Settable and Non-Interfering Signal Functions for FRP

Daniel Winograd-Cort
Paul Hudak

Department of Computer Science
Yale University

ICFP
Göteborg, Sweden
Tuesday, September 2, 2014

The Context:
Functional Reactive Programming

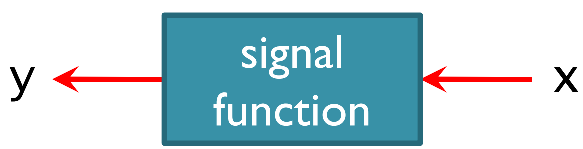
- Programming with *continuous values* and *streams of events*.

The Context:

Functional Reactive Programming

- Programming with *continuous values* and *streams of events*.
- Like drawing *signal processing diagrams*:

<u>signal processing diagram</u> <u>equivalent arrow syntax in Haskell</u>



$$\equiv$$

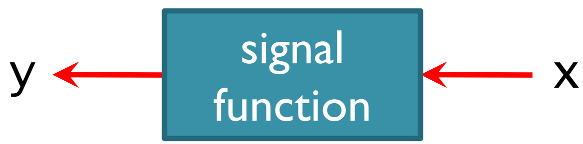
$$y \leftarrow sigfun \prec x$$

The Context:

Functional Reactive Programming

- Programming with *continuous values* and *streams of events*.
- Like drawing *signal processing diagrams*:

signal processing diagram



≡

equivalent arrow syntax in Haskell

$$y \leftarrow sigfun \prec x$$

- Previously used in:
  - Yampa: robotics, vision, animation
  - Nettle: networking
  - Euterpea: sound synthesis and audio processing

How they work and how we will represent them

ARROWS

Event-Based vs Continuous

- A stream can be continuously defined, typically as a time-varying value

  - By default, we use this notation

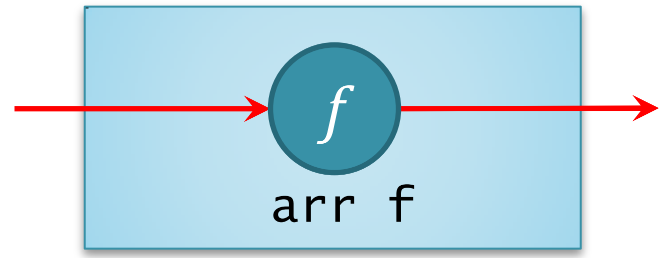
Event-Based vs Continuous

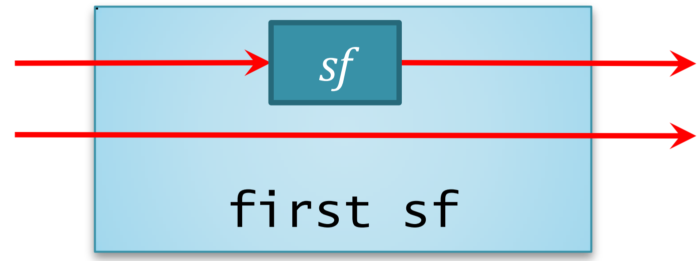
- A stream can be continuously defined, typically as a time-varying value

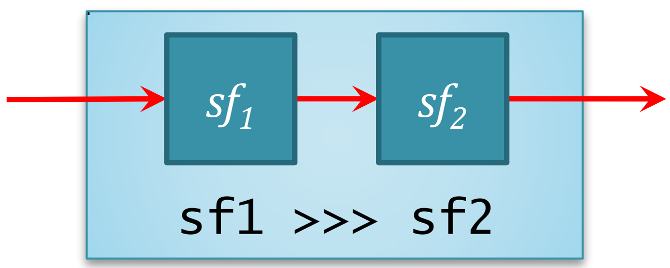
  - By default, we use this notation

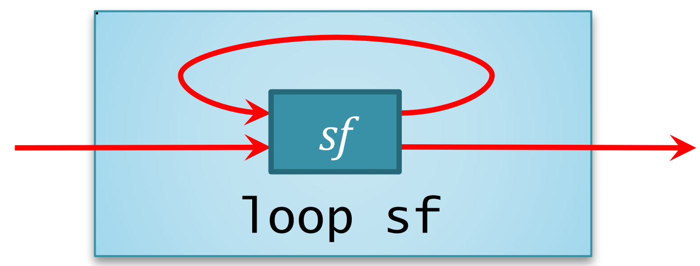
- Rather than accepting a continuous stream, some signal functions accept discrete events, defined only at specific times

Standard Arrow Operators

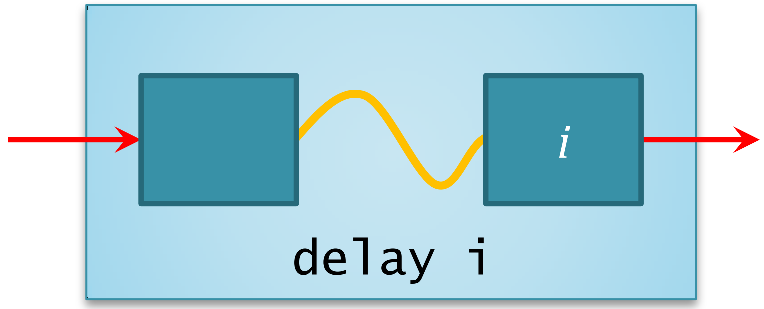






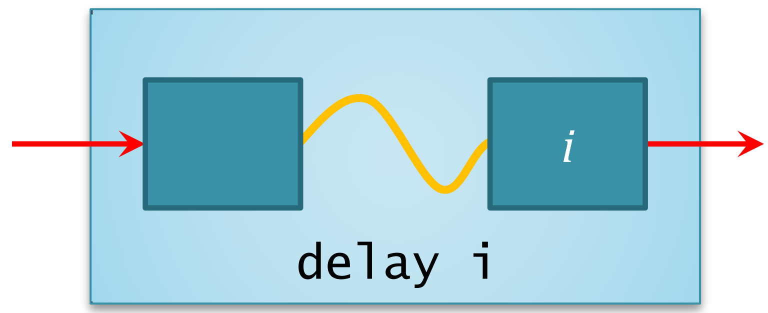


Stateful Arrows



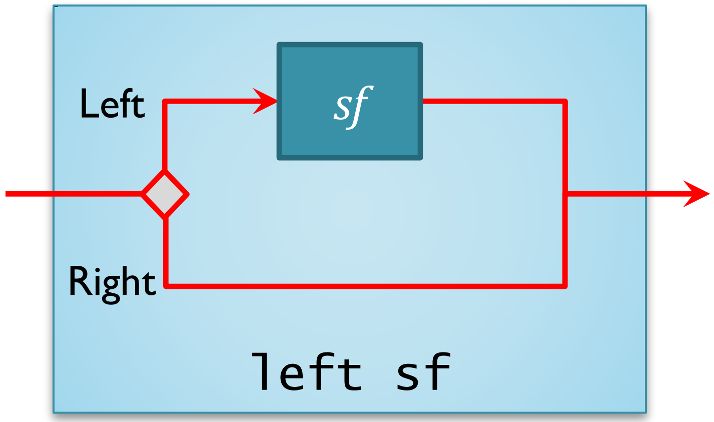
- With continuous semantics, the length of the delay approaches zero

Stateful Arrows



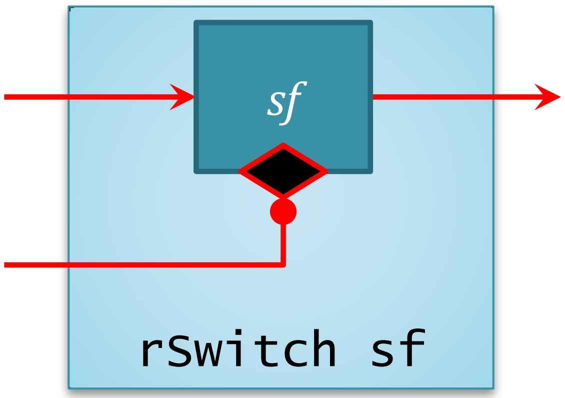
- With continuous semantics, the length of the delay approaches zero
- When used in conjunction with loop, delay allows one to create stateful signal functions

Arrow Choice



- With choice, running the signal function is a dynamic decision

Higher Order Arrows



- Dynamic
- Components that start and stop

That's just a Monad

- Arrows with switch are equivalent to Monad.

That's just a Monad

- Arrows with switch are equivalent to Monad.
- Switch takes away arrows' static structure
  - Not as easy to optimize
  - Harder for certain embedded systems

So why switch?

So why switch?

- Higher order signal expression
  - Inherently dynamic
  - Sometimes the arrow style is right even though switching is unavoidable

So why switch?

- Higher order signal expression
  - Inherently dynamic
  - Sometimes the arrow style is right even though switching is unavoidable
- Ability to start and stop signal functions
  - “Power choice”
  - Increase performance by switching out signal functions that are not necessary

Contributions

- A way to do classic switch-like behavior without switch

Contributions

- A way to do classic switch-like behavior without switch
  - Resettability allows signal functions to act as if brand new

Contributions

- A way to do classic switch-like behavior without switch
  - Resettability allows signal functions to act as if brand new
  - Non-Interfering Choice increases arrows' standard choice's power

Contributions

- A way to do classic switch-like behavior without switch
  - Resettability allows signal functions to act as if brand new
  - Non-Interfering Choice increases arrows' standard choice's power
- Extra benefits!

Contributions

- A way to do classic switch-like behavior without switch
  - Resettability allows signal functions to act as if brand new
  - Non-Interfering Choice increases arrows' standard choice's power
- Extra benefits!
  - General settability
  - Arrowized Recursion

An example

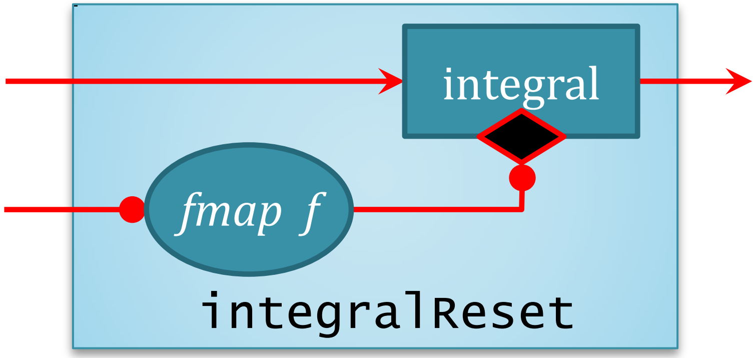
SWITCHING FOR STATE

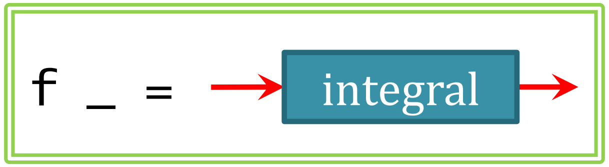
Example: IntegralReset

- A signal function that calculates an integral but can be reset with an event.

