Optimisation of Dynamic, Hybrid Signal Function Networks

Neil Sculthorpe and Henrik Nilsson

School of Computer Science

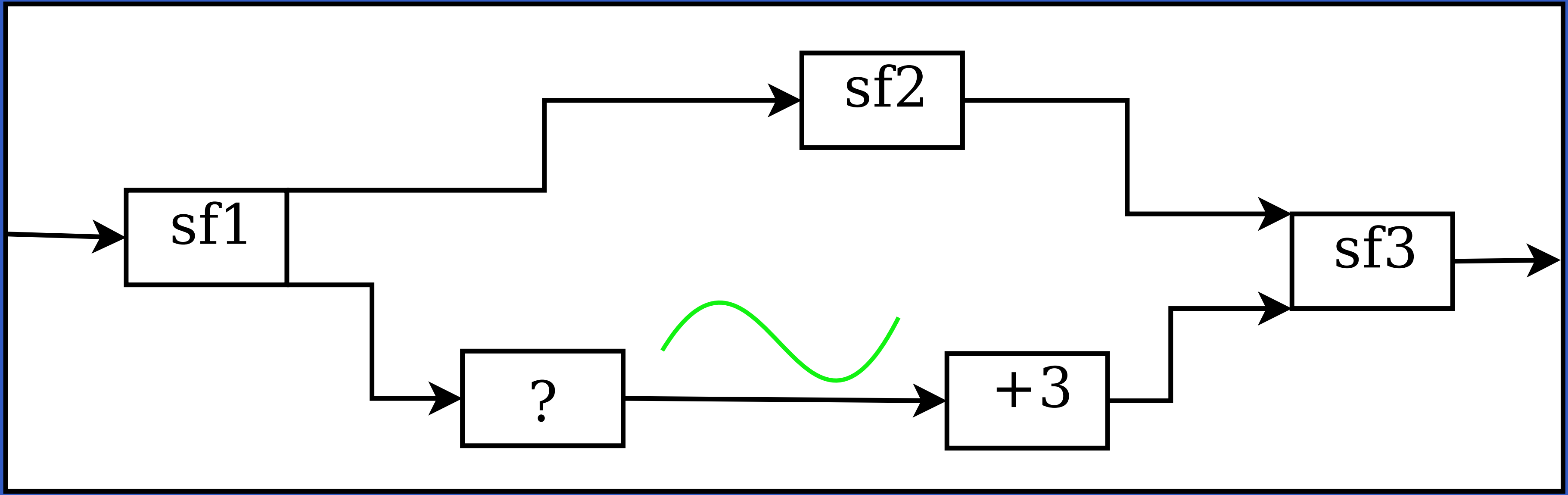
University of Nottingham, United Kingdom

Trends in Functional Programming, 27th May 2008

Outline

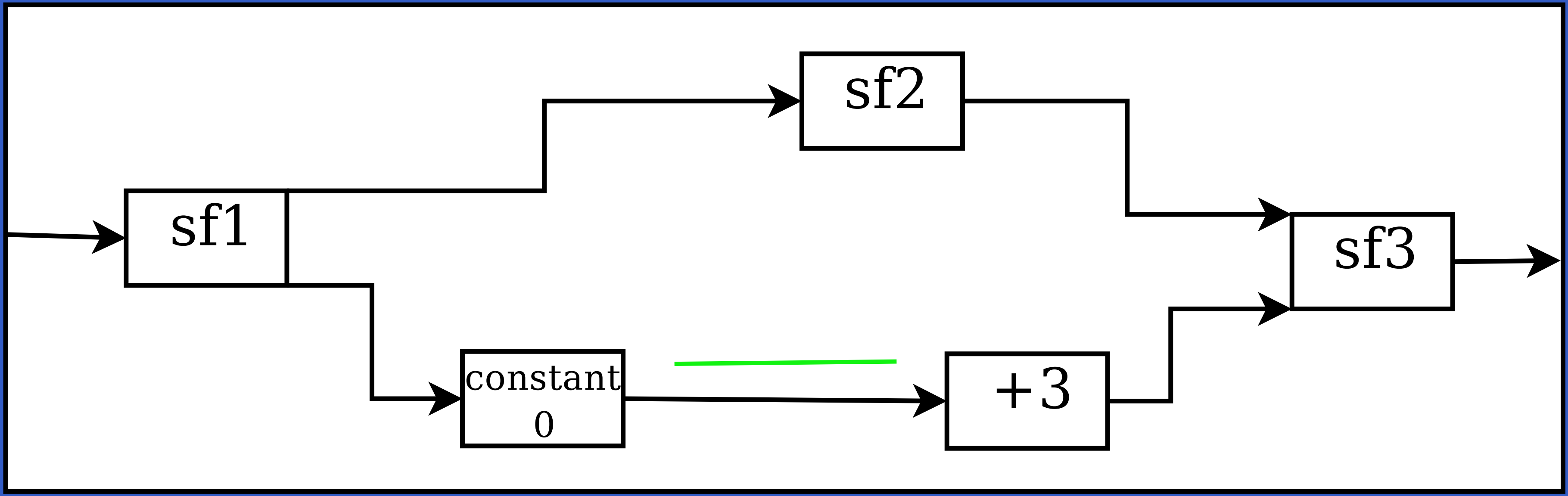
- A Brief Example
- Functional Reactive Programming and Yampa
- Our New Conceptual Framework
- A Notion of Change in that Framework
- Optimisation Examples

