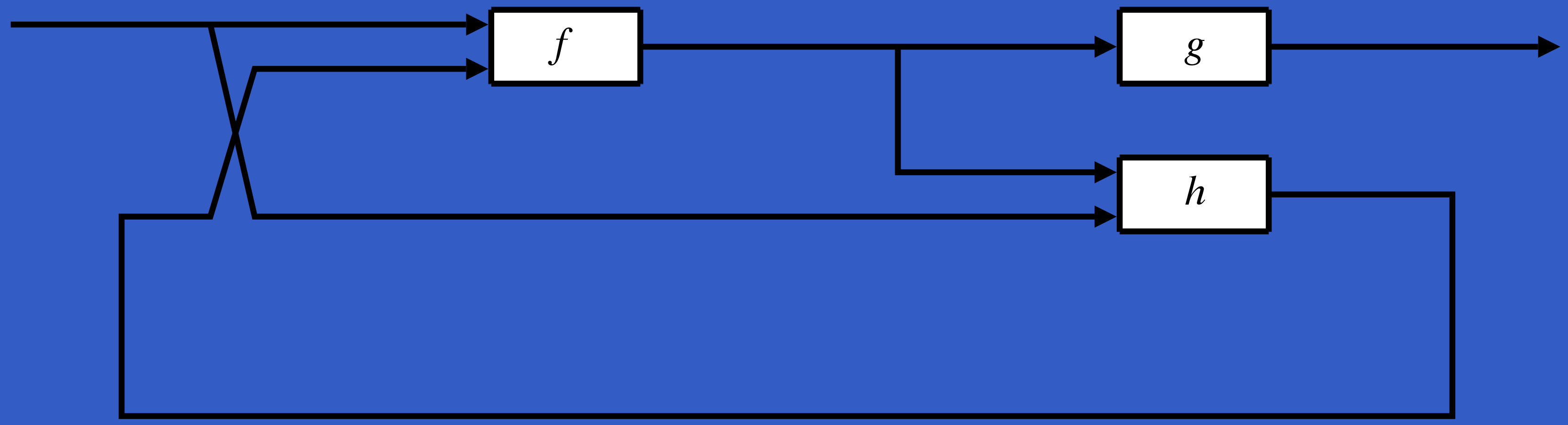
Programming with signal functions (2)

What about larger networks?
How many combinators are needed?



Programming with signal functions (2)

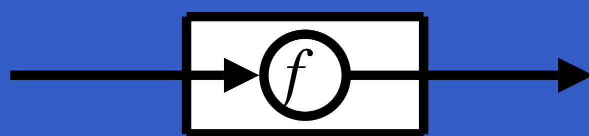
What about larger networks?
How many combinators are needed?



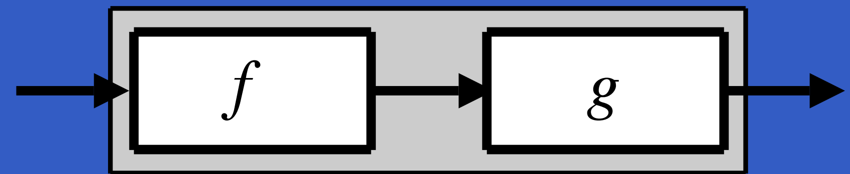
John Hughes's ***Arrow*** framework provides a good answer!

The Arrow framework (1)

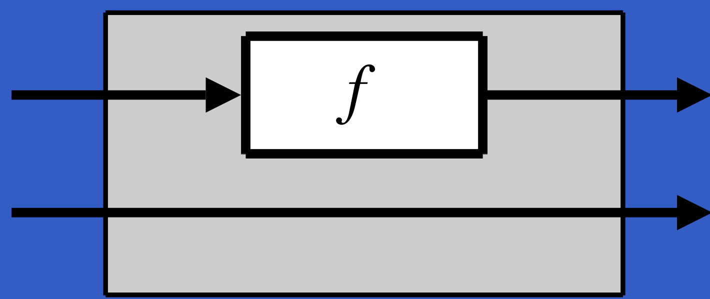
These diagrams convey the general idea:



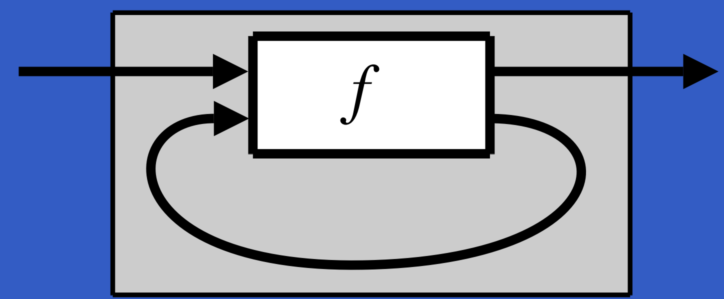
$arr\ f$



$\ggg$



$first\ f$



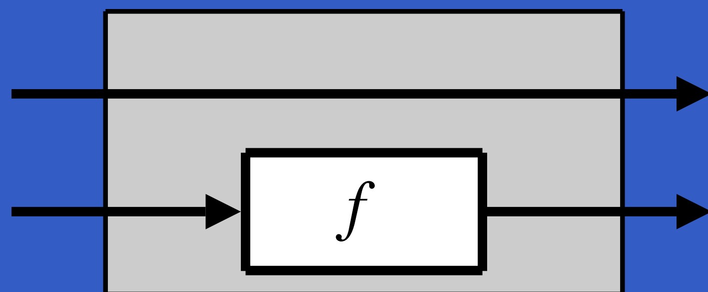
$loop\ f$

$$first :: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)$$

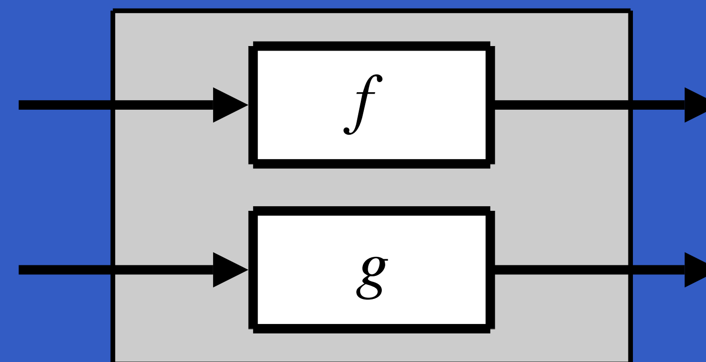
$$loop :: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b$$

The Arrow framework (2)

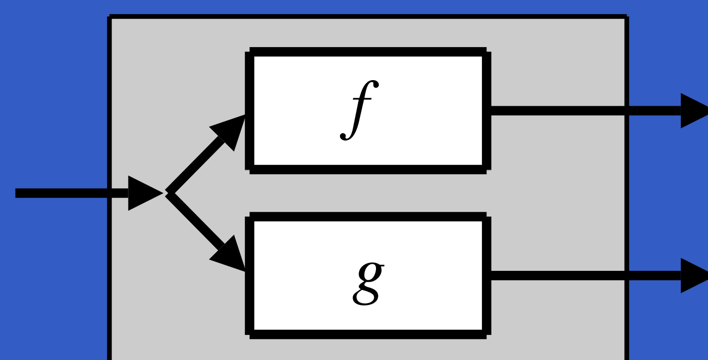
Some derived combinators:



$second\, f$

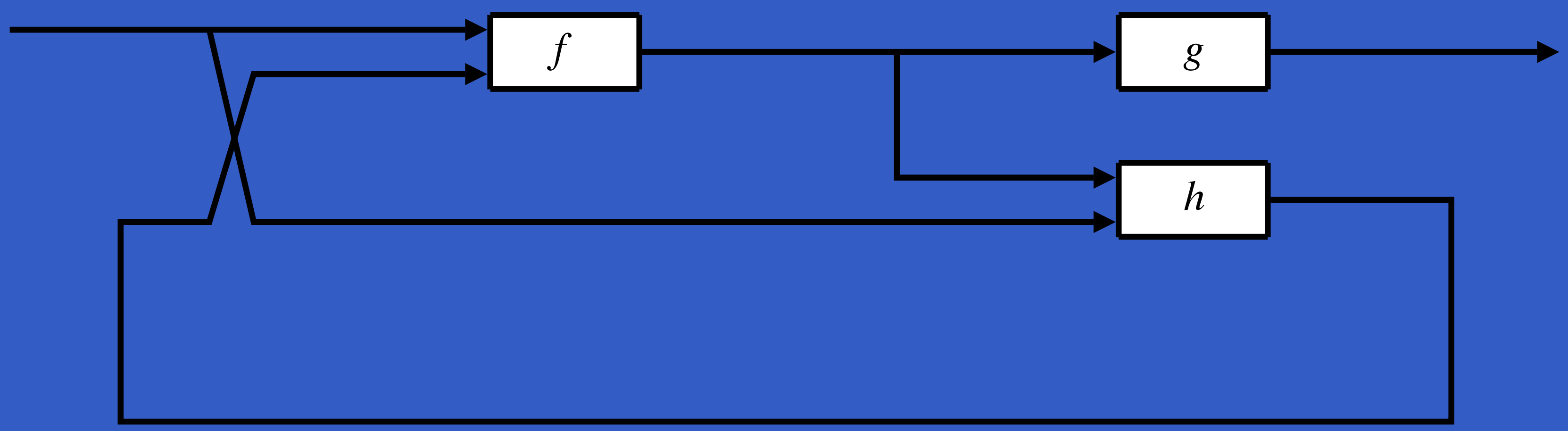


$f \ast\!\ast\!\ast\, g$

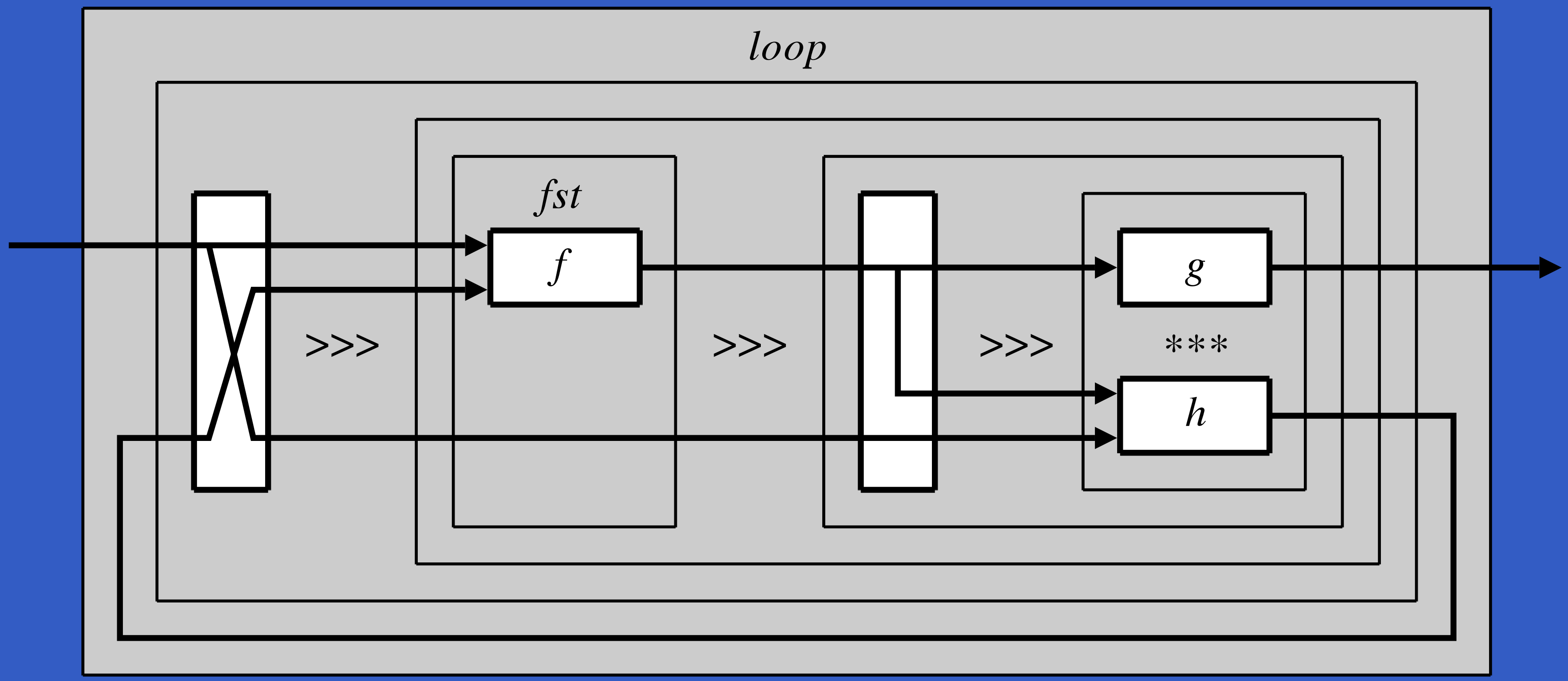


$f \,\&\&\&\, g$