Example: IntegralReset

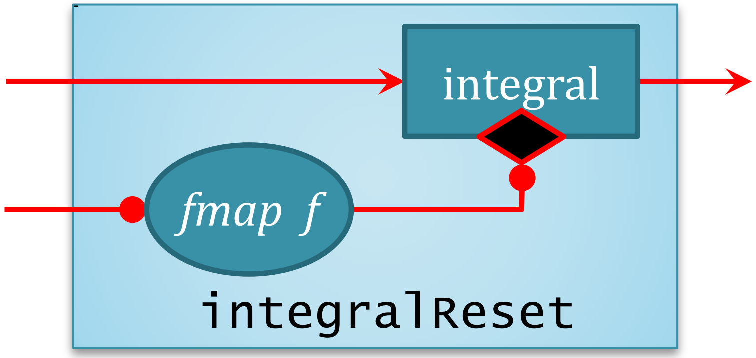
- A signal function that calculates an integral but can be reset with an event.

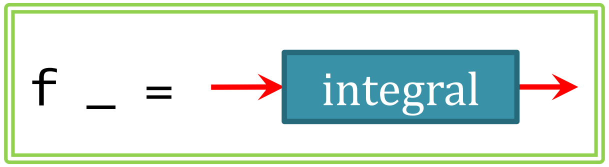




Example: IntegralReset

- A signal function that calculates an integral but can be reset with an event.

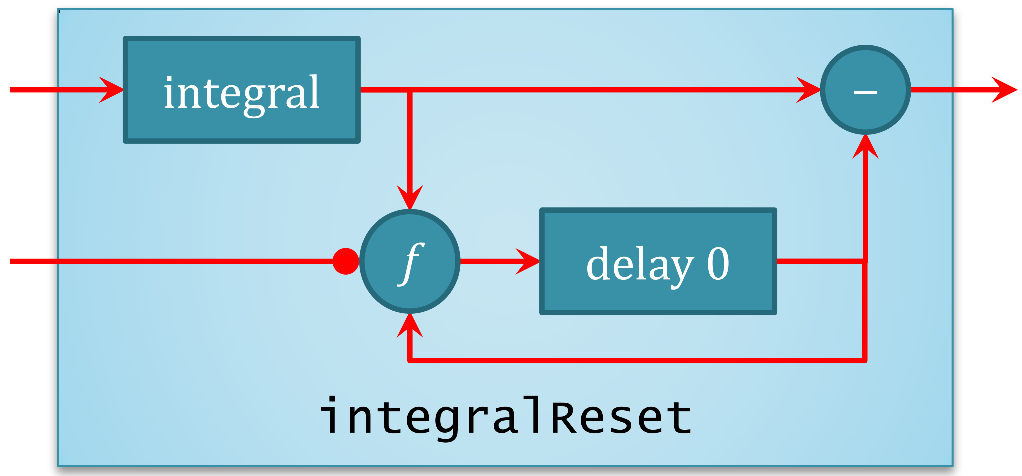




- Can we even do this without switch?

Example: IntegralReset

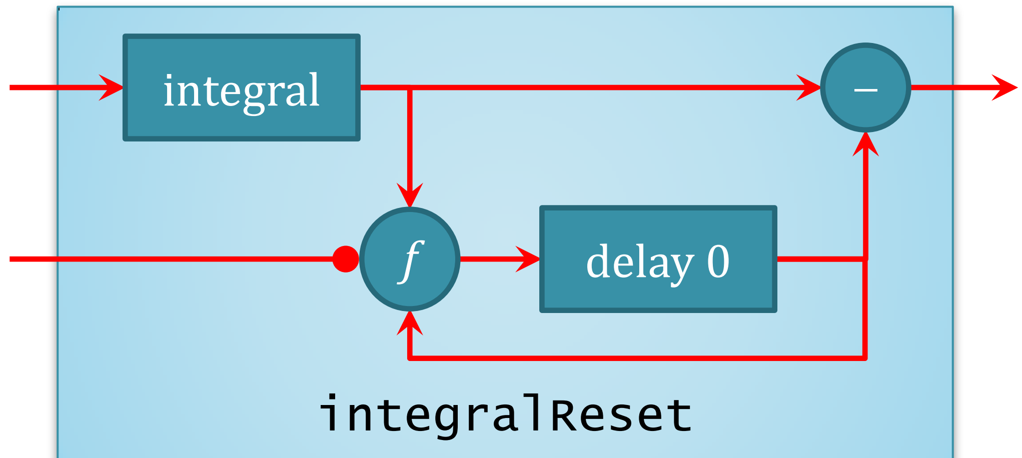
- Without switch, we can simulate a reset, but we can't modify integral itself




```
f v e k = if isEvent e then v else k
```

Example: IntegralReset

- Without switch, we can simulate a reset, but we can't modify integral itself




```
f v e k = if isEvent e then v else k
```

- This solution is inelegant and does not scale

Resetting State

- We want to access the state inside a signal function

- But what's inside of an arbitrary signal function?

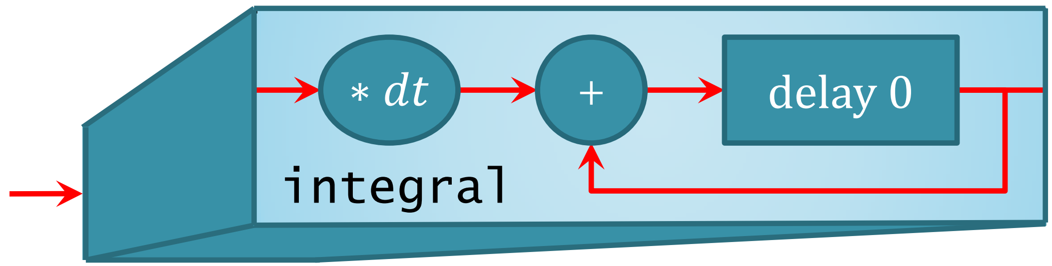
Resetting State

- We want to access the state inside a signal function

- But what's inside of an arbitrary signal function?
- All state is saved with loop and delay

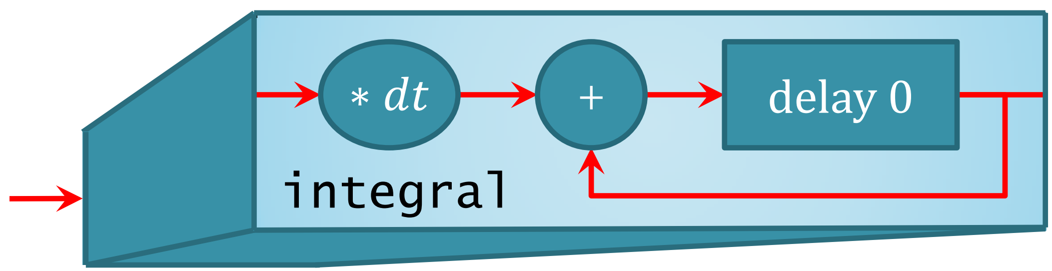
Resetting State

- We want to access the state inside a signal function



Resetting State

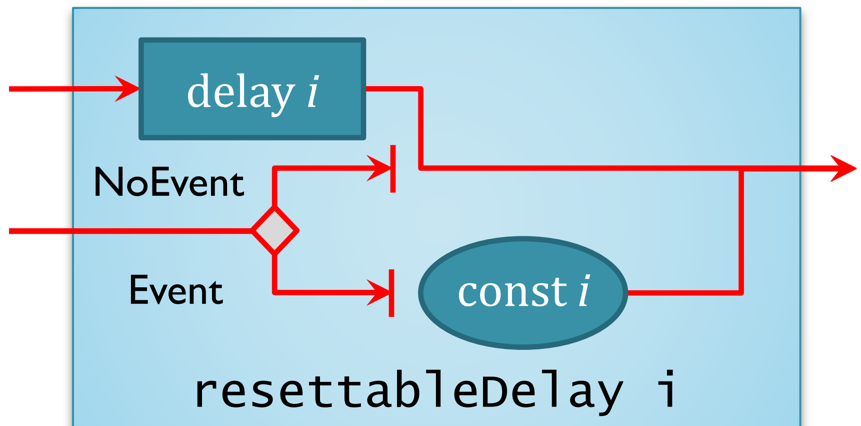
- We want to access the state inside a signal function



- If we could reach in and restart the delay, then integral would behave as if it just started

Resettable Delay

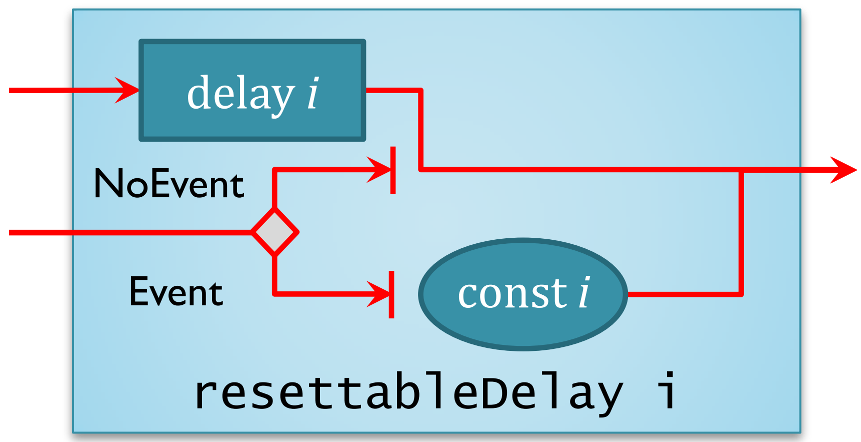
- Let's consider a new delay that can be reset directly



- When the event is given, resettableDelay reverts to its starting state.

Resettable Delay

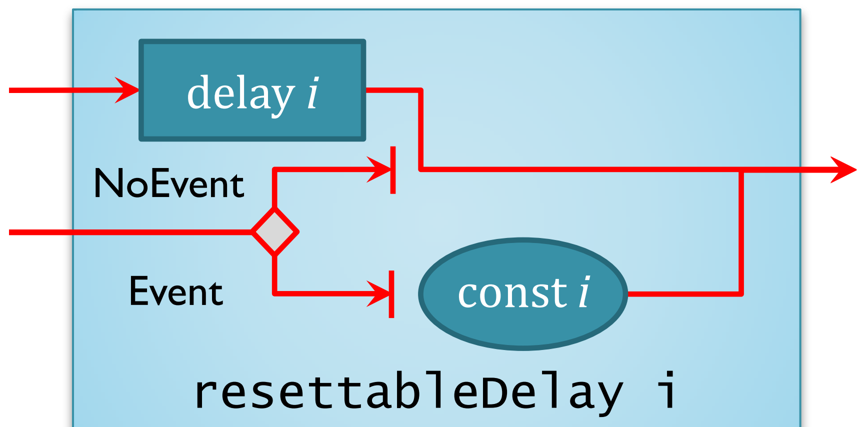
- Let's consider a new delay that can be reset directly



- When the event is given, resettableDelay reverts to its starting state.
- Does this scale?

