

A Functional Approach to Programming and Modelling of Reactive, Hybrid Systems

Henrik Nilsson

University of Nottingham, UK

-
-
-

Functional Reactive Programming

Functional Reactive Programming (FRP):

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.
- Supports *Hybrid Systems*; i.e., continuous time and discrete time.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.
- Supports *Hybrid Systems*; i.e., continuous time and discrete time.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.
- Supports *Hybrid Systems*; i.e., continuous time and discrete time.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.
- Supports *Hybrid Systems*; i.e., continuous time and discrete time.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Yampa: an instance of FRP embedded in Haskell.

Related languages

FRP related to:

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

Distinct features of FRP:

- First class reactive components.
- Highly structurally dynamic systems can be described declaratively.

FRP applications

Some domains where FRP has been used:

- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)
- Sound Synthesis (Giorgidze, Nilsson)

Functional Hybrid Modelling

- The goal of **Functional Hybrid Modelling** (FHM) is to combine an FRP-approach with non-causal modelling yielding:

Functional Hybrid Modelling

- The goal of **Functional Hybrid Modelling** (FHM) is to combine an FRP-approach with non-causal modelling yielding:
 - a powerful, fully-declarative, non-causal modelling language supporting highly structurally dynamic systems;

Functional Hybrid Modelling

- The goal of **Functional Hybrid Modelling** (FHM) is to combine an FRP-approach with non-causal modelling yielding:
 - a powerful, fully-declarative, non-causal modelling language supporting highly structurally dynamic systems;
 - a semantic framework for studying modelling and simulation languages supporting structural dynamism.

Functional Hybrid Modelling

- The goal of **Functional Hybrid Modelling** (FHM) is to combine an FRP-approach with non-causal modelling yielding:
 - a powerful, fully-declarative, non-causal modelling language supporting highly structurally dynamic systems;
 - a semantic framework for studying modelling and simulation languages supporting structural dynamism.

Work still in early stages.

The Rest of the Talk

- Introduction to Yampa.
- A Yampa animation example in some detail.
- Non-causal modelling and applying FRP-like ideas to that setting.

Yampa

The most recent Yale FRP implementation is called *Yampa*:

- Embedding in Haskell; i.e. a Haskell library.
- Arrows used as the basic structuring framework.
- Advanced switching constructs allows for highly dynamic system structure.

-
-
-

Yampa?

Yampa?

*Y*et
*A*nother
*M*ostly
*P*ointless
*A*cronym

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

Yampa?

*Y*et
*A*nother
*M*ostly
*P*ointless
*A*cronym

???

No ...

Yampa?

Yampa is a river . . .



Yampa?

... with long calmly flowing sections ...



Yampa?

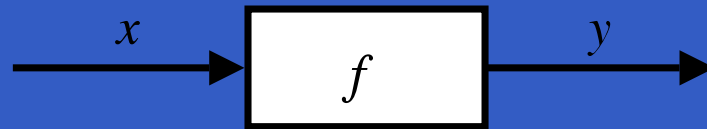
... and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

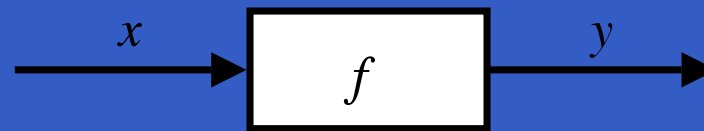
Signal functions

Key concept: *functions on signals* (first class).



Signal functions

Key concept: *functions on signals* (first class).



Intuition:

Signal $\alpha \approx \text{Time} \rightarrow \alpha$

$x :: \text{Signal } T1$

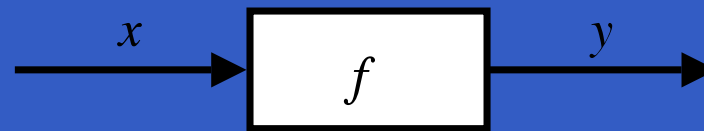
$y :: \text{Signal } T2$

SF $\alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$

$f :: \text{SF } T1 \ T2$

Signal functions

Key concept: *functions on signals* (first class).



Intuition:

Signal $\alpha \approx \text{Time} \rightarrow \alpha$

$x :: \text{Signal } T1$

$y :: \text{Signal } T2$

SF $\alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$

$f :: \text{SF } T1 \ T2$

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

-
-
-

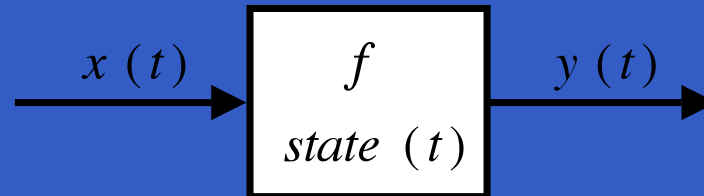
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

Signal functions can encapsulate *state*.

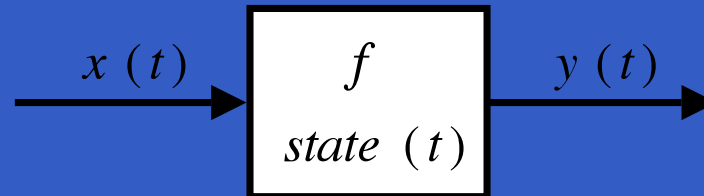


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

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 $x(t)$

Integral is an example of a stateful signal function.

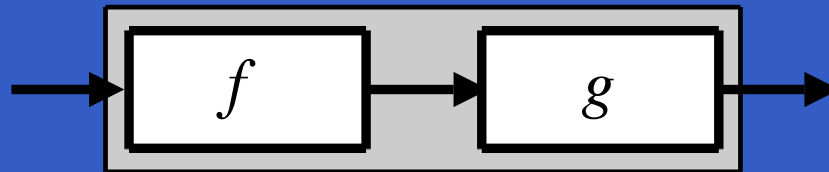
Programming with signal functions

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).