Example Signal Function Network



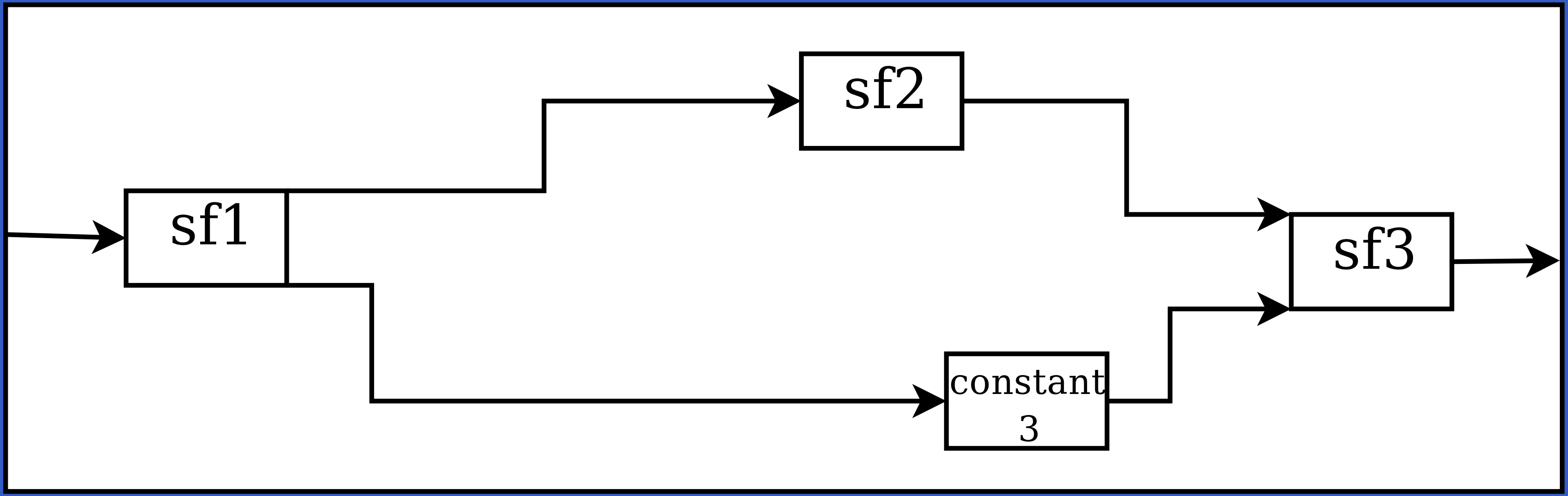
A synchronous data-flow network with hybrid and dynamic aspects.

Example Signal Function Network



A synchronous data-flow network with hybrid and dynamic aspects.

Example Signal Function Network



A synchronous data-flow network with hybrid and dynamic aspects.

Reactive Programming

- *Reactive program*: one that continually interacts with its environment, interleaving input and output in a *timely* manner.

Reactive Programming

- *Reactive program*: one that continually interacts with its environment, interleaving input and output in a *timely* manner.
- Examples include robot controllers, video games, and aeroplane control systems.

Reactive Programming

- *Reactive program*: one that continually interacts with its environment, interleaving input and output in a *timely* manner.
- Examples include robot controllers, video games, and aeroplane control systems.
- Contrast this with a *transformational program*: one that takes all input at the start of execution, and produces all output at the end (e.g. a compiler).

Reactive Programming

- *Reactive program*: one that continually interacts with its environment, interleaving input and output in a *timely* manner.
- Examples include robot controllers, video games, and aeroplane control systems.
- Contrast this with a *transformational program*: one that takes all input at the start of execution, and produces all output at the end (e.g. a compiler).
- Functional Reactive Programming (FRP) is a functional approach to reactive programming.

Yampa: A recent FRP Implementation

- *Signals* are time-varying values.

$$Signal\ a \approx Time \rightarrow a$$

Yampa: A recent FRP Implementation

- *Signals* are time-varying values.

$$Signal\ a \approx Time \rightarrow a$$

- *Signal Functions* are functions mapping signals to signals.

$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

Yampa: A recent FRP Implementation

- *Signals* are time-varying values.

$$Signal\ a \approx Time \rightarrow a$$

- *Signal Functions* are functions mapping signals to signals.

$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

- Signal functions can be *stateful* or *stateless*.

Yampa: Hybrid and Dynamic

- *Hybrid*: Conceptually both discrete-time (events) and continuous-time signals.

Yampa: Hybrid and Dynamic

- *Hybrid*: Conceptually both discrete-time (events) and continuous-time signals.
- In practice, events (conceptually sparse occurrences) are embedded in continuous-time signals.

$$\begin{aligned}\mathbf{data}\ \mathit{Event}\ a &= \mathit{NoEvent}\\ &\mid\ \mathit{Event}\ a\end{aligned}$$

Yampa: Hybrid and Dynamic

- *Hybrid*: Conceptually both discrete-time (events) and continuous-time signals.
- In practice, events (conceptually sparse occurrences) are embedded in continuous-time signals.

$$\begin{aligned} \mathbf{data}\ \mathit{Event}\ a &= \mathit{NoEvent} \\ &\mid\ \mathit{Event}\ a \end{aligned}$$

- *Dynamic*: The structure of the network can change at run-time.

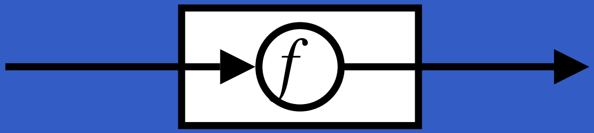
Yampa: Hybrid and Dynamic

- *Hybrid*: Conceptually both discrete-time (events) and continuous-time signals.
- In practice, events (conceptually sparse occurrences) are embedded in continuous-time signals.