Resettable Delay

- Let's consider a new delay that can be reset directly



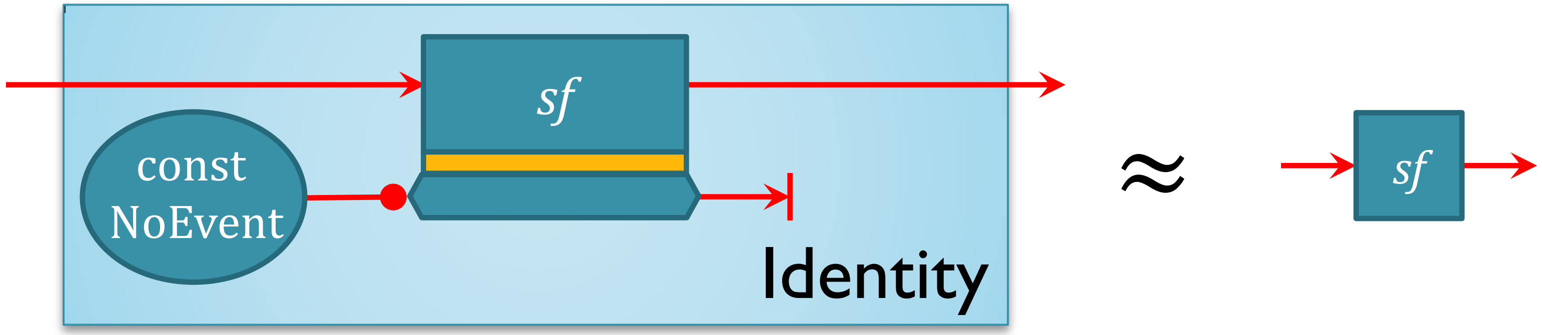
- When the event is given, resettableDelay reverts to its starting state.
- Does this scale? YES

General Settability

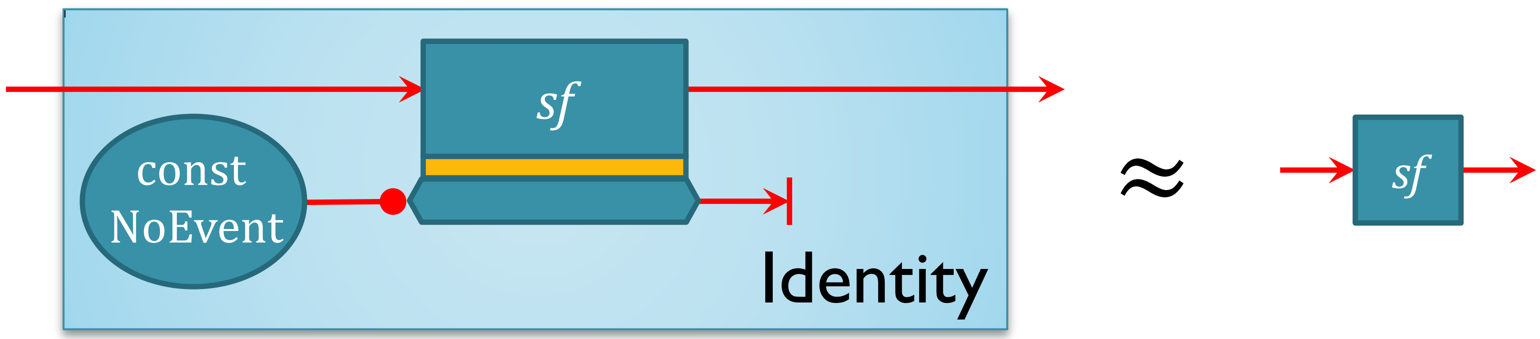
- We can take any signal function and transform it into a settable signal function

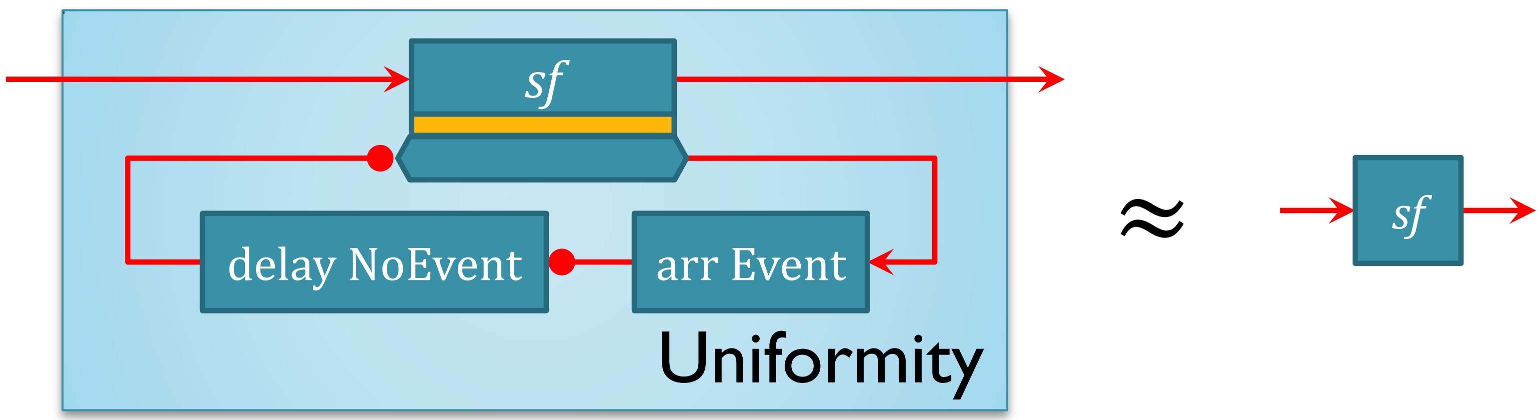
- The top wires are the standard signals
- The bottom wires are State signals
  - The input Event State can be used to change *sf*'s internal state
  - The output State is used to capture the current internal state

Settable Laws

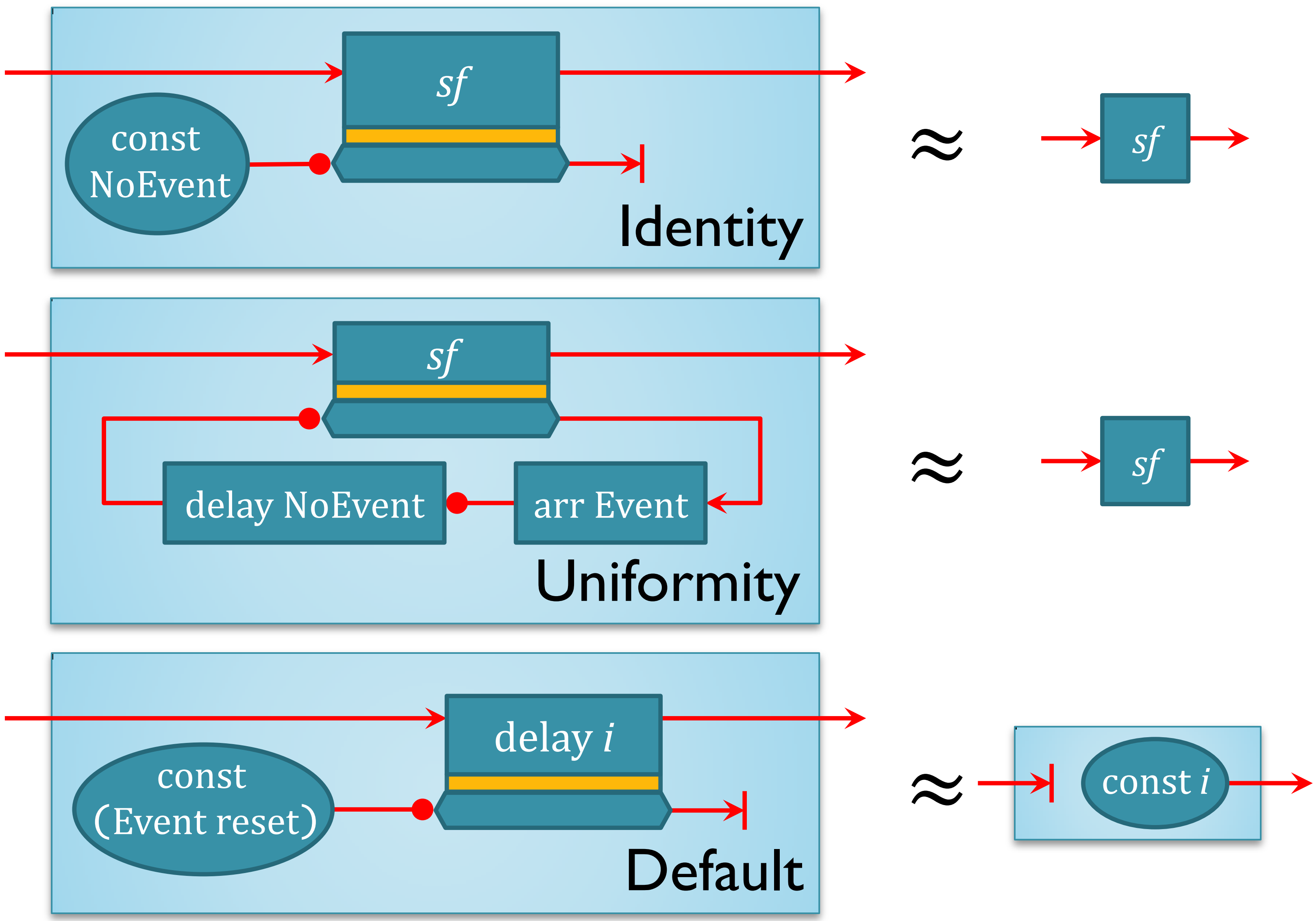


Settable Laws



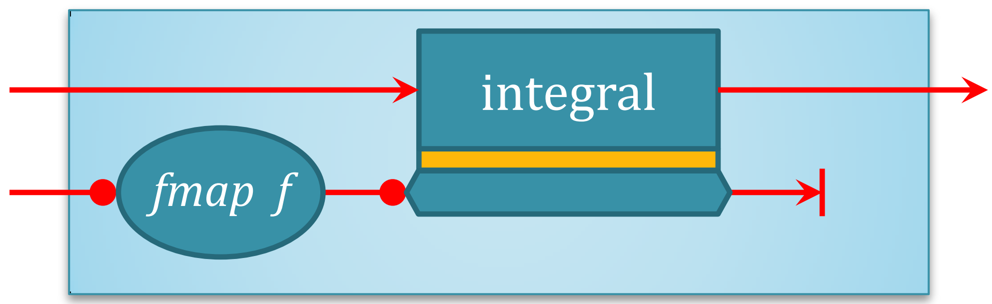


Settable Laws



Example: IntegralReset

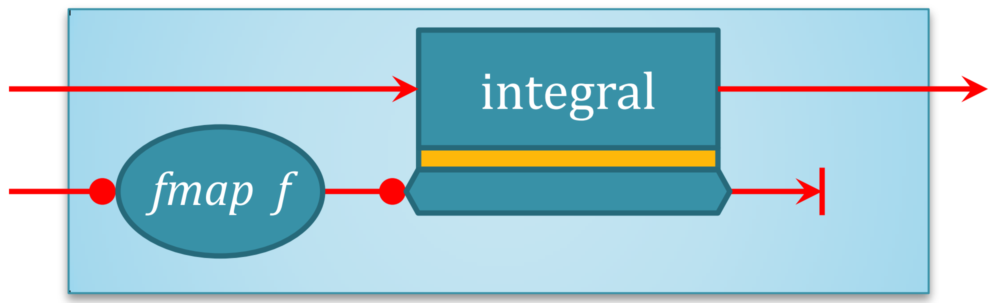
- Settability makes the problem trivial




```
f _ = reset
```

