Improving Push-based FRP

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Ninth Symposium on Trends in Functional Programming
May 26–28, 2008

Functional Reactive Programming

- Functional Reactive Programming (FRP) for declarative programming of reactive/interactive systems
- Key concept: the signal
- two flavors of signals: discrete and continuous
- in this talk only data flow aspects (in particular, no continuous signals)
- push-based implementation to avoid unnecessary recomputations

Key problems addressed in this talk

H 10:40 M

- a simple clock application as an example
- key problems:
  - If an hour is completed, also a minute is completed. Both completion events should be combined to induce a single display update.
    - no performance loss
    - no inconsistent intermediate states
  - The input of the display depends on the output of the minute button which comes after the display. We need support for feedbacks to realize this.

Circuits

- interaction with the outside world via circuits
  - arrows with signal tuples as inputs and outputs
- the core of the clock application:

```
clockApp :: Circuit () ()
clockApp = proc () → do
  rec let
        ...
      mPulse ← minutePulse    ⊣ ()
      hPress ← button "H"     ⊣ empty
      ()     ← label "00:00"  ⊣ ...
      mPress ← button "M"     ⊣ empty
  returnA ⊣ ()
```

Construction of an update signal (part 1)

- counting:

$$count :: DSignal\ val \rightarrow DSignal\ Int$$
$$count = scan\ (\lambda num\ \_ \rightarrow succ\ num)\ 0$$

- hour pulse:

$$hPulse :: DSignal\ Int$$
$$hPulse = filter\ (\lambda m \rightarrow m\ `mod`\ 60 \equiv 0)\ (count\ mPulse)$$

Construction of an update signal (part 2)

- hour und minute updates:

$$
\begin{aligned}
&\mathit{update} :: \mathit{DSignal}\ (\mathit{MergeVal}\ \mathit{Int}\ \mathit{Int}) \\
&\mathit{update} = \mathit{merge}\ (\mathit{count}\ (\mathit{hPulse}\ `\mathit{merge}`\ \mathit{hPress})) \\
&\qquad\qquad\qquad\qquad (\mathit{count}\ (\mathit{mPulse}\ `\mathit{merge}`\ \mathit{mPress})) \\
&\mathbf{data}\ \mathit{MergeVal}\ \mathit{val}_1\ \mathit{val}_2 = \mathit{First}\ \mathit{val}_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathit{Second}\ \mathit{val}_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathit{Both}\ \mathit{val}_1\ \mathit{val}_2
\end{aligned}
$$

- Values are only simultaneous if induced by the same external event.
  - inner *merge* applications do not combine occurences
  - outer *merge* application combines hour pulse updates with minute pulse updates

The key idea for handling simultaneity

- Remember: Values are only simultaneous if induced by the same external event.
- One and the same event arises from one and the same source.
- Key idea: differentiation according to event sources
- type *DSource* (discrete source) for describing event sources

Implementation of *DSignal*

- generator generates occurence or non-occurence depending on references to mutable variables holding accumulators

$$\textbf{type } \mathit{Gen\ locals\ val} = \mathit{locals} \rightarrow \mathit{IO}\ (\mathit{Maybe\ val})$$

- generators are assigned to sources whose events trigger their execution

$$\textbf{type } \mathit{GenMap\ locals\ val} = \mathit{Map\ DSource}\ (\mathit{Gen\ locals\ val})$$

- merging of signals combines generators for the same source (using *unionWith*)
- signal covers locals creation action and generator map

$$\begin{aligned}\textbf{data } \mathit{DSignal\ val} = \forall \mathit{locals}.\ & \\ & \mathit{DSignal}\ (\mathit{IO\ locals}) \\ & \quad (\mathit{GenMap\ locals\ val})\end{aligned}$$

Basic circuit implementation

- Idea: A circuit is executed by just running an event loop.
- first approach: A circuit is an initialization action which outputs the corresponding finalization action.