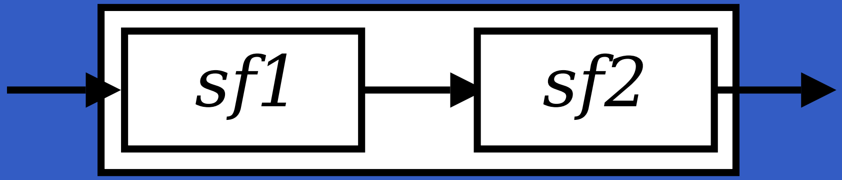
$$\begin{aligned}\textbf{data}\ \mathit{Event}\ a &= \mathit{NoEvent}\\ &\mid\ \mathit{Event}\ a\end{aligned}$$

- *Dynamic*: The structure of the network can change at run-time.
- Signal functions (not signals!) are first class.

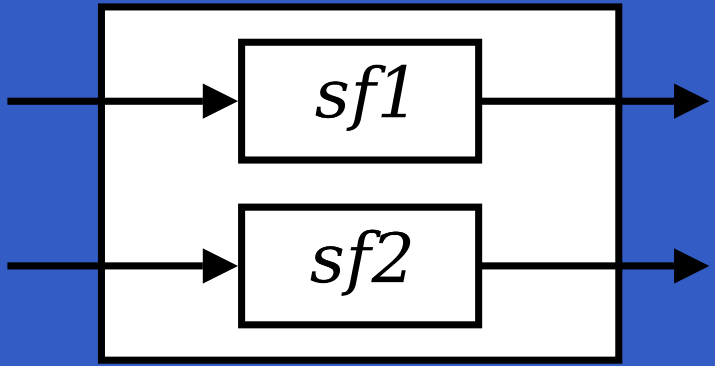
Some Yampa Primitives



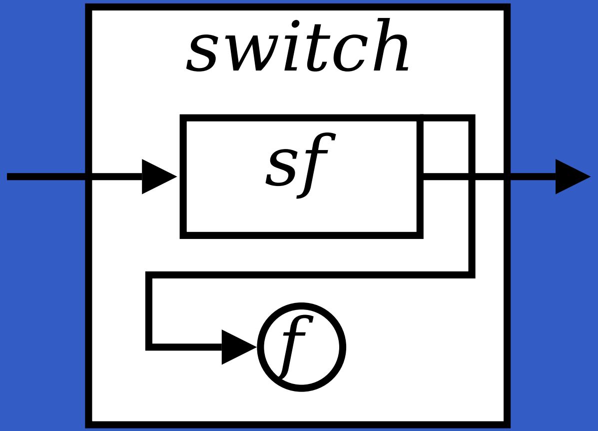
$$pure :: (a \rightarrow b) \rightarrow SF\ a\ b$$



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



$$(\ast\!\ast\!\ast) :: SF\ a\ c \rightarrow SF\ b\ d \rightarrow SF\ (a, b)\ (c, d)$$



$$switch :: SF\ a\ (b, Event\ e) \rightarrow (e \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

A New Conceptual Framework

- Our new framework better supports optimisation than Yampa.

A New Conceptual Framework

- Our new framework better supports optimisation than Yampa.
- We stress that this is a *conceptual* framework, not an actual implementation.

A New Conceptual Framework

- Our new framework better supports optimisation than Yampa.
- We stress that this is a *conceptual* framework, not an actual implementation.
- The key differences from Yampa are:

A New Conceptual Framework

- Our new framework better supports optimisation than Yampa.
- We stress that this is a *conceptual* framework, not an actual implementation.
- The key differences from Yampa are:
    - a type system that makes a precise distinction between *discrete-time* and *continuous-time* signals,

A New Conceptual Framework

- Our new framework better supports optimisation than Yampa.
- We stress that this is a *conceptual* framework, not an actual implementation.
- The key differences from Yampa are:
  - a type system that makes a precise distinction between *discrete-time* and *continuous-time* signals,
  - using (heterogeneous) vectors of signals instead of nested tuples.

Continuous and Discrete Time

We give a new conceptual definition of signals to make a clear distinction between continuous and discrete time.

$$\begin{aligned}
&\textbf{type}\ \mathit{CSignal}\ a \approx \mathit{Time} \rightarrow a\\
&\textbf{type}\ \mathit{ESignal}\ a \approx \mathit{Time} \rightarrow \mathit{Maybe}\ a
\end{aligned}$$

$$\begin{aligned}
\textbf{data}\ \mathit{Signal}\ a \quad &= C\ (\mathit{CSignal}\ a)\\
&\mid E\ (\mathit{ESignal}\ a)
\end{aligned}$$

Signal Vectors

Instead of tuples, we introduce *signal vectors* (a type level construct) to combine signals.

$$\begin{aligned} SigVec = \; & \langle\rangle \\ \mid \; & \langle C \;\; t\rangle \\ \mid \; & \langle E \;\; t\rangle \\ \mid \; & SigVec \mathbin{:\!\!+\!\!+\!\!:} SigVec \end{aligned}$$

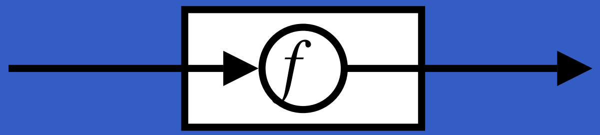
Signal Vectors

Instead of tuples, we introduce *signal vectors* (a type level construct) to combine signals.