Example: IntegralReset

- Settability makes the problem trivial




```
f _ = reset
```

- We no longer need the overkill of lifting a signal function to the signal level

Another example

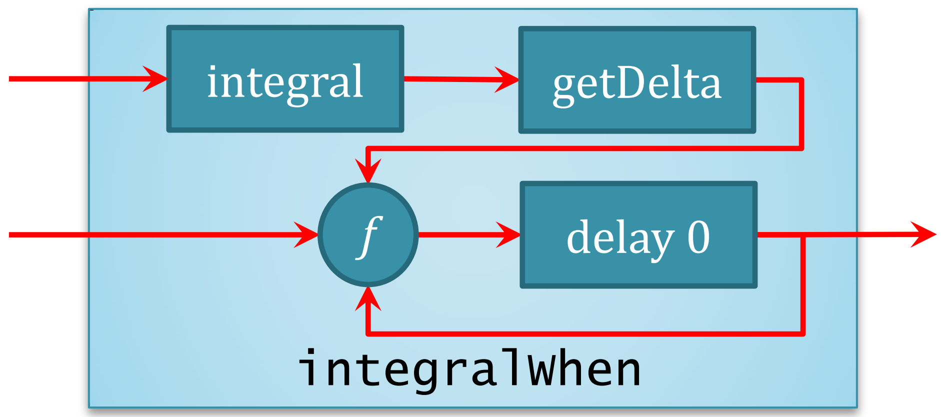
A STOPPING SWITCH

Example: IntegralWhen

- A signal function that performs an integral only under a given condition

Example: IntegralWhen

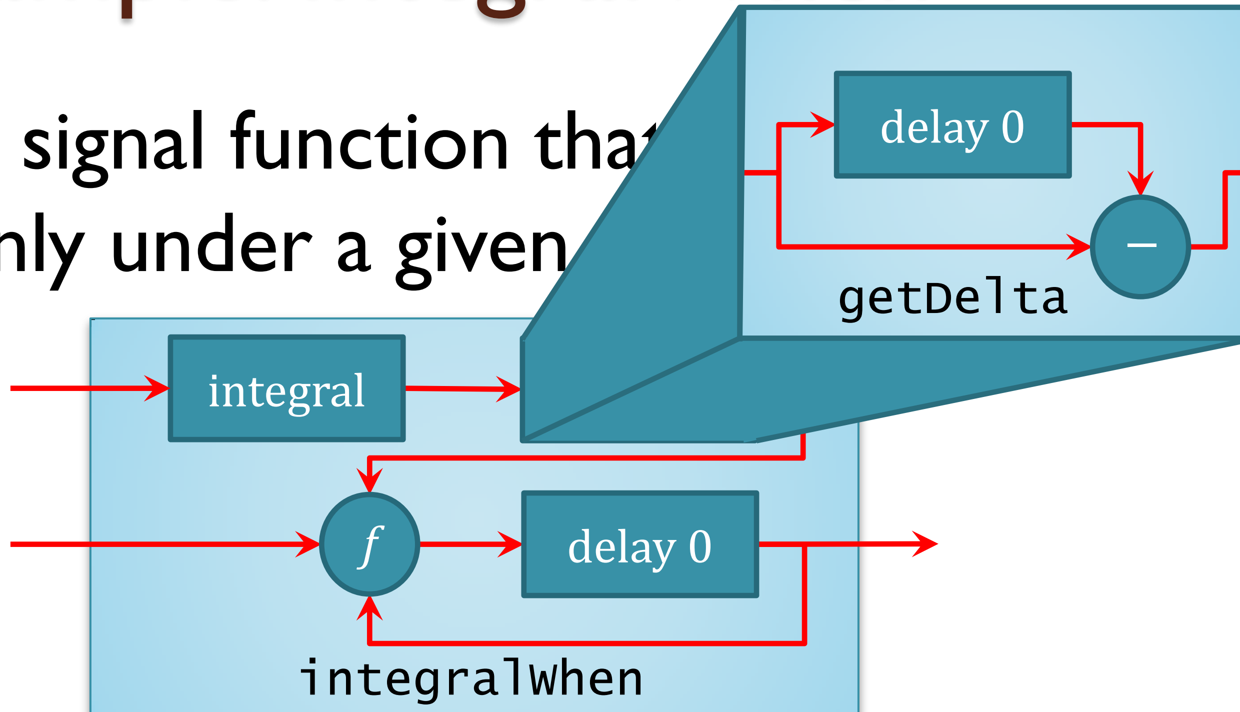
- A signal function that performs an integral only under a given condition



f Δv b res_{prev} = **if** b **then** res_{prev} + Δv **else** res_{prev}

Example: IntegralWhen

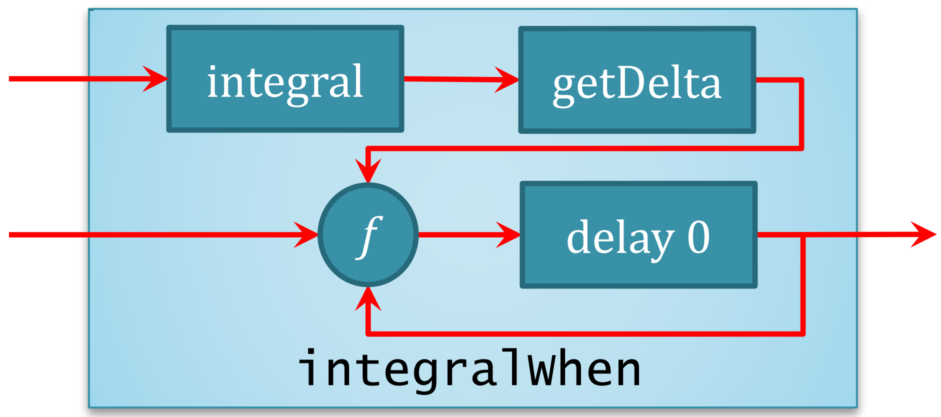
- A signal function tha egral only under a given



$$f\ \Delta v\ b\ res_{prev} = \textbf{if}\ b\ \textbf{then}\ res_{prev} + \Delta v\ \textbf{else}\ res_{prev}$$

Example: IntegralWhen

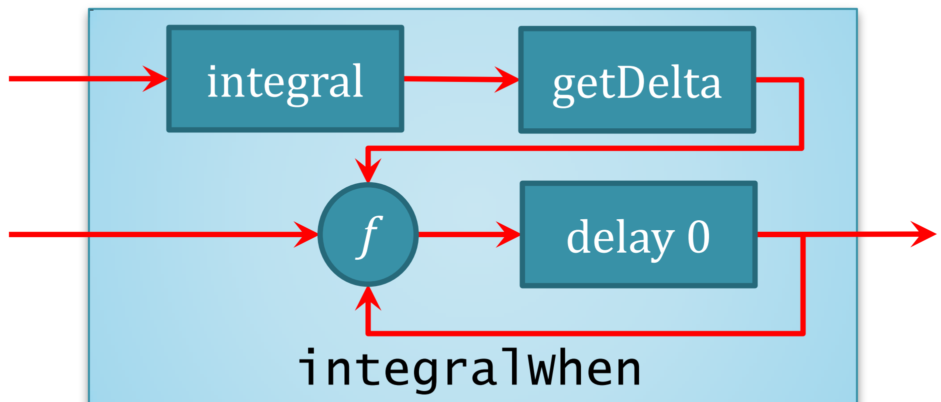
- A signal function that performs an integral only under a given condition



$$f\ \Delta v\ b\ res_{prev} = \textbf{if}\ b\ \textbf{then}\ res_{prev} + \Delta v\ \textbf{else}\ res_{prev}$$

Example: IntegralWhen

- A signal function that performs an integral only under a given condition

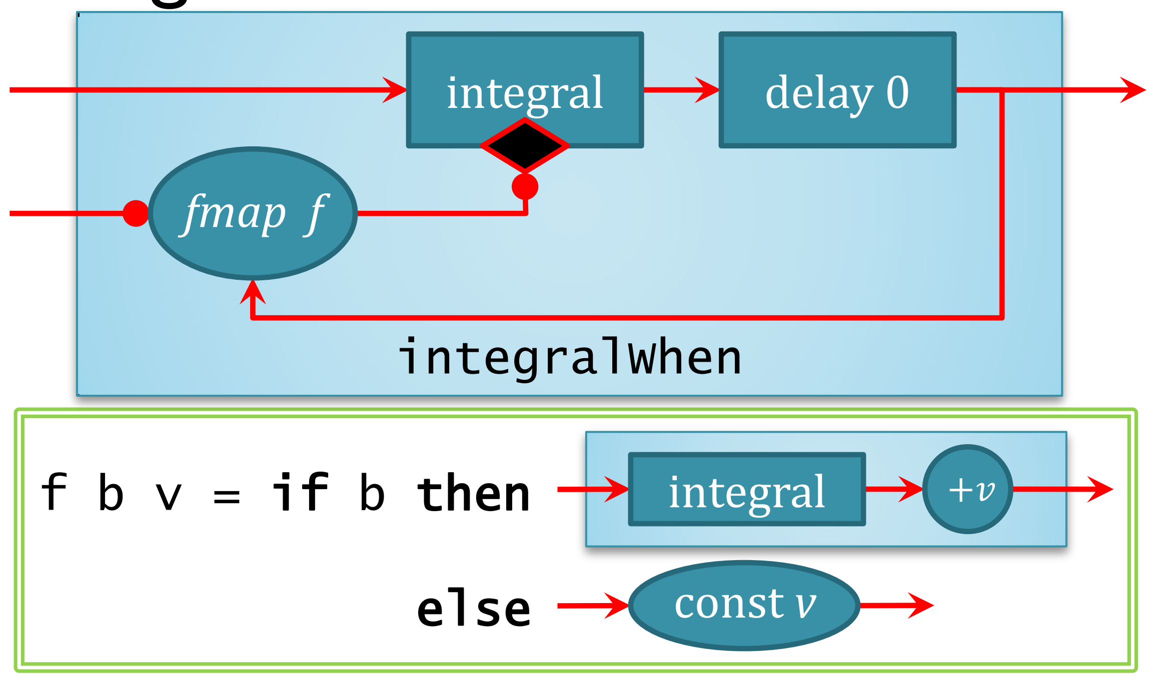



```
f Δv b res_prev = if b then res_prev + Δv else res_prev
```