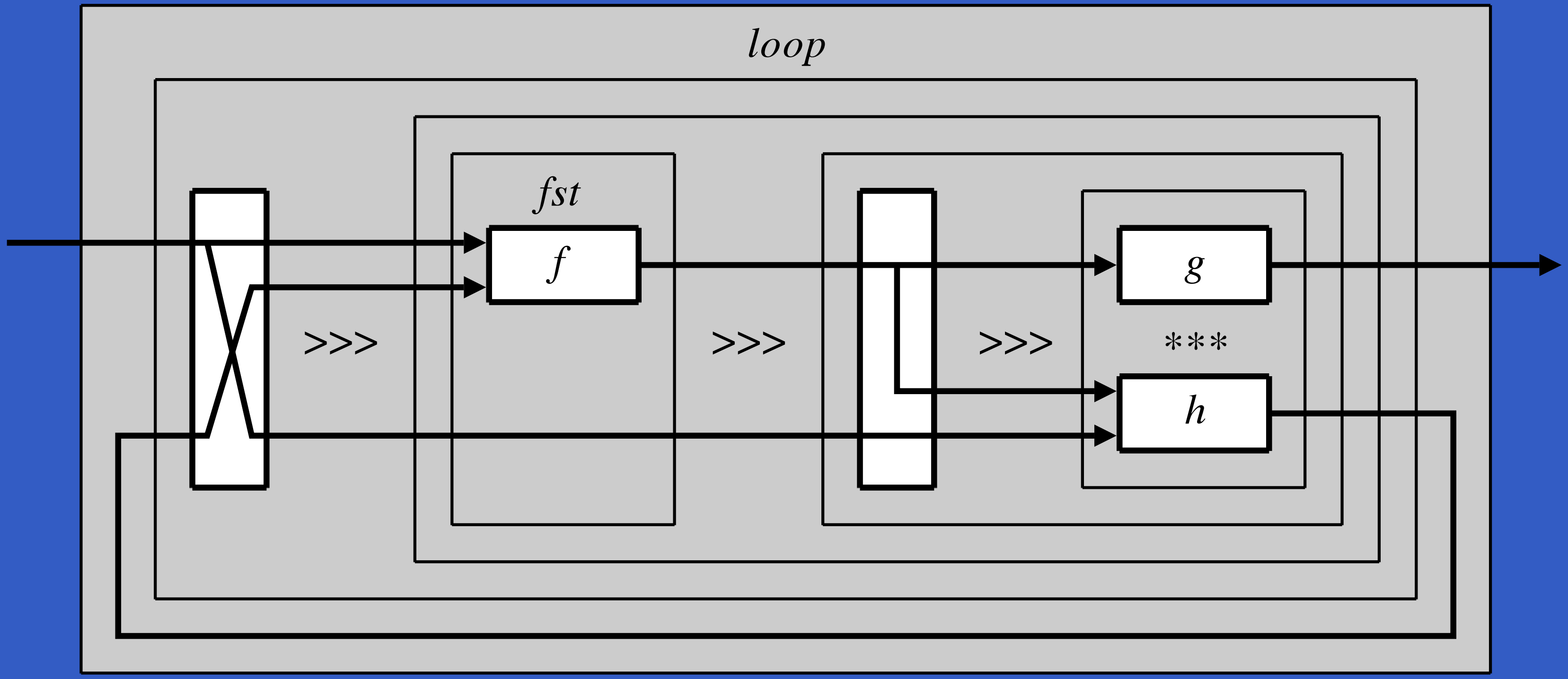
Example: Constructing a network



Example: Constructing a network

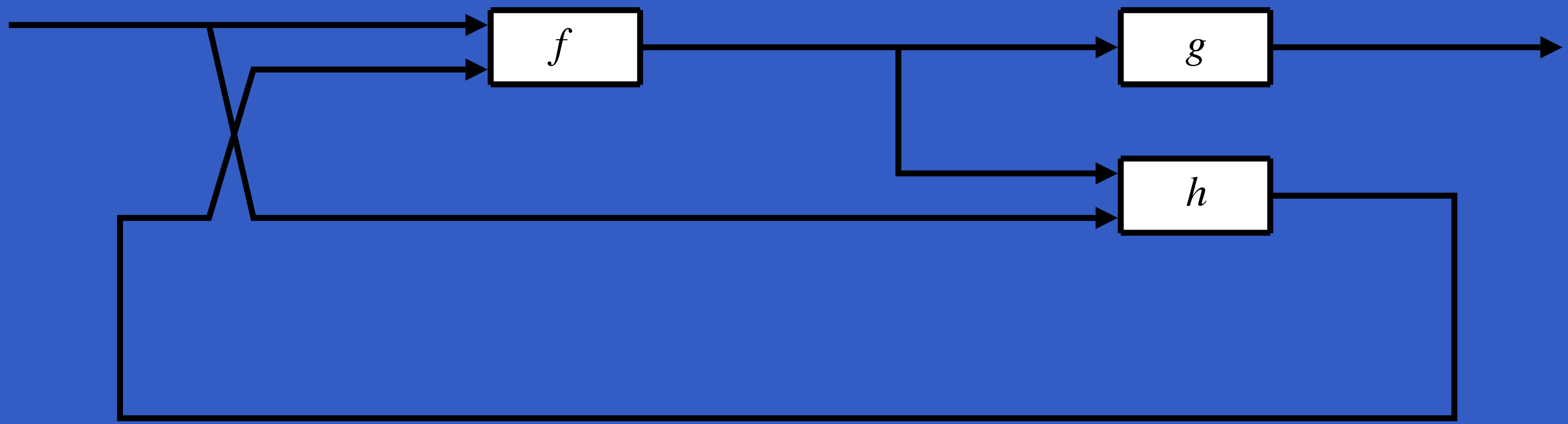