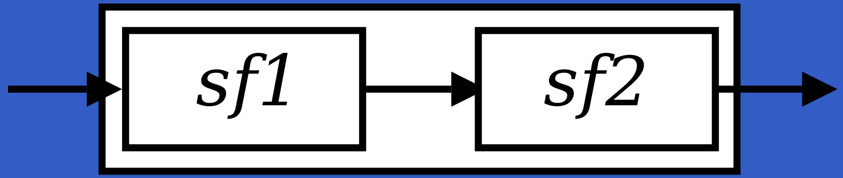
$$\begin{aligned} \mathit{SigVec} = \;& \langle\rangle \\ \mid\;& \langle C\ t\rangle \\ \mid\;& \langle E\ t\rangle \\ \mid\;& \mathit{SigVec} \mathbin{:\!\!+\!\!+\!\!:} \mathit{SigVec} \end{aligned}$$

$$\mathbf{type}\ (\mathit{SigVec}\ as, \mathit{SigVec}\ bs) \Rightarrow \mathit{SF}\ \ as\ \ bs \approx as \longrightarrow bs$$

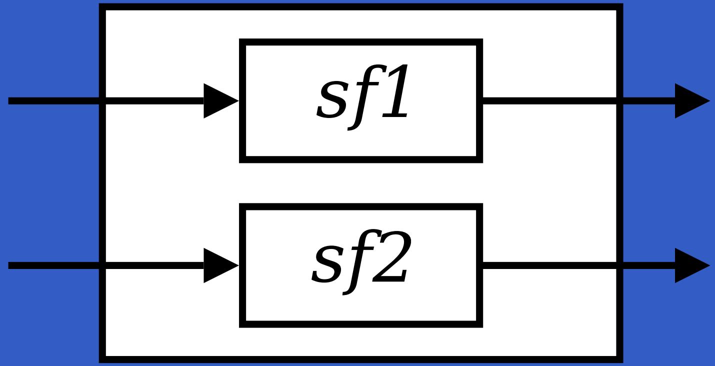
Primitives in this framework



$$pure :: (a \rightarrow b) \rightarrow SF\ \langle td\ a \rangle\ \ \langle td\ b \rangle$$

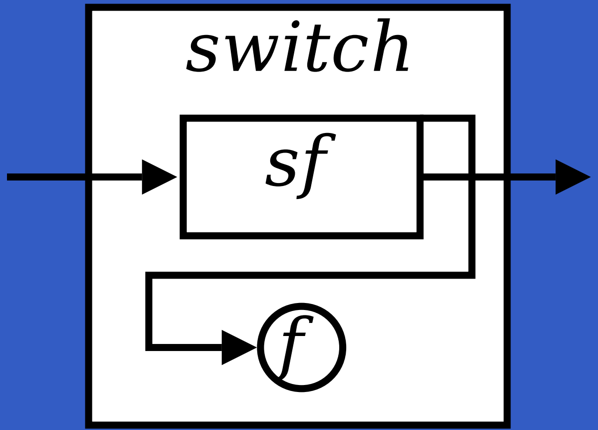


$$(\ggg) :: SF\ as\ bs \rightarrow SF\ bs\ cs \rightarrow SF\ as\ cs$$



$$(\ast\!\!\ast\!\!\ast) :: SF\ as\ cs \rightarrow SF\ bs\ ds \rightarrow$$

$$SF\ (as :\!+\!\!\!+\!: bs)\ (cs :\!+\!\!\!+\!: ds)$$



$$switch :: SF\ as\ (\ \langle E\ e \rangle\ :\!+\!\!\!+\!:\ bs) \rightarrow$$

$$(e \rightarrow SF\ as\ bs) \rightarrow SF\ as\ bs$$

Why is this useful for Optimisation?

- While we model time as continuous, at the implementation level a signal function network is executed over a discrete sequence of sample times.

Why is this useful for Optimisation?

- While we model time as continuous, at the implementation level a signal function network is executed over a discrete sequence of sample times.
- Many signal functions will produce the same output at many (if not all) sample times.

Why is this useful for Optimisation?

- While we model time as continuous, at the implementation level a signal function network is executed over a discrete sequence of sample times.
- Many signal functions will produce the same output at many (if not all) sample times.
- We would like to avoid re-computation of unchanged signals.

Change

We need a precise notion of change, which we define as follows:

Change

We need a precise notion of change, which we define as follows:

- A continuous-time signal is changing *iff* its value at the current time sample differs from its value at the preceding time sample.

Change

We need a precise notion of change, which we define as follows:

- A continuous-time signal is changing *iff* its value at the current time sample differs from its value at the preceding time sample.
- An event signal is changing *iff* there is an event occurrence at the current time sample.

Signal Function Change Classifications

- *Unchanging* (U) signal functions produce unchanging output.

Signal Function Change Classifications

- *Unchanging* (U) signal functions produce unchanging output.
- *Input-Dependent* (I) signal functions where unchanging input ⇒ unchanging output.

Signal Function Change Classifications

- *Unchanging* (U) signal functions produce unchanging output.

- *Input-Dependent* (I) signal functions where unchanging input ⇒ unchanging output.

- *Varying* (V) signal functions where the output may always change, regardless of input.