- The integral is calculated regardless of the condition, but is only used sometimes

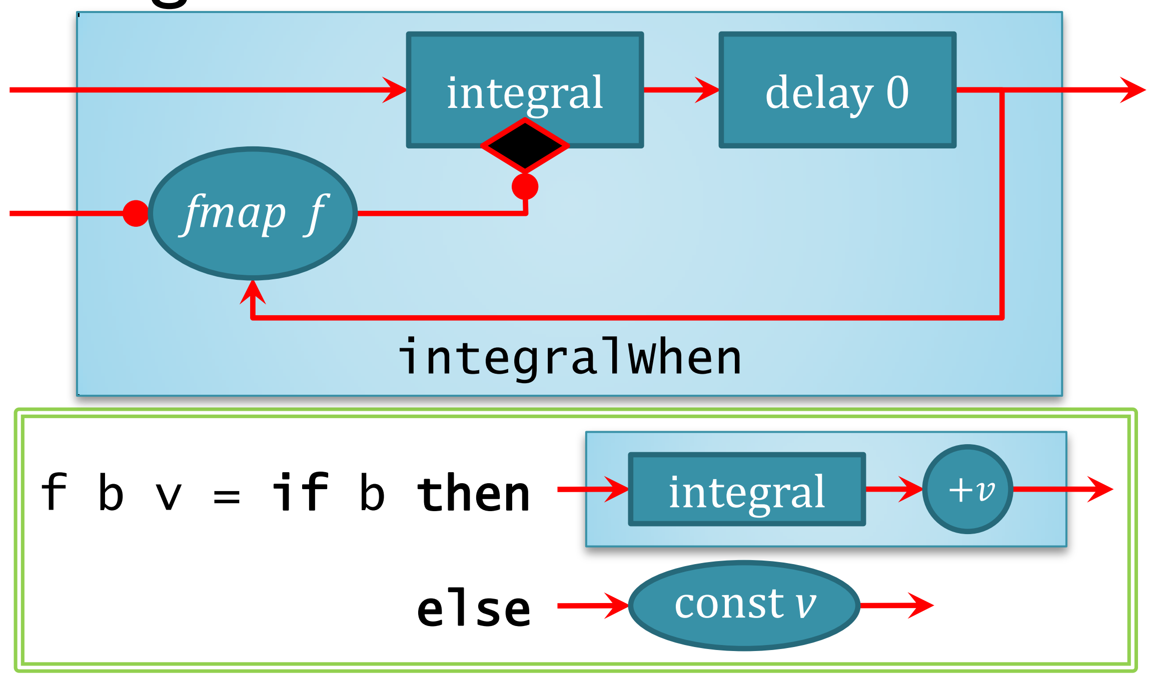
Example: IntegralWhen

- With switch, we have the power to stop the integral when it is not needed.



Example: IntegralWhen

- With switch, we have the power to stop the integral when it is not needed.



- Why do we need switch at all?
- Why can't we just use choice?

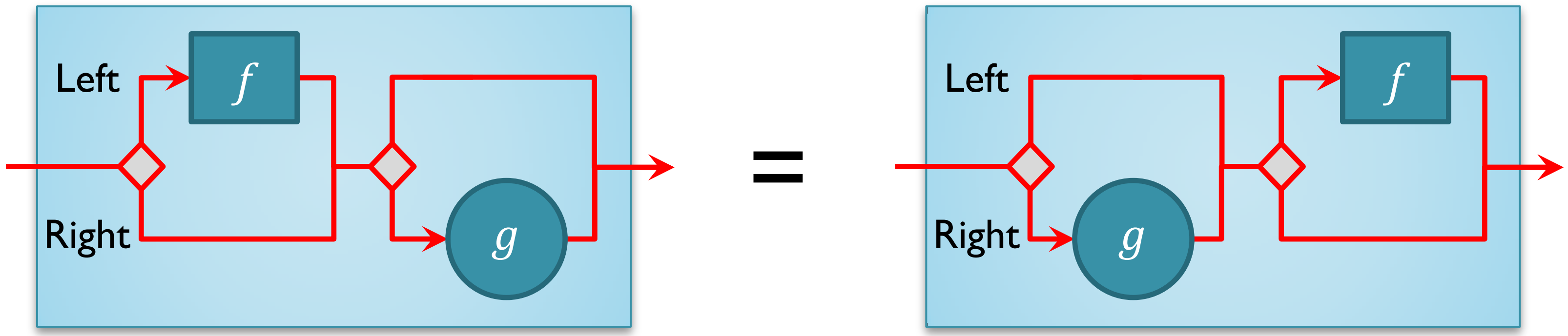
Arrow Choice Laws

Extension	`left (arr f) = arr (left f)`
Functor	`left (f >>> g) = left f >>> left g`
Exchange	`left f >>> arr (right g) = arr (right g) >>> left f`
Unit	`f >>> arr Left = arr Left >>> left f`
Assoc	left (left f) >>> arr $assoc_+$ = arr $assoc_+$ >>> left f

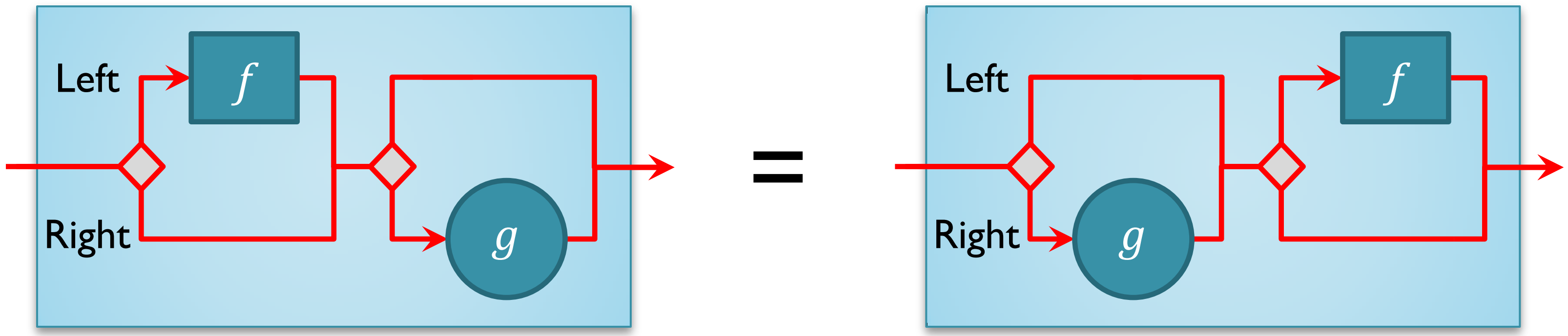
Arrow Choice Laws

Extension	`left (arr f) = arr (left f)`
Functor	`left (f >>> g) = left f >>> left g`
Exchange	`left f >>> arr (right g) = arr (right g) >>> left f`
Unit	`f >>> arr Left = arr Left >>> left f`
Assoc	left (left f) >>> arr $assoc_+$ = arr $assoc_+$ >>> left f

Exchange

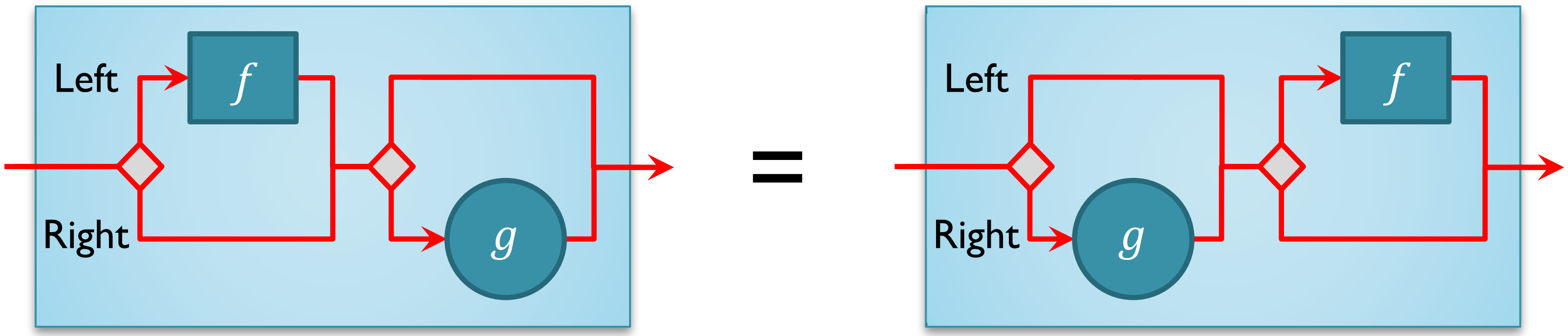


Exchange



- Why isn't this commutative?

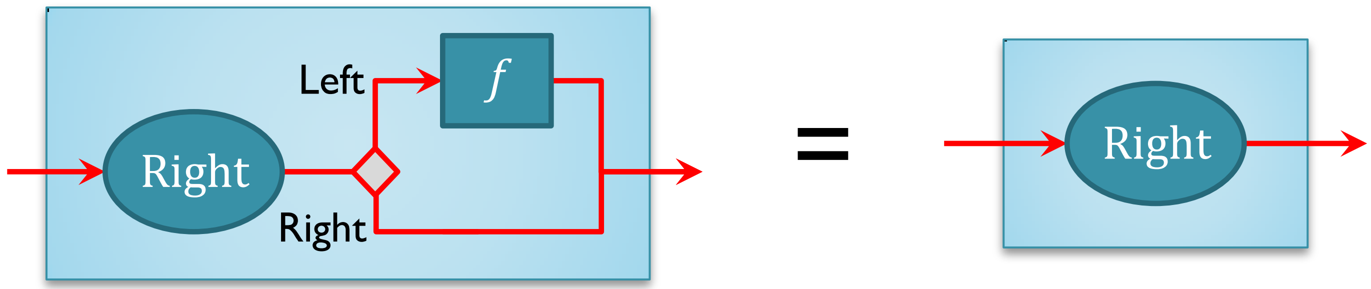
Exchange



- Why isn't this commutative?
  - Some arrows have effects

Non-Interference

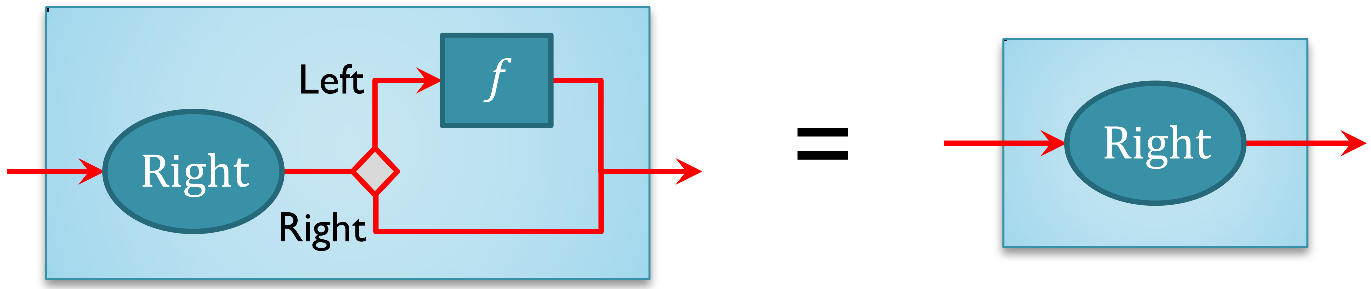
- We strengthen exchange into **non-interference**



- If the input value is a Right value, then the program will behave the same if there is a left function after it or not.