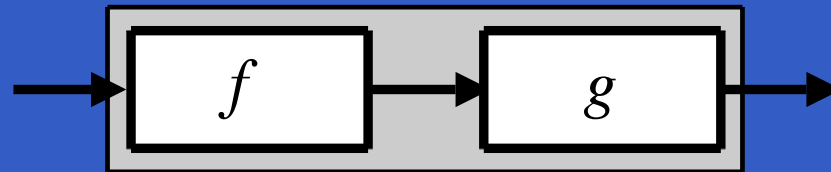
For example, serial composition:



Programming with signal functions

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

For example, serial composition:



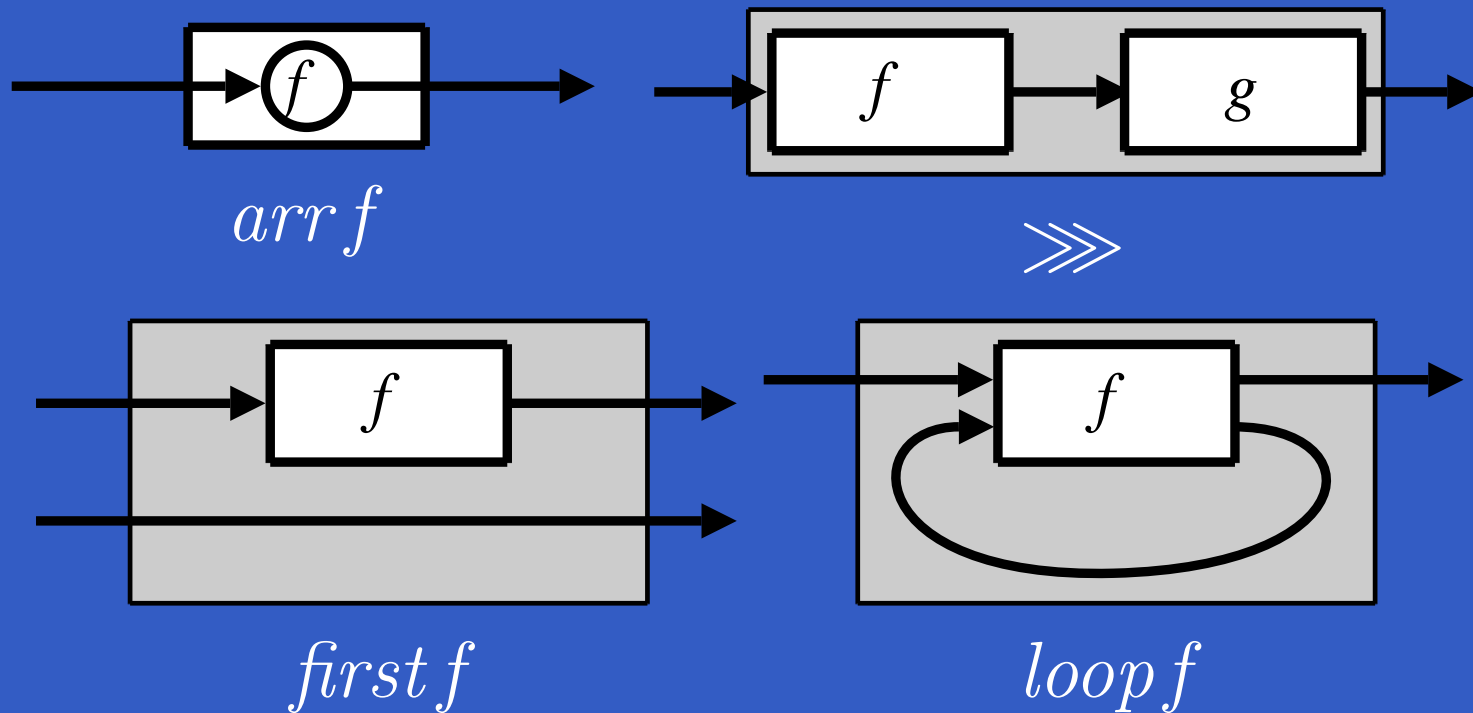
A *combinator* can be defined that captures this:

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

Note: plain function operating on first-class signal function.

The Arrow framework (1)

These diagrams convey the general idea:

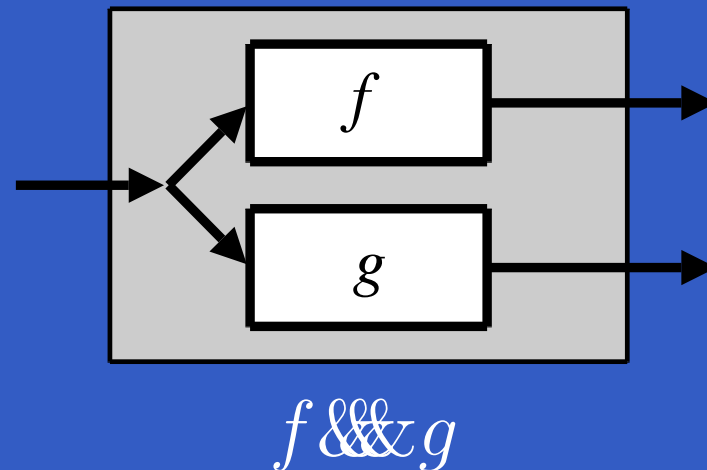
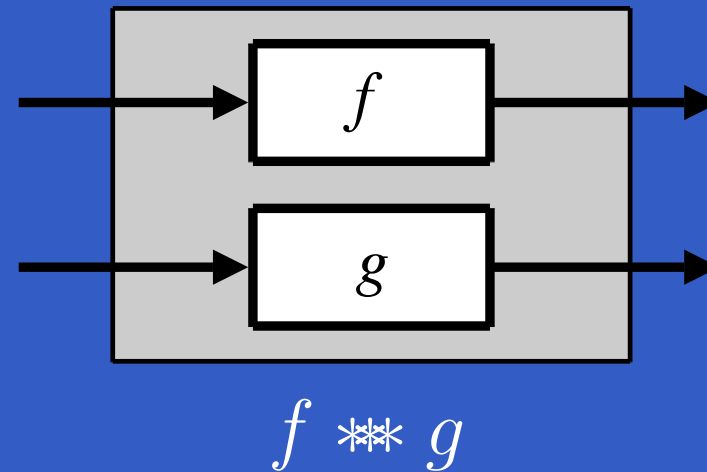
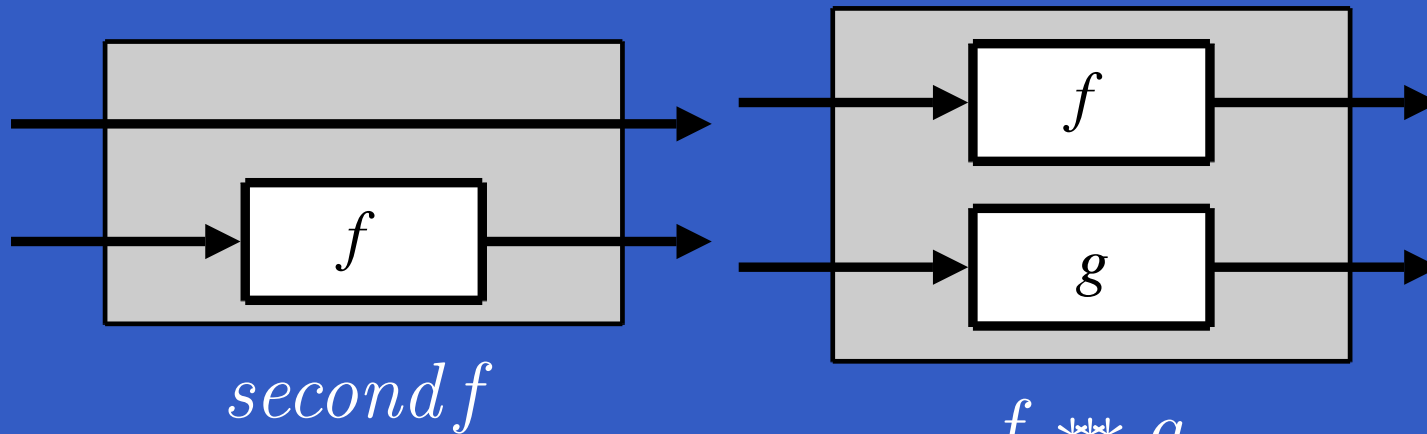