Examples

$$constant :: c \rightarrow SF_U\ as\ \langle C\ c\rangle$$

$$never :: SF_U\ as\ \langle E\ e\rangle$$

$$pure :: (a \rightarrow b) \rightarrow SF_I\ \langle td\ a\rangle\ \ \langle td\ b\rangle$$

$$edge :: SF_I\ \langle C\ Bool\rangle\ \ \langle E\ ()\rangle$$

$$iPre :: a \rightarrow SF_V\ \langle C\ a\rangle\ \ \langle C\ a\rangle$$

Combining Change Classifications

$$
\begin{aligned}
&(\ggg) && :: SF_x\ as\ bs \rightarrow SF_y\ bs\ cs \rightarrow SF_{(x\ \ggg\ y)}\ as\ cs\\
&(\ast\!\!\ast\!\!\ast) && :: SF_x\ as\ cs \rightarrow SF_y\ bs\ ds \rightarrow SF_{(x\ \sqcup\ y)}\ (as :\!+\!\!\!+\!: bs)\ (cs :\!+\!\!\!+\!: ds)\\
&\mathit{switch} && :: SF_x\ as\ (\langle E\ e\rangle :\!+\!\!\!+\!: bs) \rightarrow (e \rightarrow SF_y\ as\ bs) \rightarrow SF_{(x\ `sw`\ y)}\ as\ bs
\end{aligned}
$$

$$\mathbf{data}\ \mathit{ChangeClass} = U \mid I \mid V\ \mathbf{deriving}\ (\mathit{Eq}, \mathit{Ord})$$

$$
\begin{aligned}
&(\ggg) && :: \mathit{ChangeClass} \rightarrow \mathit{ChangeClass} \rightarrow \mathit{ChangeClass}\\
&x \ggg U && = U\\
&x \ggg V && = V\\
&x \ggg I && = x\\
&\mathit{sw} && :: \mathit{ChangeClass} \rightarrow \mathit{ChangeClass} \rightarrow \mathit{ChangeClass}\\
&U\ `sw`\ y && = U\\
&x\ `sw`\ y && = x \sqcup y
\end{aligned}
$$

Some Useful Optimisation Properties

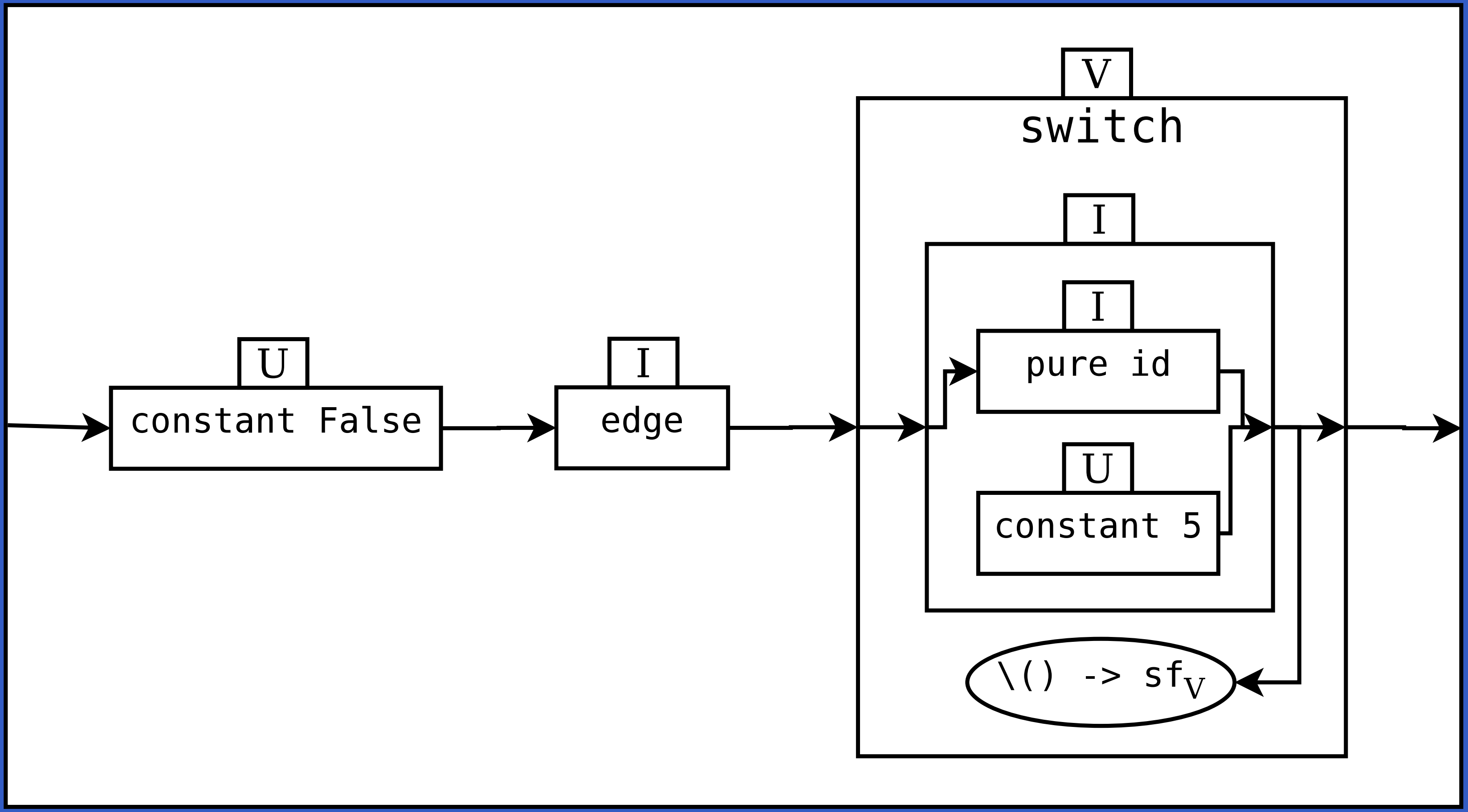
- Any composite *unchanging* signal function can be compressed into a single signal function.