Non-Interference

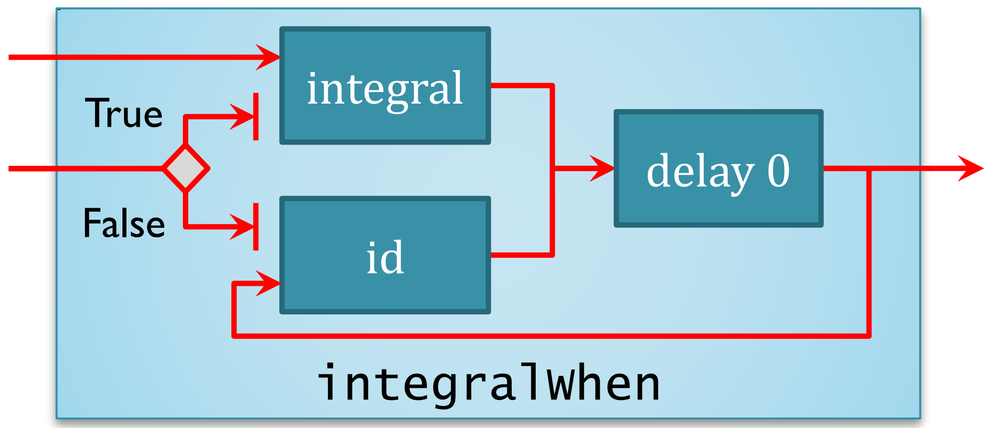
- We strengthen exchange into **non-interference**



- If the input value is a Right value, then the program will behave the same if there is a left function after it or not.
- The unused branch of a choice is now guaranteed to not run

Example: IntegralWhen

- With non-interfering choice, we make another attempt.



- When the condition is true, only integral is used, and when it is false, only id

Non-Interfering Choice gives us even more

ARROWIZED RECURSION

Recursion in AFRP

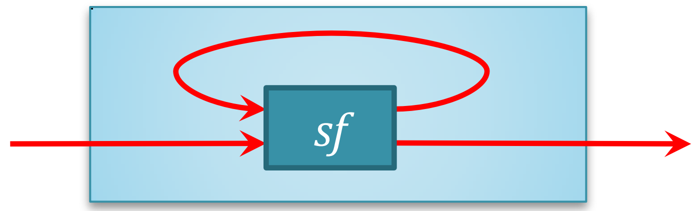
- AFRP typically provides two forms of recursion

Recursion in AFRP

- AFRP typically provides two forms of recursion
  - Value-level recursion
  - Structural Recursion

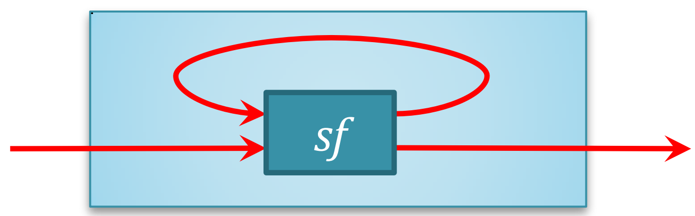
Value-Level Recursion

- Value-level recursion is achieved with the loop operator



Value-Level Recursion

- Value-level recursion is achieved with the loop operator



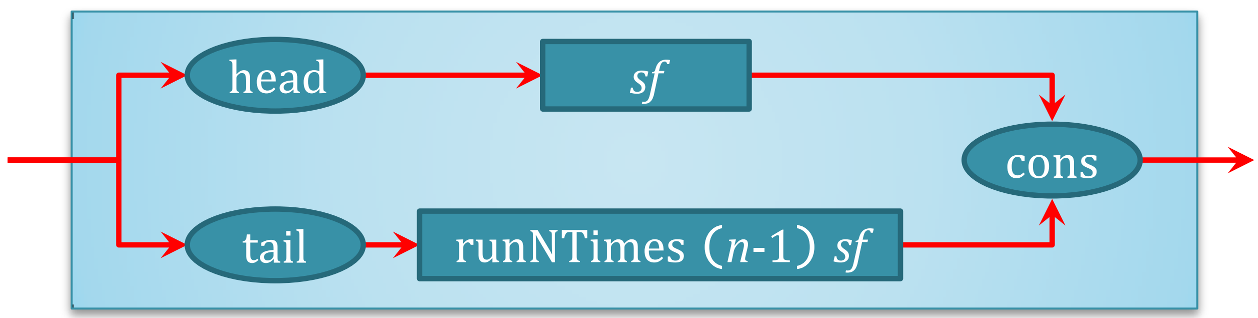
- loop is essentially an extension of fix

Structural Recursion

- Structural recursion is "outside" the arrow and uses the native conditional

```
runNTimes :: Int -> (a ~> b) -> ([a] ~> [b])
runNTimes 0 _ =
```

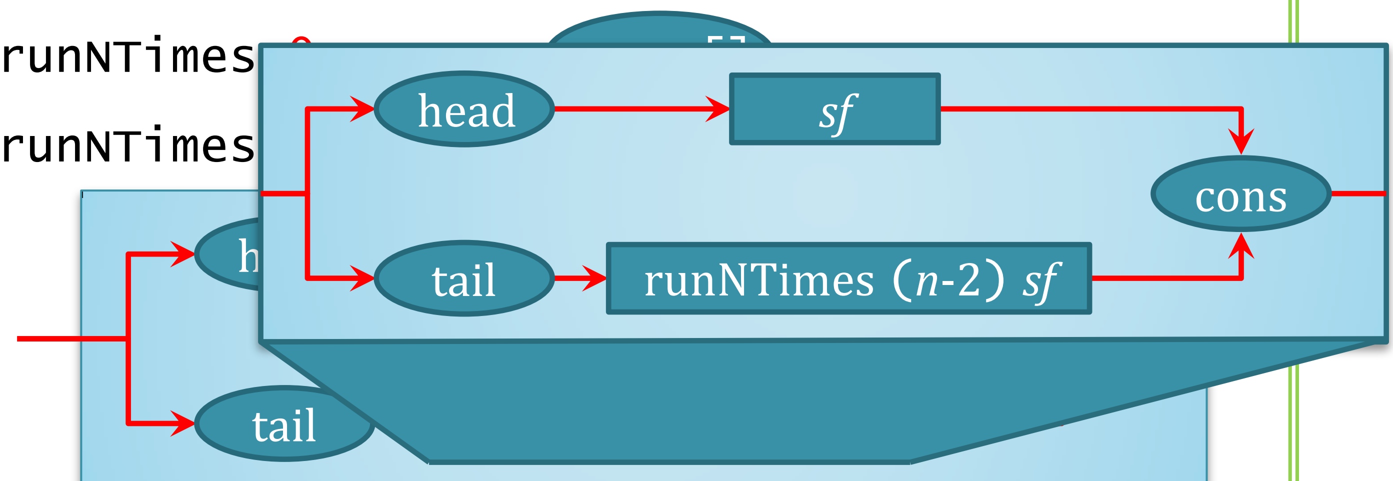

```
runNTimes n sf =
```



Structural Recursion

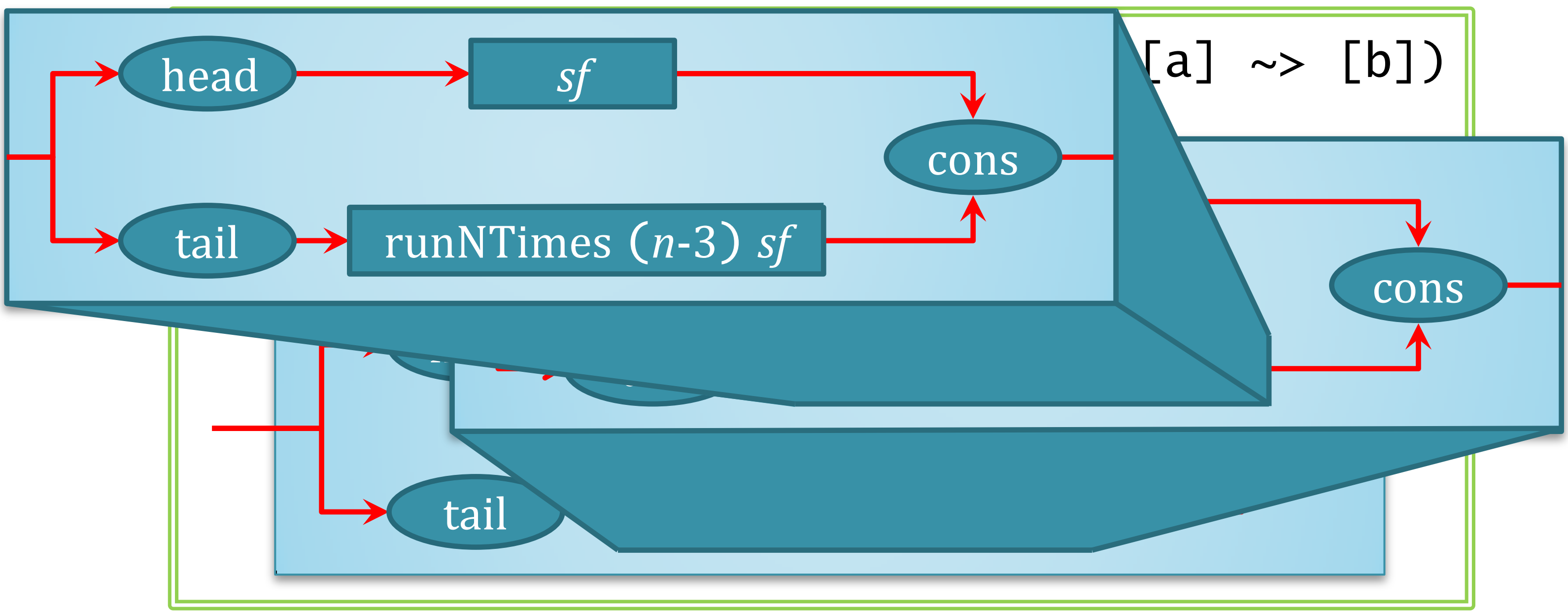
- Structural recursion is "outside" the arrow and uses the native conditional

```
runNTimes :: Int -> (a ~> b) -> ([a] ~> [b])
runNTimes
runNTimes
```