$first :: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c)$

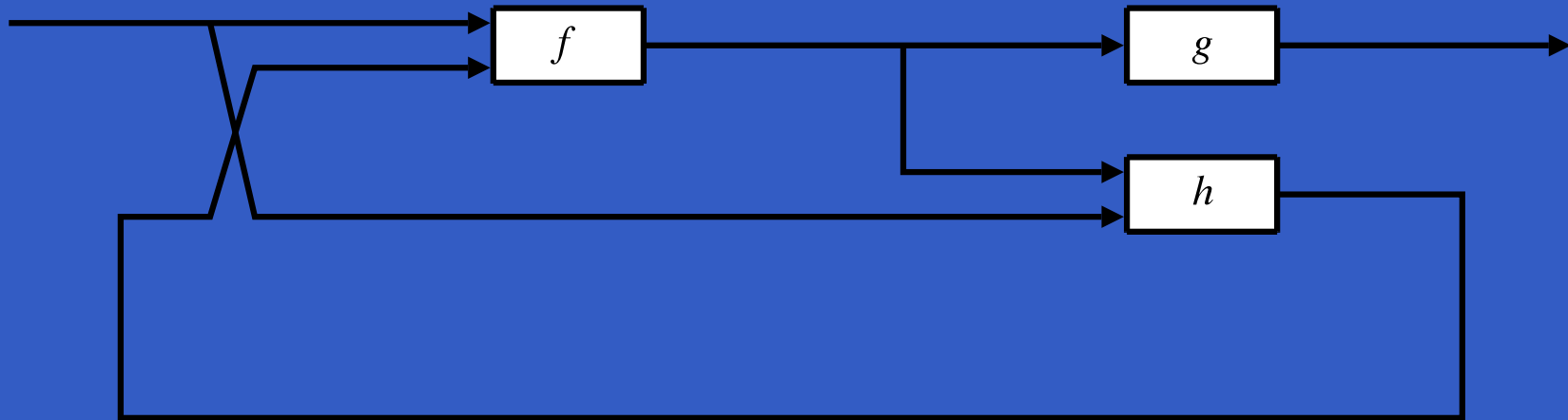
$loop :: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b$

The Arrow framework (2)

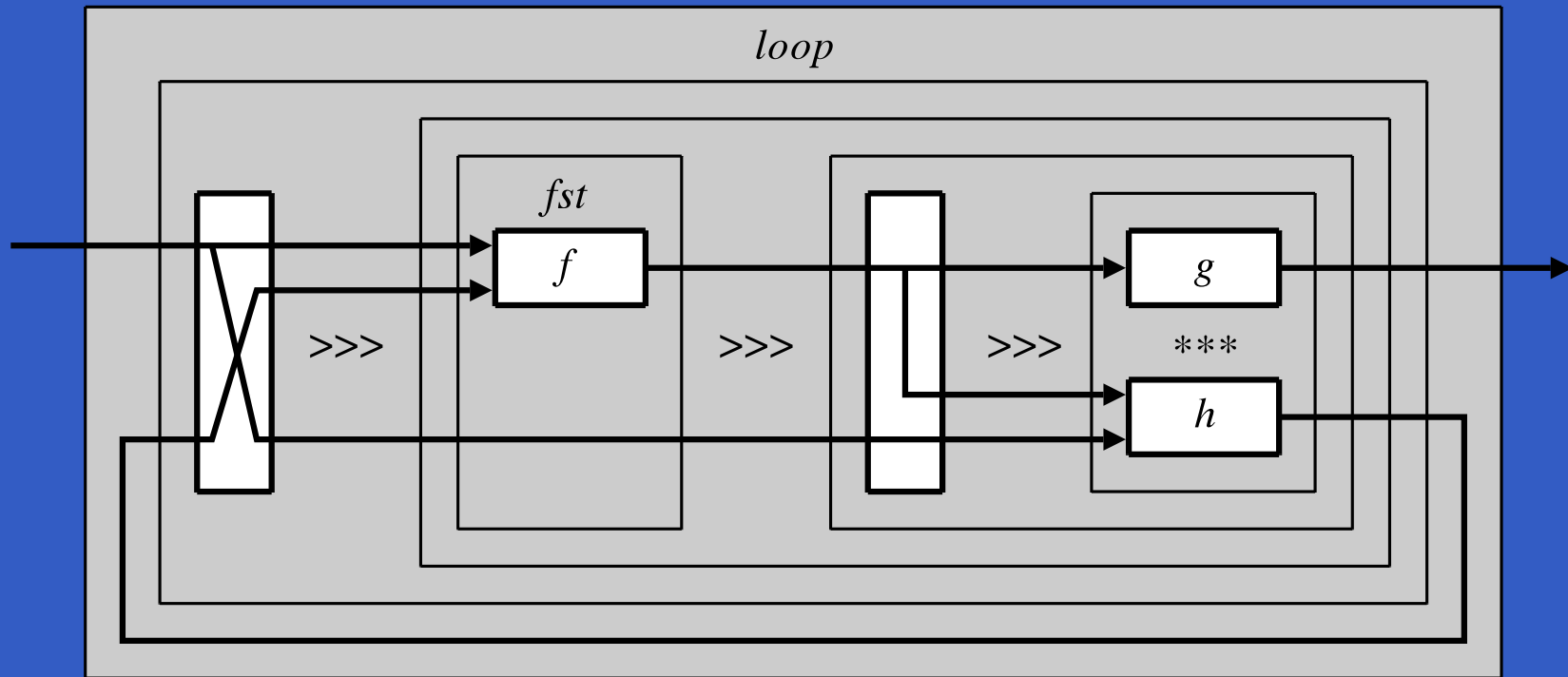
Some derived combinators:



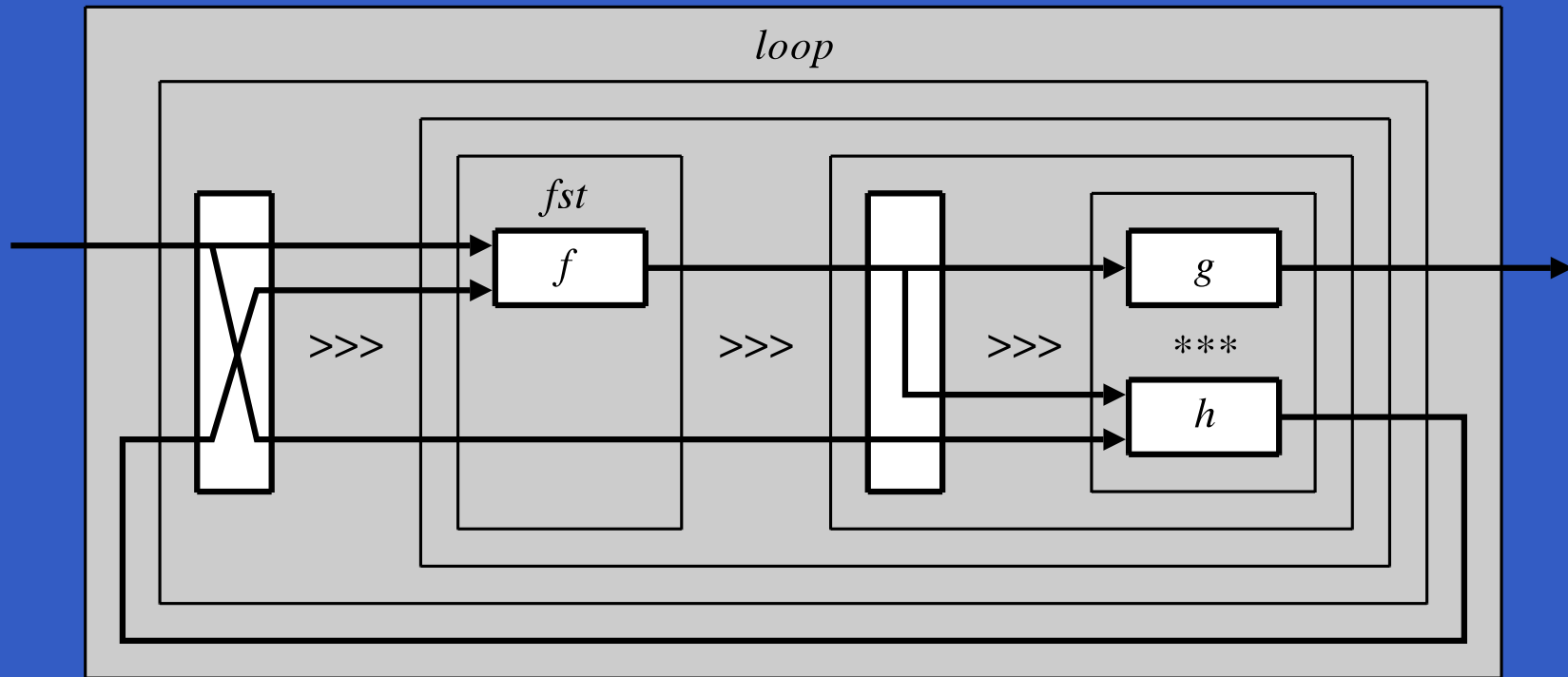
Example: Constructing a network



Example: Constructing a network



Example: Constructing a network

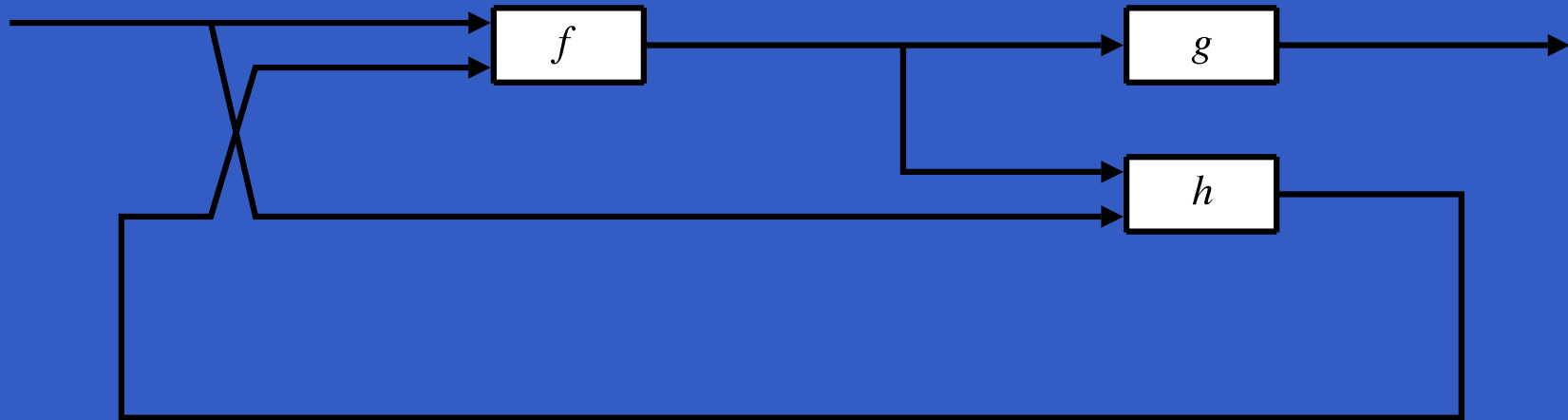


$loop (arr (\lambda(x, y) \rightarrow ((x, y), x)))$

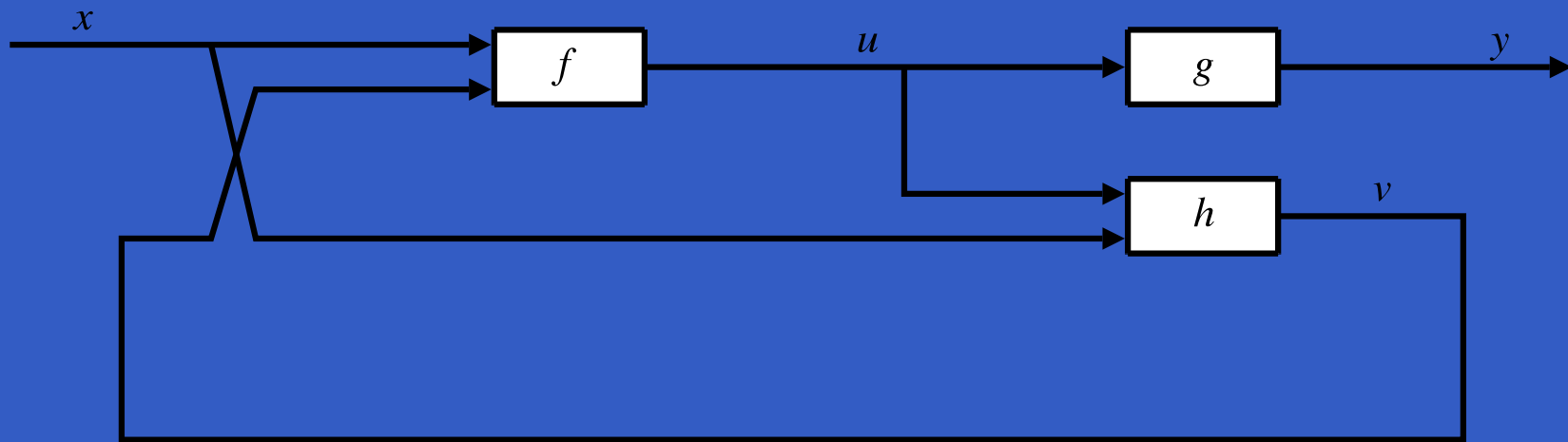
$\ggg (fst f$

$\ggg (arr (\lambda(x, y) \rightarrow (x, (x, y))) \ggg (g ** h))))$

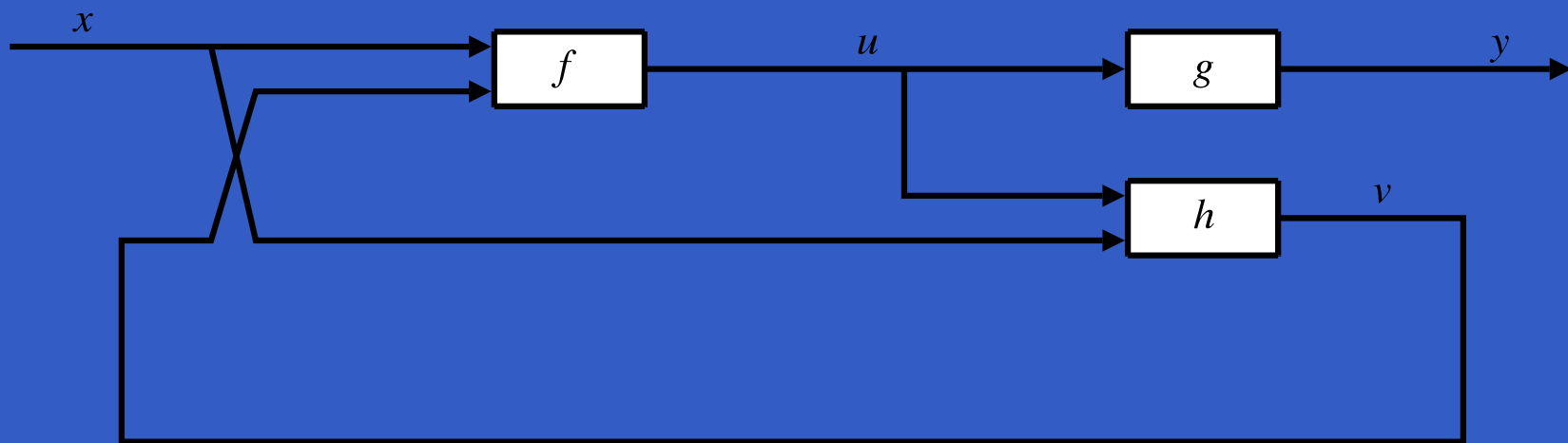
The Arrow notation



The Arrow notation



The Arrow notation



`proc $x \rightarrow$ do`

`rec`

$$u \leftarrow f \multimap (x, v)$$

$$y \leftarrow g \multimap u$$

$$v \leftarrow h \multimap (u, x)$$

`return $A \multimap y$`

Switching (1)

Q: How and when do signal functions “start”?

Switching (1)

Q: How and when do signal functions “start”?

- A:
- **Switchers** “apply” a signal functions to its input signal at some point in time.