Example: Constructing a network

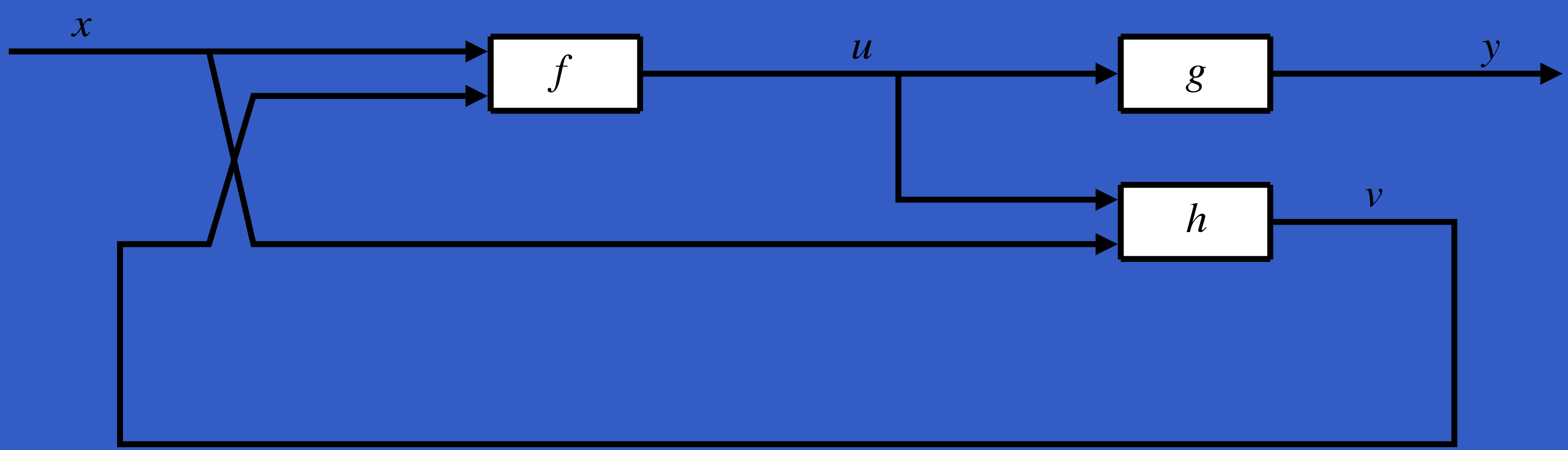


$$
\begin{aligned}
&loop\ (arr\ (\lambda(x, y) \rightarrow ((x, y), x)) \\
&\qquad \ggg (fst\ f \\
&\qquad\qquad \ggg (arr\ (\lambda(x, y) \rightarrow (x, (x, y))) \ggg (g *\!\!*\!\!* h))))
\end{aligned}
$$