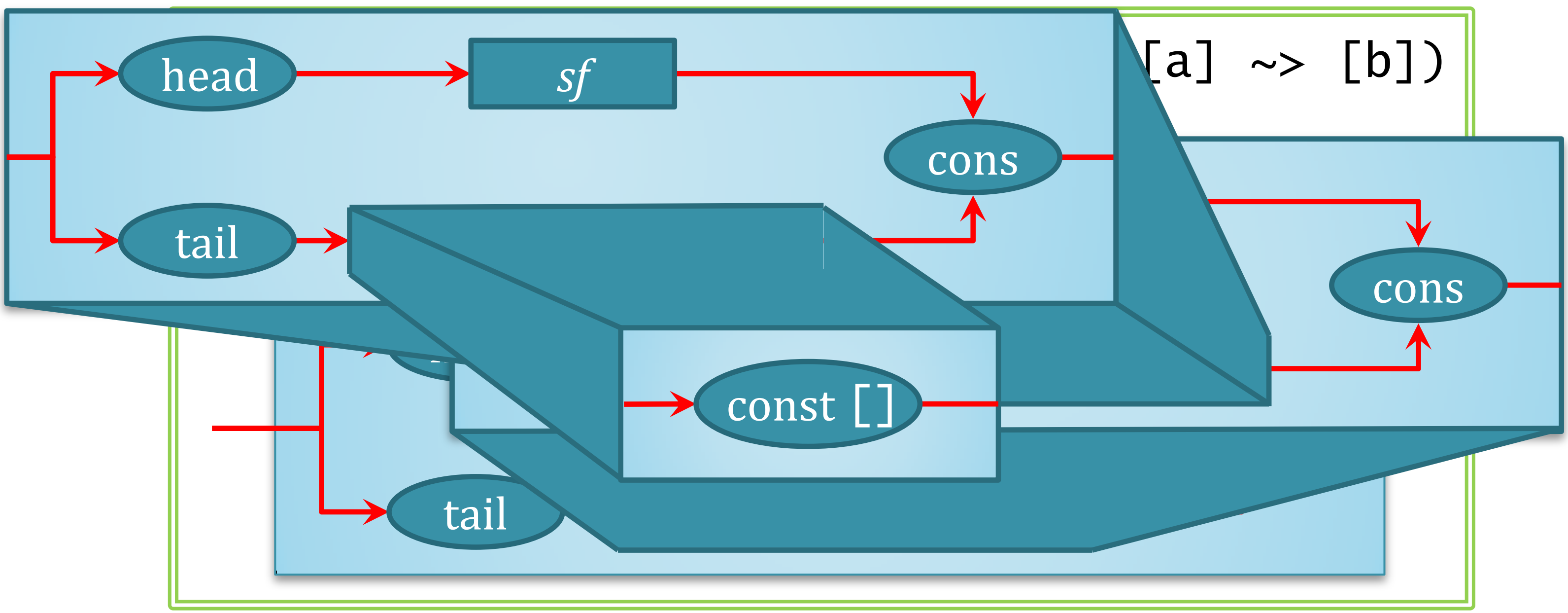
Structural Recursion

- Structural recursion is "outside" the arrow and uses the native conditional



Structural Recursion

- Structural recursion is "outside" the arrow and uses the native conditional

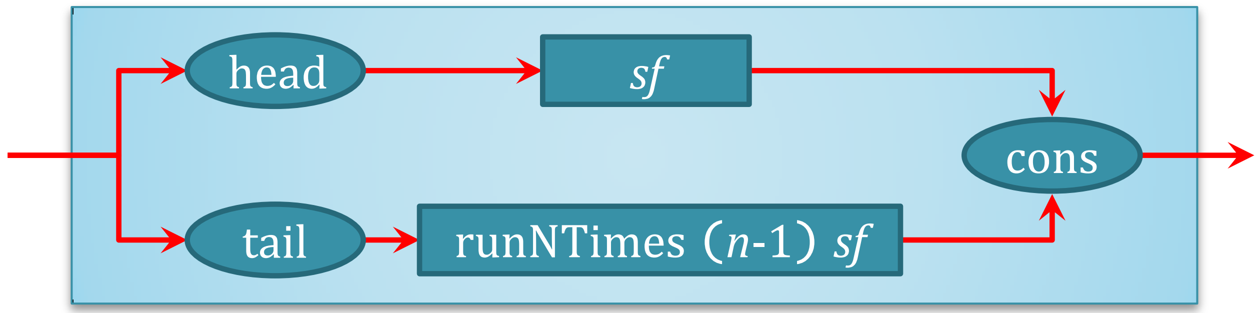


Structural Recursion

- Structural recursion is "outside" the arrow and uses the native conditional

```
runNTimes :: Int -> (a ~> b) -> ([a] ~> [b])
runNTimes 0 _ =
```


```
runNTimes n sf =
```

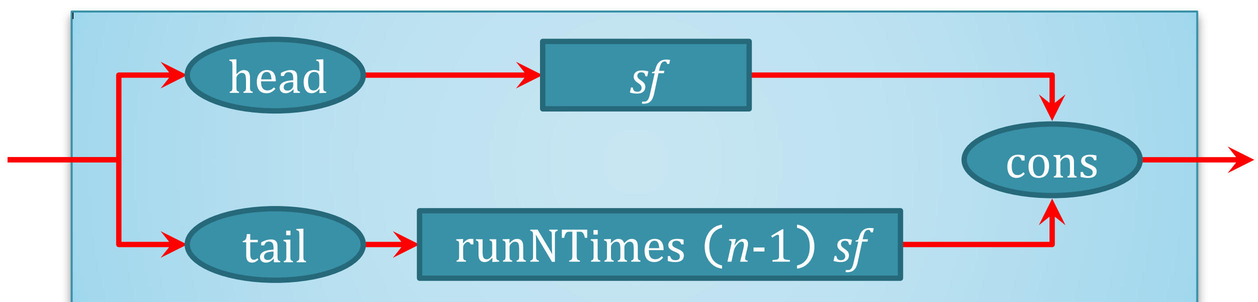


Structural Recursion

- Structural recursion is "outside" the arrow and uses the native conditional

```
runNTimes :: Int -> (a ~> b) -> ([a] ~> [b])
runNTimes 0 _ =
```


```
runNTimes n sf =
```

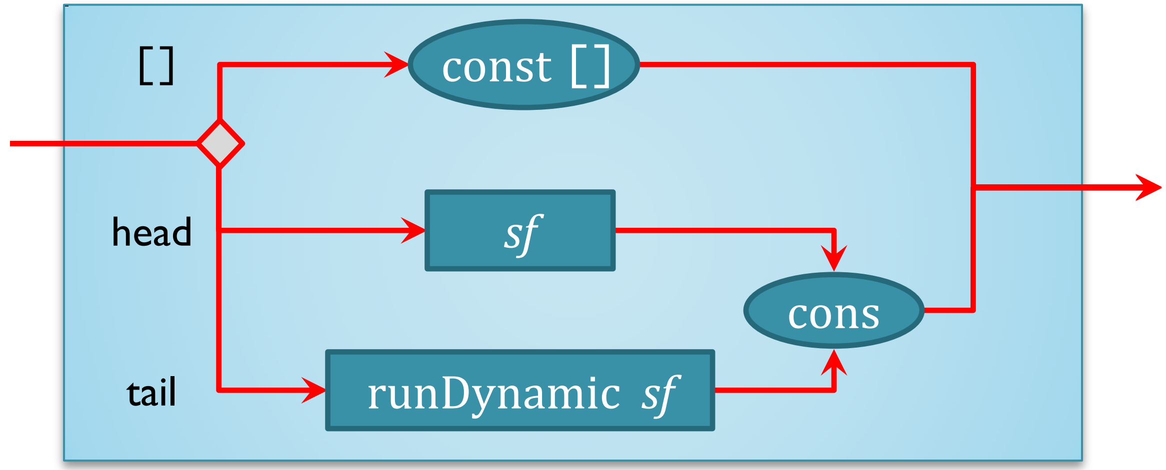


- Note that the arrow's structure is static and independent of the arrowized data

Arrowized Recursion

- Arrowized recursion uses arrow choice to determine branching

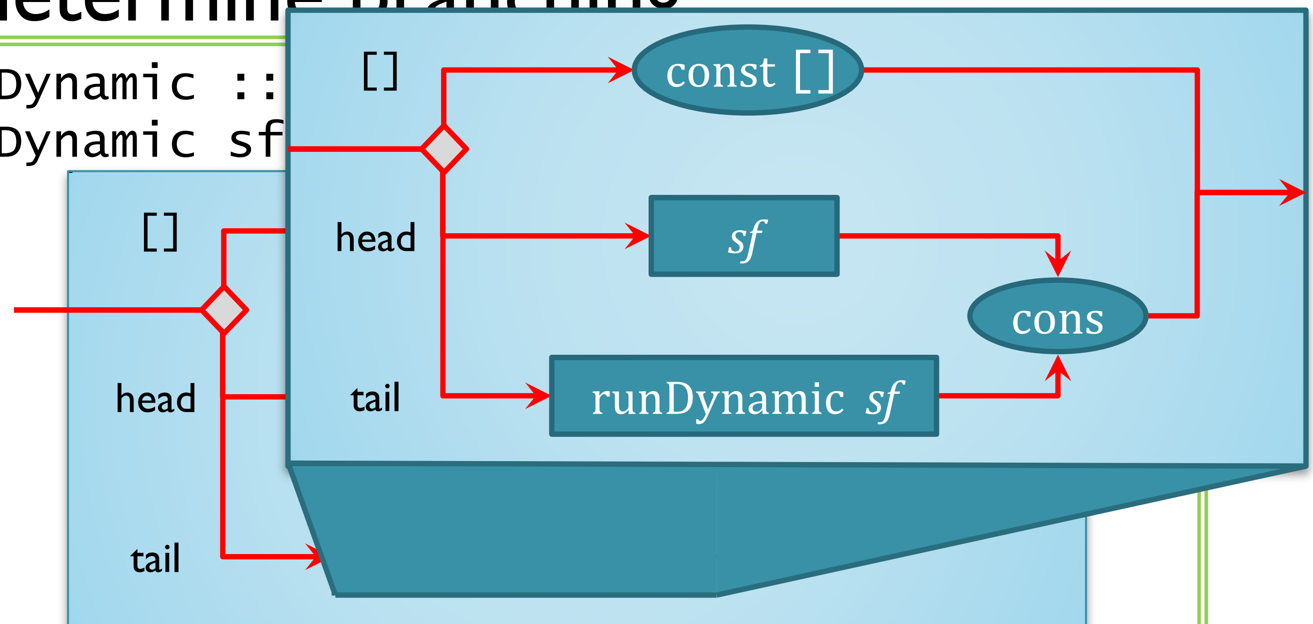
```
runDynamic :: (a ~> b) -> ([a] ~> [b])
runDynamic sf =
```