 - This creates a “running” signal function *instance*, which often replaces the previously running instance.

Switching (1)

Q: How and when do signal functions “start”?

A: • **Switchers** “apply” a signal functions to its input signal at some point in time.

- This creates a “running” signal function **instance**, which often replaces the previously running instance.

≫≫, *first* etc. are **spatial** combinators.
Switchers are **temporal** combinators allowing systems with **varying structure** to be described.

Switching (2)

Option type *Event* represents discrete-time signals:

$$\text{data } Event\ a = NoEvent \mid Event\ a$$

Switching (2)

Option type *Event* represents discrete-time signals:

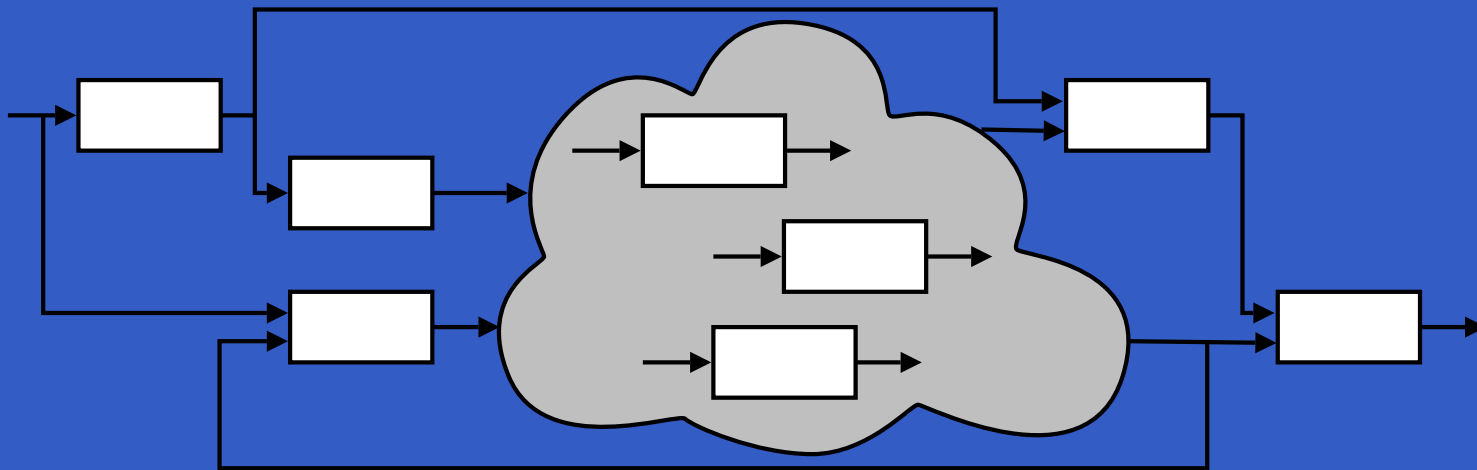
`data Event a = NoEvent | Event a`

A basic switching combinator:

`switch :: SF a (b, Event c) → (c → SF a b)
→ SF a b`

Switching (3)

Advanced switching combinators supports even more radical changes to the system structure:



Example: Space Invaders



Describing the alien behavior (1)

```
type Object = SF ObjInput ObjOutput
```

```
alien :: RandomGen g =>
```

```
  g → Position2 → Velocity → Object
```

```
alien g p0 vyd = proc oi → do
```

```
  rec
```

```
    -- Pick a desired horizontal position
```

```
    rx ← noiseR (xMin, xMax) g ↯ ()
```

```
    simpl ← occasionally g 5 () ↯ ()
```

```
    xd ← hold (point2X p0) ↯ simpl 'tag' rx
```

```
    ...
```

Describing the alien behavior (2)

...

-- Controller

```
let  $axd = 5 * (xd - point2X\ p)$   
     $- 3 * (vector2X\ v)$   
 $ayd = 20 * (vyd - (vector2Y\ v))$   
 $ad = vector2\ axd\ ayd$   
 $h = vector2Theta\ ad$ 
```

...

Describing the alien behavior (3)

...

-- Physics

let a = vector2Polar (min alienAccMax (vector2Rho ad)) h

vp ← iPre v0 $\prec v$

ffi ← forceField $\prec (p, vp)$

v ← arr (v0 ↑+↑) ≪≪ impulseIntegral $\prec (gravity ↑+↑ a, ffi)$

p ← arr (p0 ·+↑) ≪≪ integral $\prec v$

...

Describing the alien behavior (4)

...

-- Shields

$sl \leftarrow shield \multimap oiHit oi$

$die \leftarrow edge \multimap sl \leq 0$

$returnA \multimap ObjOutput\{$

$ooObsObjState = oosAlien p h v,$

$ooKillReq = die,$

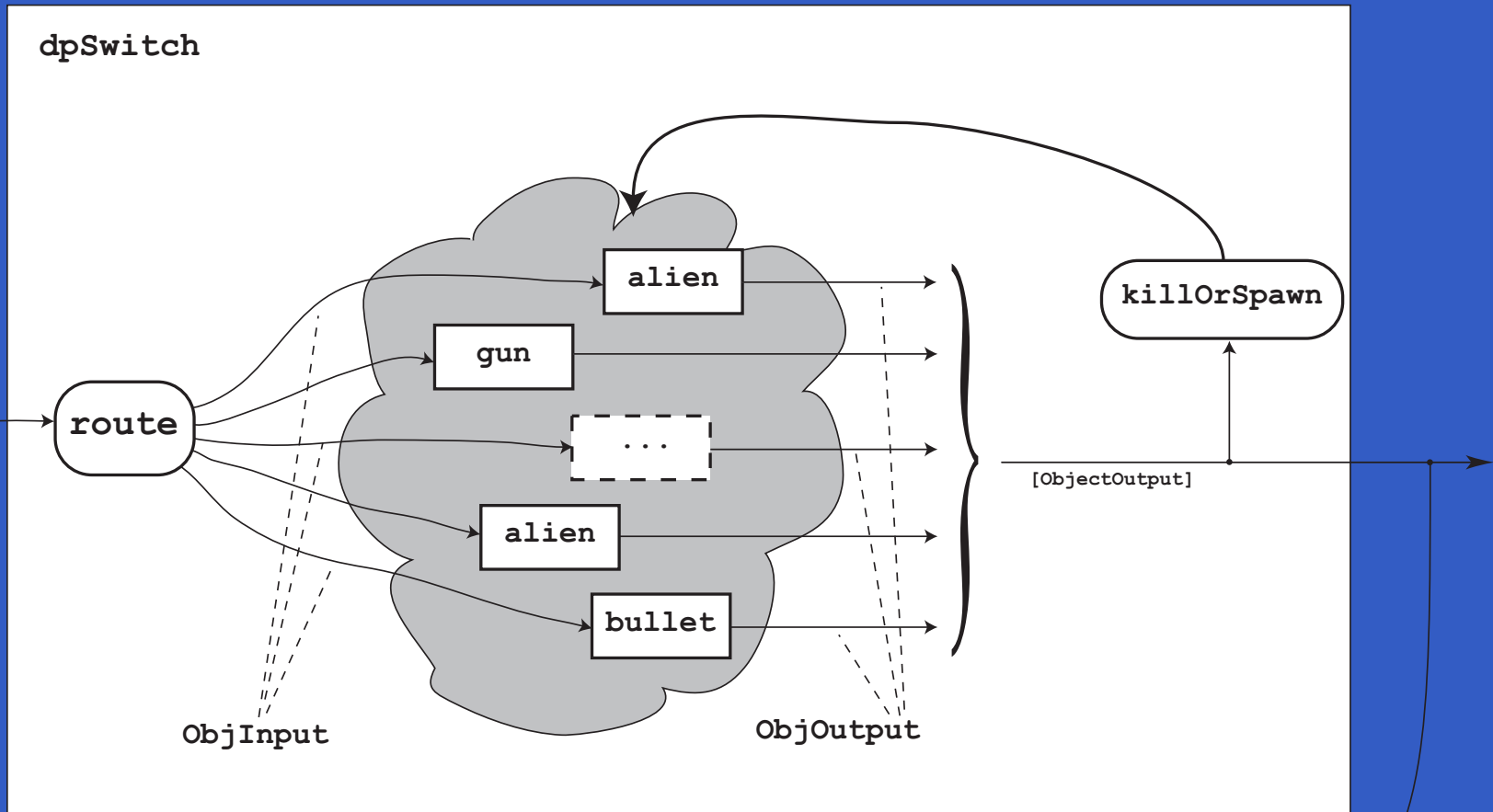
$ooSpawnReq = noEvent$

$\}$

where

$v0 = zeroVector$

Overall game structure



Causal vs. non-causal modeling (1)

Causal or **block-oriented** modeling: model is ODE in **explicit** form:

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}, \mathbf{u}, t)$$

Causality, i.e. cause-effect relationship, given by the modeler.