Some Useful Optimisation Properties

- Any composite *unchanging* signal function can be compressed into a single signal function.
- Unchanging signal functions distribute into switches over sequential composition:

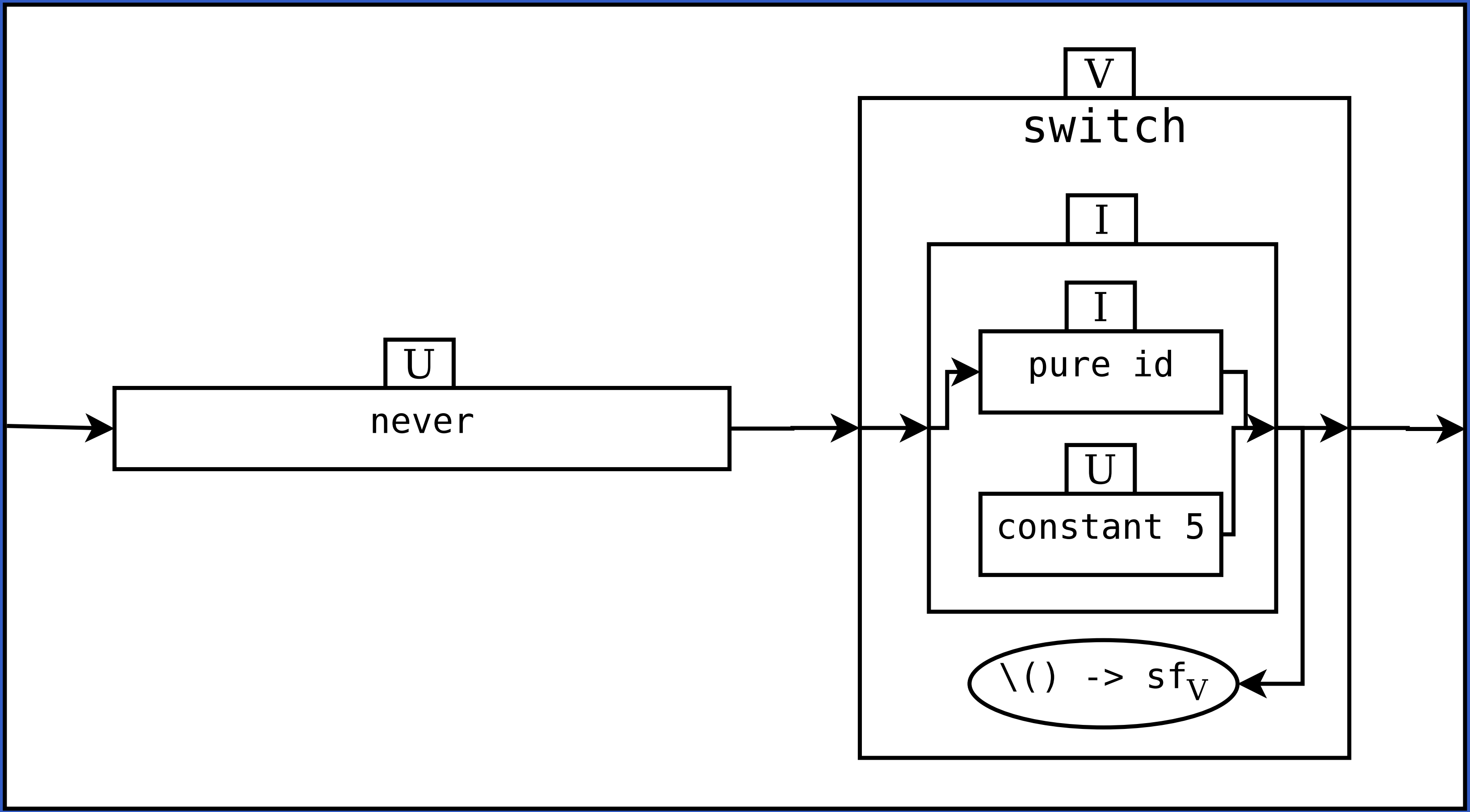
$$
\begin{array}{l}
\quad sf1_U \ggg switch\ sf2\ f \\
\equiv \\
\quad switch\ (sf1_U \ggg sf2)\ (\lambda a \rightarrow sf1_U \ggg f\ a)
\end{array}
$$