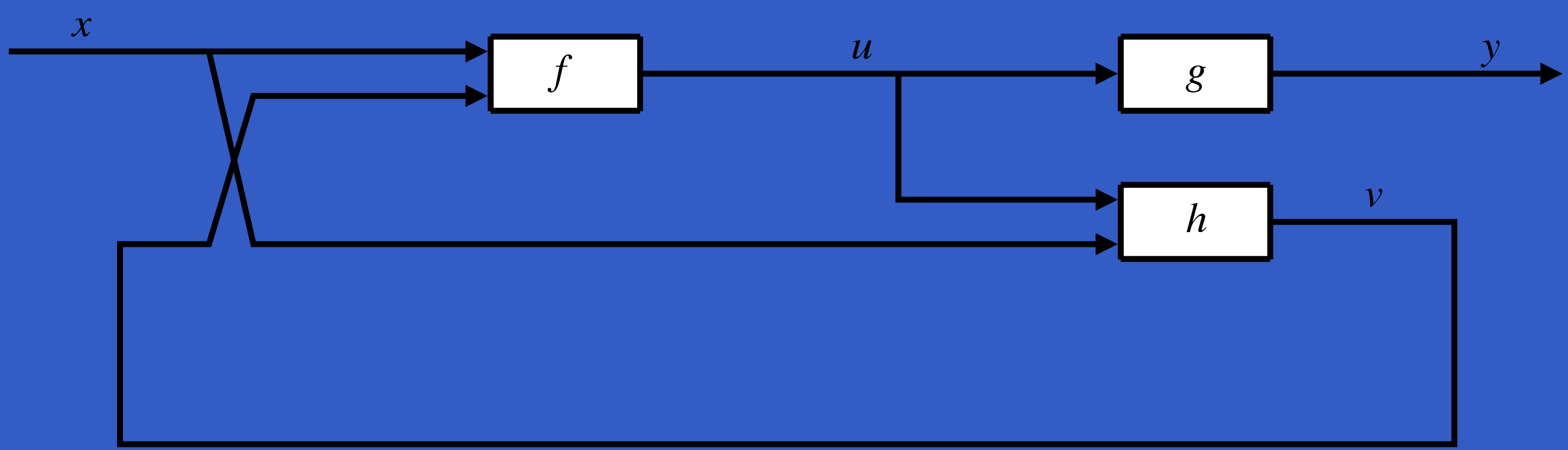
The Arrow notation



The Arrow notation



The Arrow notation



$$
\begin{array}{l}
\textbf{proc}\ x \rightarrow \textbf{do} \\
\quad \textbf{rec} \\
\qquad u \leftarrow f \prec (x, v) \\
\qquad y \leftarrow g \prec u \\
\qquad v \leftarrow h \prec (u, x) \\
\quad returnA \prec y
\end{array}
$$

How does it work?

- Essentially:

$$\textbf{newtype}\ SF\ a\ b =$$
$$\quad SF\ (DeltaTime \rightarrow a \rightarrow (SF\ a\ b, b))$$

How does it work?

- Essentially:

$$\begin{aligned}&\textbf{newtype}\ \mathit{SF}\ a\ b =\\&\quad \mathit{SF}\ (\mathit{DeltaTime} \rightarrow a \rightarrow (\mathit{SF}\ a\ b, b))\end{aligned}$$

- A top-level loop, ***reactimate***, drives the computation.

How does it work?

- Essentially:

$$\begin{aligned}&\mathbf{newtype}\ SF\ a\ b =\\&\quad SF\ (DeltaTime \rightarrow a \rightarrow (SF\ a\ b, b))\end{aligned}$$

- A top-level loop, ***reactimate***, drives the computation.

Note that the system representation in principle is reconstructed at every time step.

Related languages and paradigms

FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.

Related languages and paradigms

FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

What makes Yampa interesting?

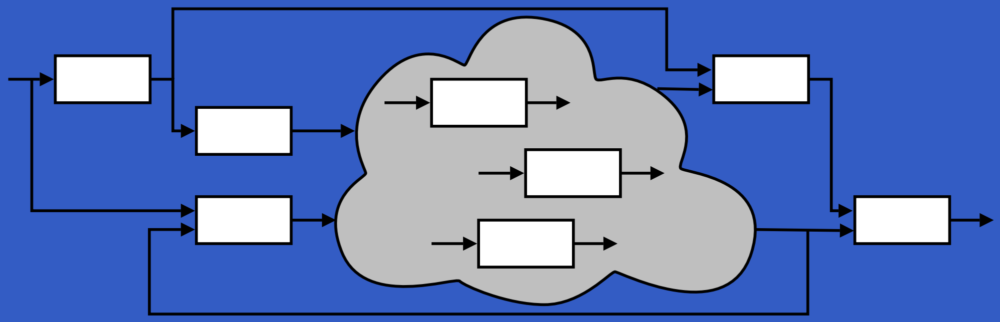
- First class reactive components (signal functions).

What makes Yampa interesting?

- First class reactive components (signal functions).
- Supports hybrid (mixed continuous and discrete time) systems: option type $Event$ represents discrete-time signals.

What makes Yampa interesting?

- First class reactive components (signal functions).
- Supports hybrid (mixed continuous and discrete time) systems: option type $Event$ represents discrete-time signals.
- Supports dynamic system structure through ***switching combinators***:



Example: Space Invaders