Causal modeling is the dominating modeling paradigm. Languages: Simulink, Yampa, ...

Causal vs. non-causal modeling (2)

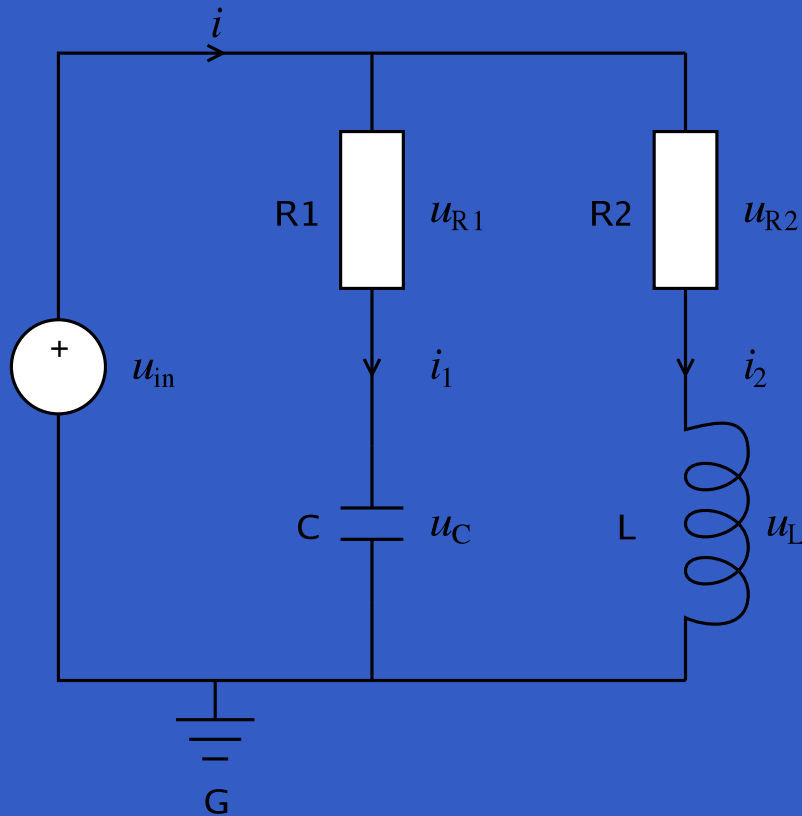
Non-causal or “***object-oriented***” modeling:
model is DAE in implicit form:

$$\mathbf{f}(\mathbf{x}, \mathbf{x}', \mathbf{w}, \mathbf{u}, t) = \mathbf{0}$$

Causality inferred by simulation tool from usage context.

Non-causal modeling is a fairly recent development. Languages: Dymola, Modelica, ...

Causal modeling: example (1)



$$u_{R_2} = R_2 i_2$$

$$u_L = u_{in} - u_{R_2}$$

$$i_2' = \frac{u_L}{L}$$

$$u_{R_1} = u_{in} - u_C$$

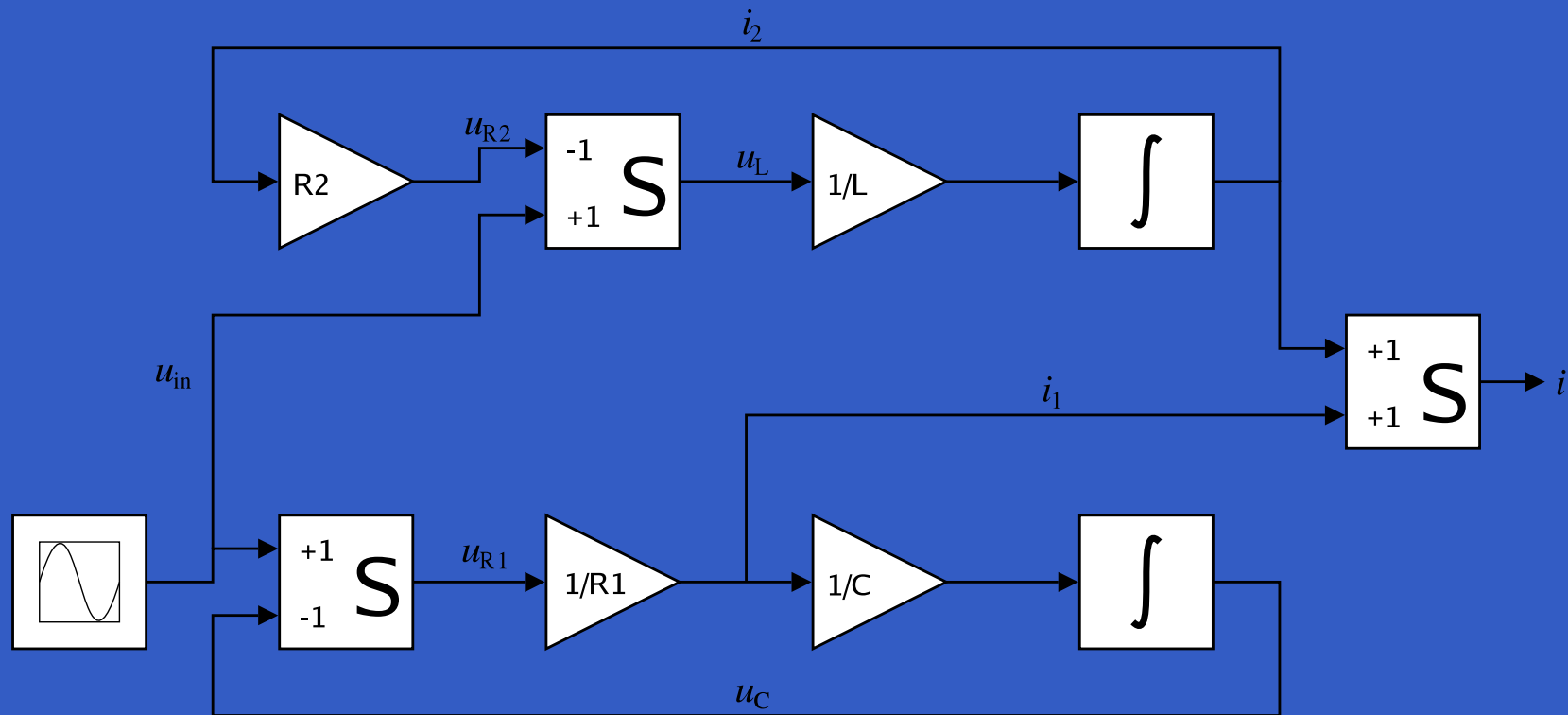
$$i_1 = \frac{u_{R_1}}{R_1}$$

$$u_C' = \frac{i_1}{C}$$

$$i = i_1 + i_2$$

Causal modeling: example (2)

Or, as a block diagram:



Drawbacks of causal modeling

Casual modeling bad fit for fundamentally non-causal domains like physical modeling:

- Structure of model and modeled system does not agree.
- Model not simple composition of models of physical components.
- Fixed causality hampers reuse.
- Burden of deriving a non-causal model rests with the modeler.