Example Optimisation



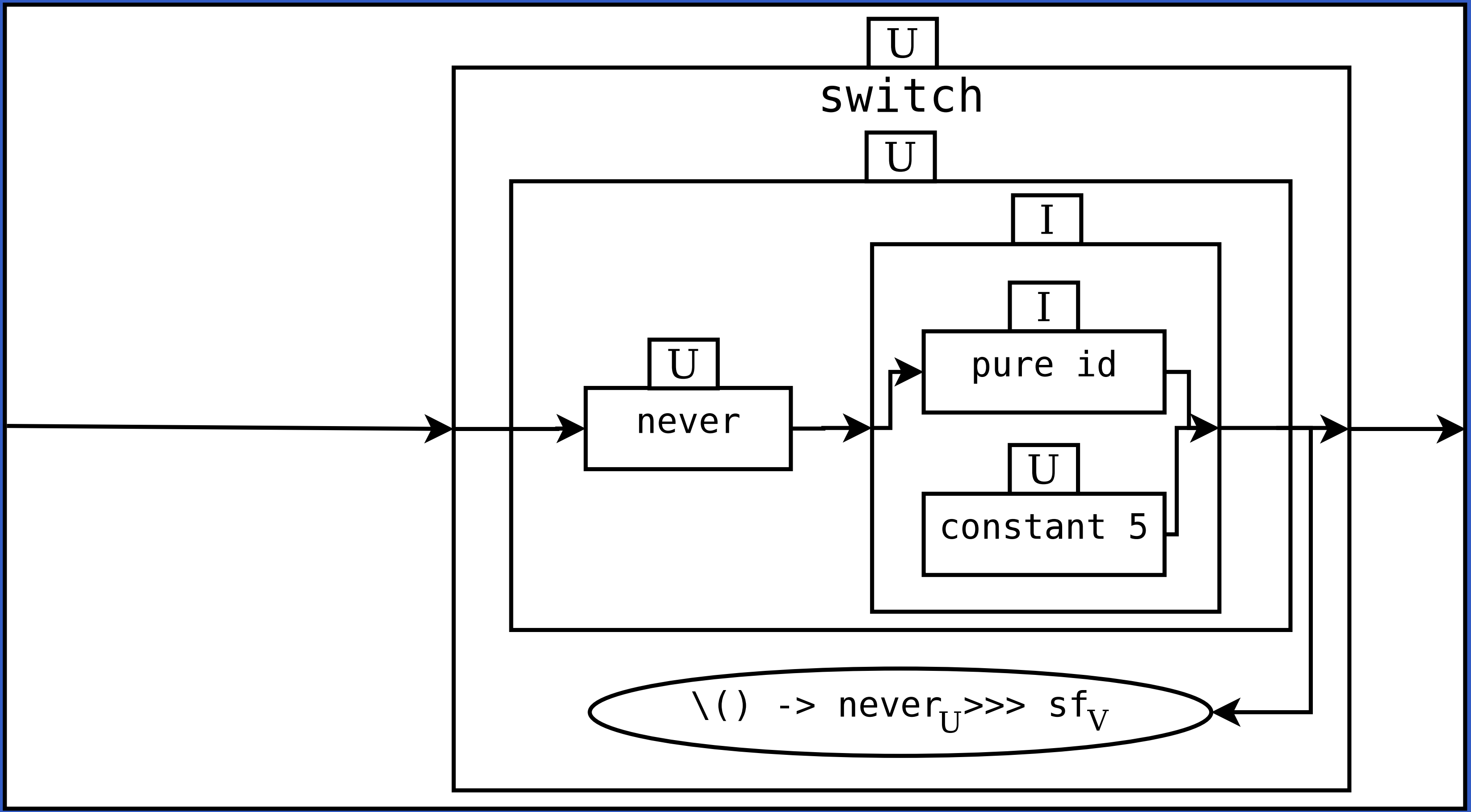
$$\mathit{constant}\ \mathit{False} \ggg \mathit{edge} \ggg \mathit{switch}\ (\mathit{pure}\ \mathit{id} \mathbin{*\!*\!*} \mathit{constant}\ 5)\ (\lambda() \rightarrow \mathit{sf})$$

Example Optimisation



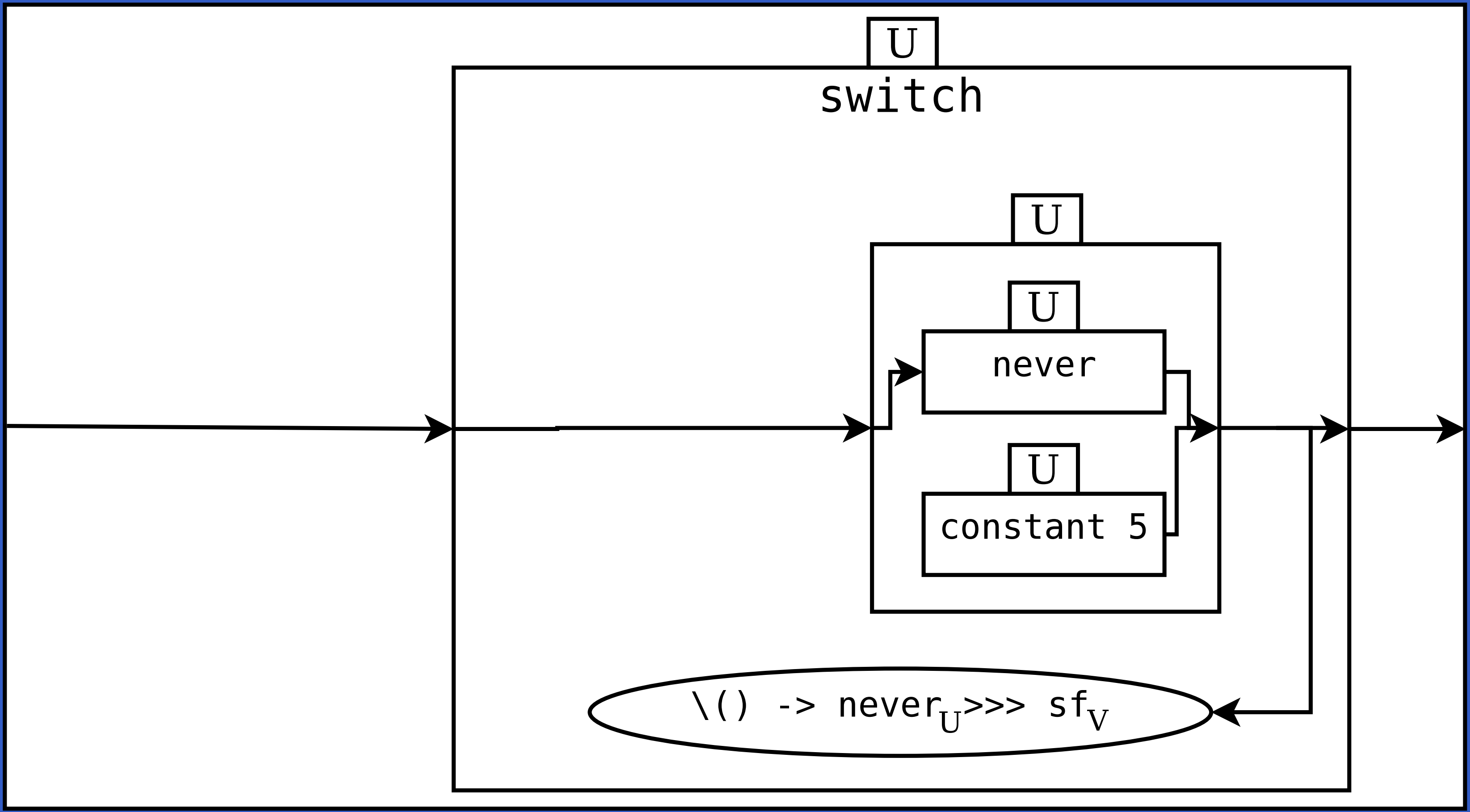
$$\mathit{never} \ggg \mathit{switch}\ (\mathit{pure}\ \mathit{id} \ast\!\!\ast\!\!\ast\ \mathit{constant}\ 5)\ (\lambda() \rightarrow \mathit{sf}\,)$$