Arrowized Recursion

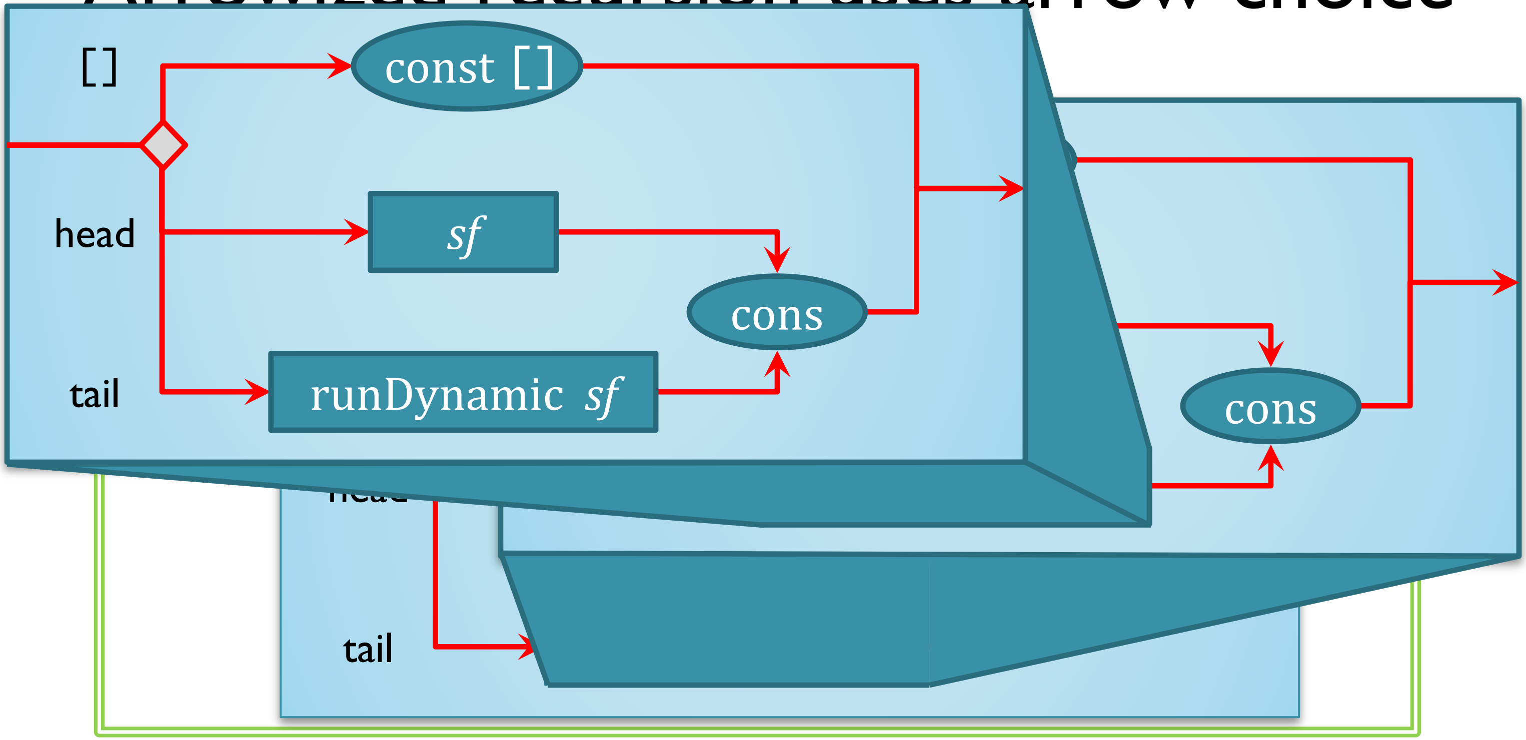
- Arrowized recursion uses arrow choice to determine branching

```
runDynamic ::
runDynamic sf
```



Arrowized Recursion

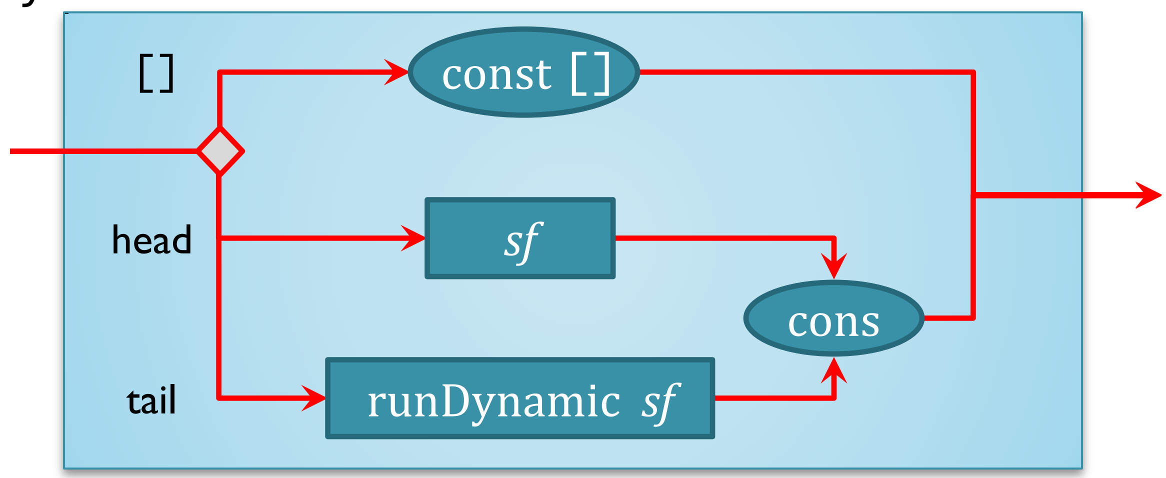
- Arrowized recursion uses arrow choice



Arrowized Recursion

- Arrowized recursion uses arrow choice to determine branching

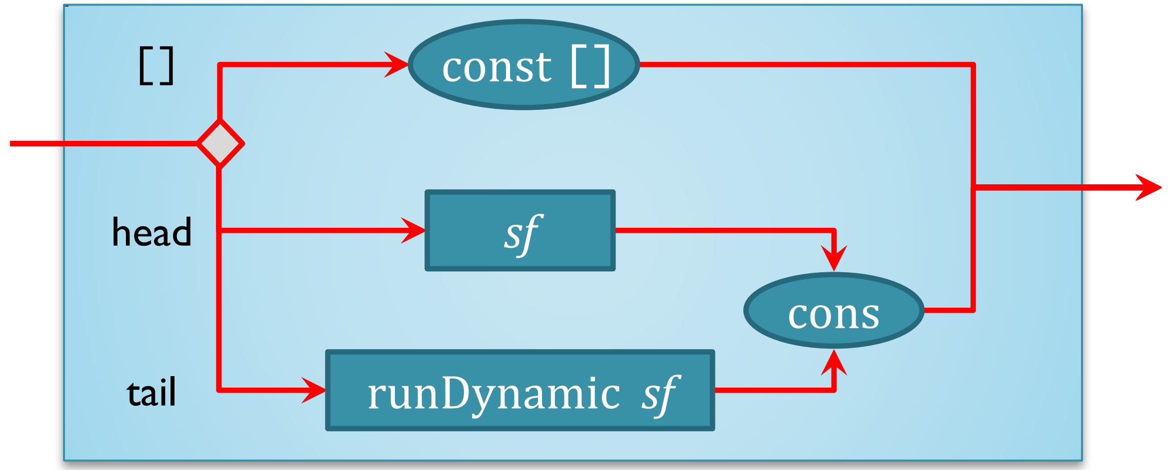
```
runDynamic :: (a ~> b) -> ([a] ~> [b])
runDynamic sf =
```



Arrowized Recursion

- Arrowized recursion uses arrow choice to determine branching

```
runDynamic :: (a ~> b) -> ([a] ~> [b])
runDynamic sf =
```



- The arrow structure is not technically static, but it is predictably dynamic

The benefit of static arrows over dynamic arrows

OPTIMIZATION

Causal Commutative Arrows

- Liu, Cheng, Hudak [2009] introduced CCA
  - CCAs can be heavily optimized
  - Performance increases of 10-40 times

Causal Commutative Arrows

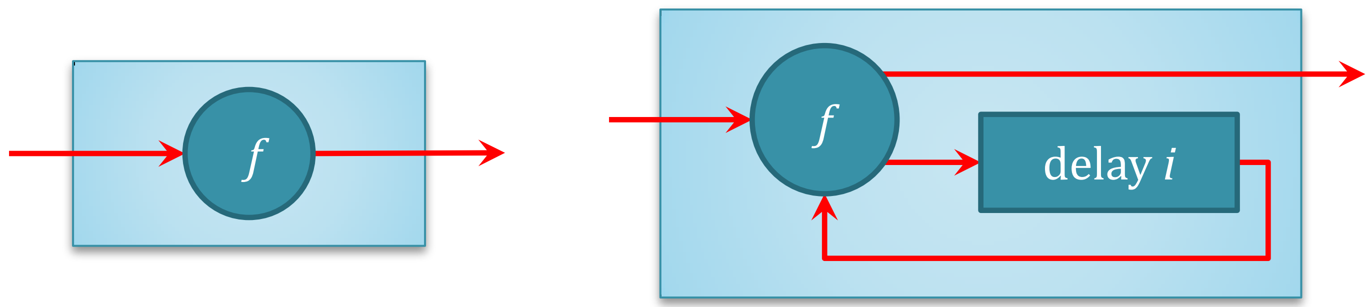
- Liu, Cheng, Hudak [2009] introduced CCA
  - CCAs can be heavily optimized
  - Performance increases of 10-40 times
  - CCAs allow choice but do not allow switch

Causal Commutative Arrows

- Liu, Cheng, Hudak [2009] introduced CCA
  - CCAs can be heavily optimized
  - Performance increases of 10-40 times
  - CCAs allow choice but do not allow switch
- CCAs can allow Non-interfering choice
  - Arrowized recursion is not supported by default, but it can be added

How CCA Works

- The CCA optimization reduces arrows to one of two forms:



- We extend this with the ability to handle arrowized recursion and call it CCA*

Performance Results

	GHC	CCA* + Stream
Chained Adder	1.0	4.06
Chained Integral	1.0	13.27
Dynamic Counters	1.0	10.91

- 3 sample programs using arrowized recursion
- 10x performance increase is comparable to Liu et al's results
  - Chained Adder is stateless, and thus more optimized by GHC

Summary

- Settability
  - New model for controlling FRP state
  - Ability to restart, pause, and duplicate signal functions while retaining a static structure

Summary

- Settability
  - New model for controlling FRP state
  - Ability to restart, pause, and duplicate signal functions while retaining a static structure
- Non-interfering choice
  - New forms of expression
  - Arrowized Recursion

Summary

- Settability
  - New model for controlling FRP state
  - Ability to restart, pause, and duplicate signal functions while retaining a static structure
- Non-interfering choice
  - New forms of expression
  - Arrowized Recursion
- Switch is **only** needed for true higher order expression

Thank you

Thank you

Questions?