Non-causal modeling: example (1)

Non-causal resistor model:

$$\begin{aligned}u &= v_p - v_n \\i_p + i_n &= 0 \\u &= Ri_p\end{aligned}$$

Non-causal inductor model:

$$\begin{aligned}u &= v_p - v_n \\i_p + i_n &= 0 \\u &= Li_p'\end{aligned}$$

Non-causal modeling: example (2)

Non-causal capacitor model:

$$\begin{aligned}u &= v_p - v_n \\i_p + i_n &= 0 \\i_p &= Cu'\end{aligned}$$

Object-oriented modeling languages allow similarities between models to be exploited through classes and inheritance.

Non-causal modeling: example (3)

A non-causal model of the entire circuit is created by *instantiating* the component models (copy the equations and rename the variables).

The instantiated components are then *composed* by simply adding connection equations according to Kirchhoff's laws, e.g.:

$$\begin{aligned}v_{R_1,n} &= v_{C,p} \\i_{R_1,n} + i_{C,p} &= 0\end{aligned}$$

-
-
-

Functional Hybrid Modeling

Similar conceptual structure as Yampa, but:

Functional Hybrid Modeling

Similar conceptual structure as Yampa, but:

- First-class *relations* on signals instead of functions on signals to enable non-causal modeling.

Functional Hybrid Modeling

Similar conceptual structure as Yampa, but:

- First-class *relations* on signals instead of functions on signals to enable non-causal modeling.
- Employ state-of-the-art symbolic and numerical methods for sound and efficient simulation.

Functional Hybrid Modeling

Similar conceptual structure as Yampa, but:

- First-class *relations* on signals instead of functions on signals to enable non-causal modeling.
- Employ state-of-the-art symbolic and numerical methods for sound and efficient simulation.
- Adapted switch constructs.

First class signal relations

The type for a relation on a signal of type `Signal α`:

$SR\ \alpha$

Specific relations use a more refined type; e.g. the derivative relation:

$der :: SR\ (Real, Real)$

Since a signal carrying pairs is isomorphic to a pair of signals, *der* can be understood as a binary relation on two signals.

Defining relations

The following construct denotes a signal relation:

sigrel *pattern* **where** *equations*

The pattern introduces **signal variables** which at each point in time are going to be bound to to a “sample” of the corresponding signal.

Given $p :: t$, we have:

sigrel p **where** ... $:: SR\ t$

Equations

Let $e_i :: t_i$ be non-relational expressions possibly introducing new signal variables.

Point-wise equality; the equality must hold for all points in time:

$$e_1 = e_2$$

Relation “application”; the relation must hold for all points in time:

$$sr \diamond e_3$$

Here, sr is an **expression** having type $SR\ t_3$.

Modeling electrical components

The type `Pin` is assumed to be a record type describing an electrical connection. It has fields v for voltage and i for current.

$twoPin :: SR (Pin, Pin, Voltage)$

$twoPin = \mathbf{sigrel} (p, n, v) \mathbf{where}$

$$v = p.v - n.v$$

$$p.i + n.i = 0$$

$resistor :: Resistance \rightarrow SR (Pin, Pin)$

$resistor(r) = \mathbf{sigrel} (p, n) \mathbf{where}$

$$twoPin \diamond (p, n, v)$$

$$r \cdot p.i = v$$

Modeling an electrical circuit (1)

simpleCircuit :: SR Current

simpleCircuit = **sigrel** i where

resistor(1000) \diamond (*r1p*, *r1n*)

resistor(2200) \diamond (*r2p*, *r2n*)

capacitor(0.00047) \diamond (*cp*, *cn*)

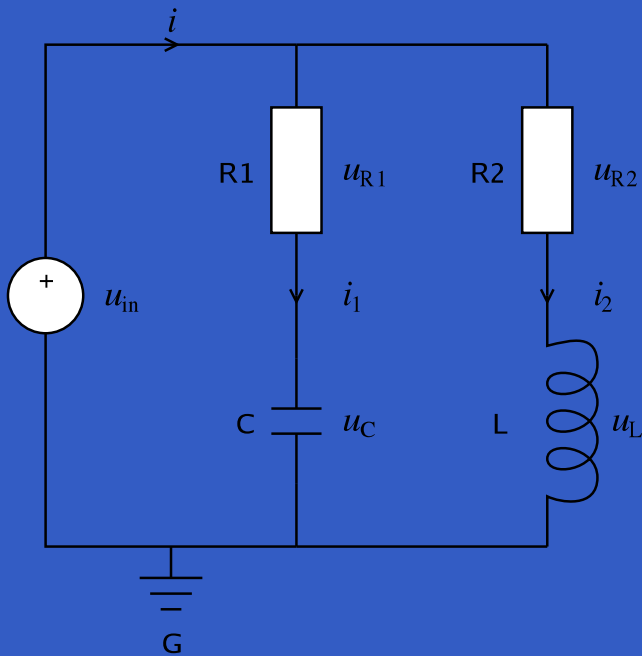
inductor(0.01) \diamond (*lp*, *ln*)

vSourceAC(12) \diamond (*acp*, *acn*)

ground \diamond *gp*

...

Modeling an electrical circuit (2)



...

connect $acp, r1p, r2p$

connect $r1n, cp$

connect $r2n, lp$

connect acn, cn, ln, gp

$i = r1p.i + r2p.i$

Central Research Questions

- Adapting Yampa's switching constructs, including handling initialization issues.

Central Research Questions

- Adapting Yampa's switching constructs, including handling initialization issues.
- Adapting non-causal modelling and simulation methods to a setting with first class signal relations: causality analysis, symbolic processing code generation after each switch.

Central Research Questions

- Adapting Yampa's switching constructs, including handling initialization issues.
- Adapting non-causal modelling and simulation methods to a setting with first class signal relations: causality analysis, symbolic processing code generation after each switch.
- Guaranteeing compositional correctness statically through the type system to the extent possible; e.g. employing dependent types to keep track of variable/equation balance.