Example Optimisation



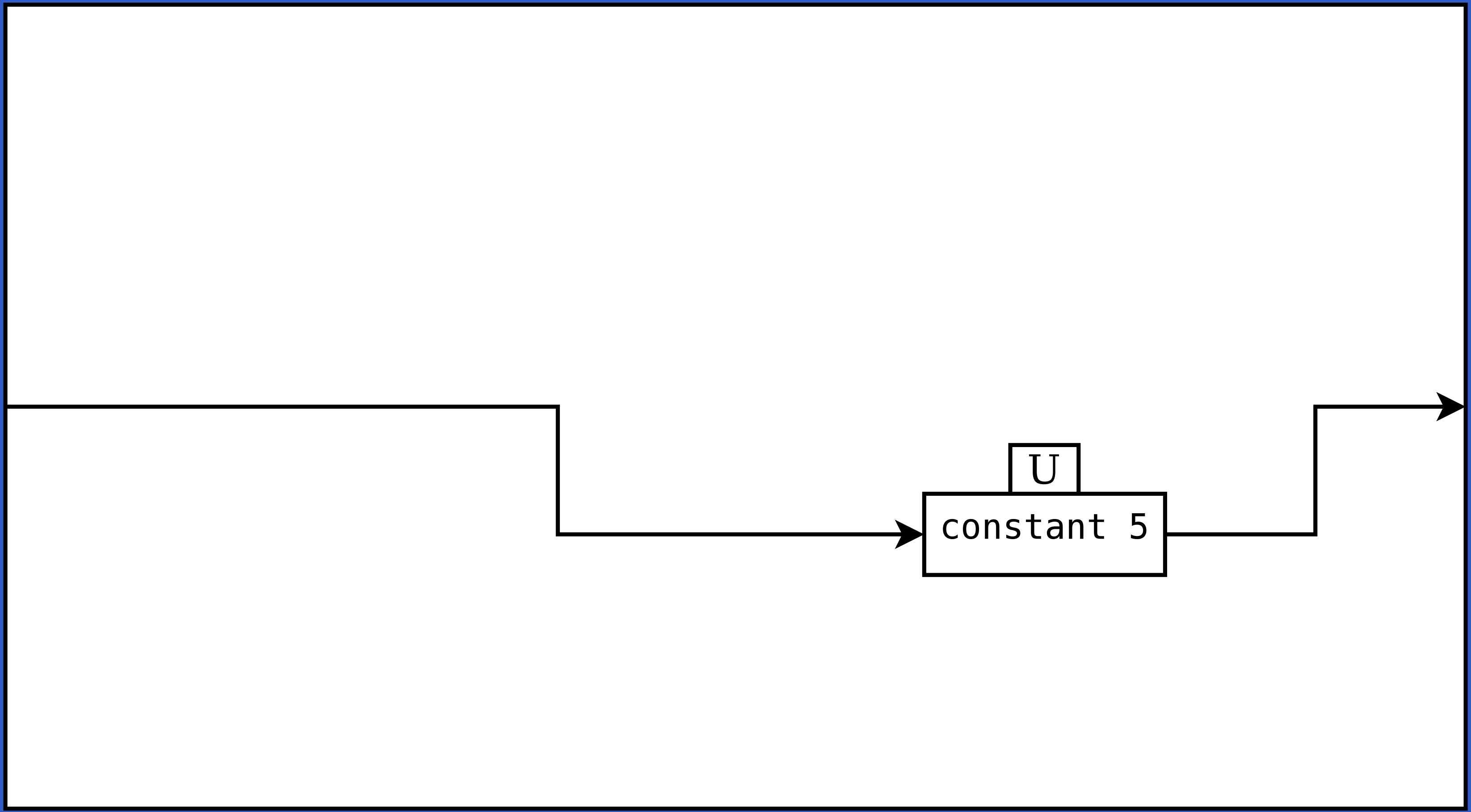
$$switch\ (never \ggg pure\ id \ \&\&\&\ constant\ 5)\ (\lambda() \rightarrow never \ggg sf)$$

Example Optimisation



$$switch\ (never \mathbin{\&\&\&} constant\ 5)\ (\lambda() \rightarrow sf)$$

Example Optimisation



$$constant\ 5$$

Summary

- We have discussed:
  - Our conceptual framework
  - A notion of change within that framework
  - Some optimisations that exploit that notion of change

Summary

- We have discussed:
  - Our conceptual framework
  - A notion of change within that framework
  - Some optimisations that exploit that notion of change
- Our framework also supports run-time *change propagation* optimisations.