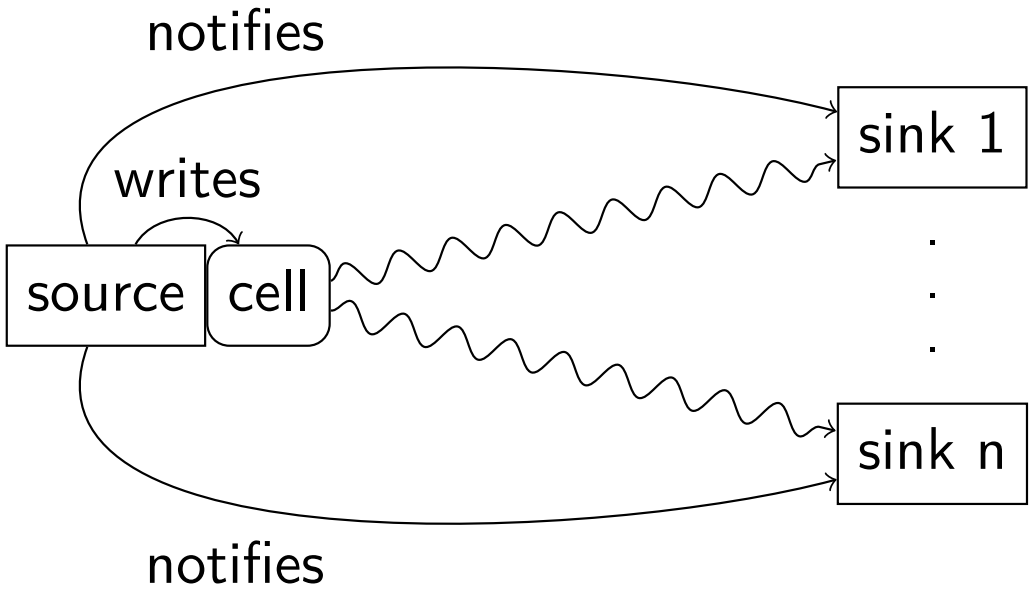
$$\mathit{WriterArrow}\ (\mathit{IO}\ ())\ (\mathit{Kleisli}\ \mathit{IO})\ i\ o$$

Problem: no feedbacks (registration before source exists)

- modification: Handler registration is output for later execution (and outputs the finalization action).

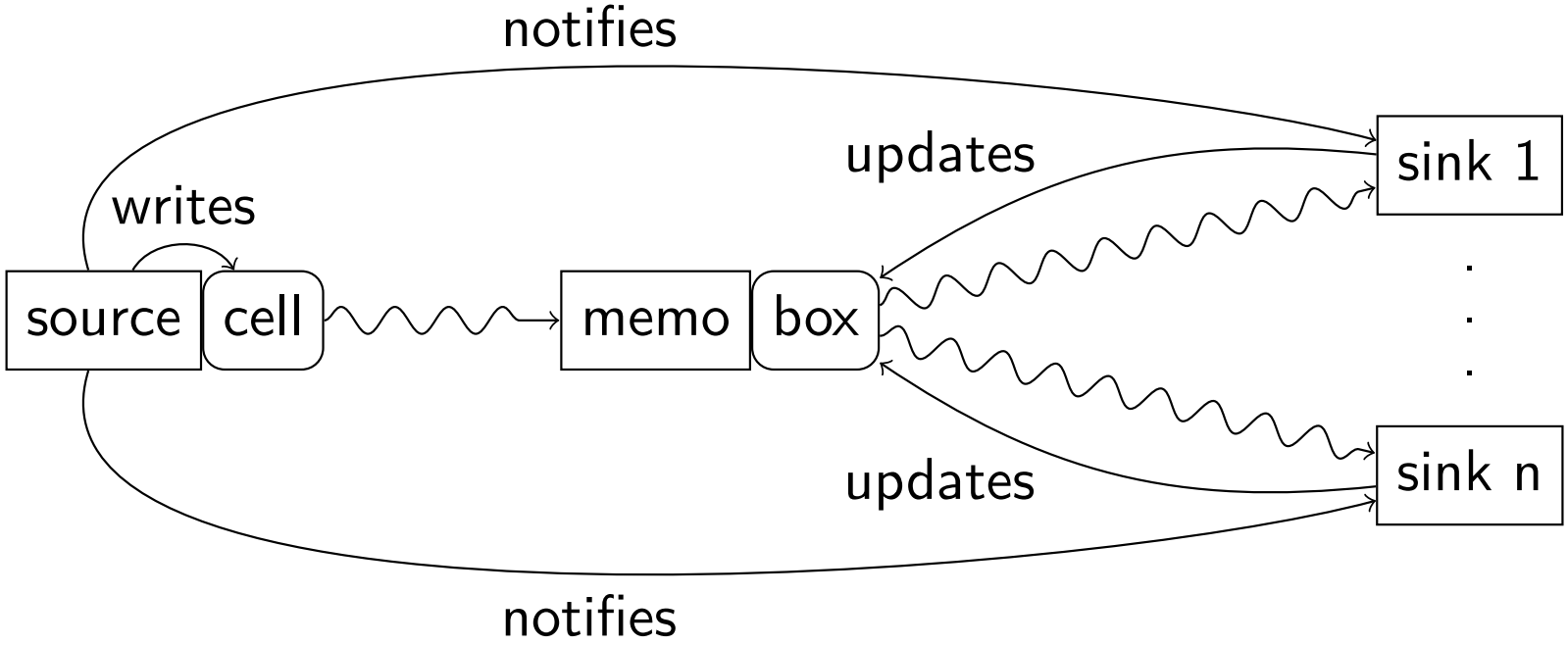
$$\mathit{WriterArrow}\ (\mathit{IO}\ (\mathit{IO}\ ()))\ (\mathit{Kleisli}\ \mathit{IO})\ i\ o$$

Wasting space and time



- Problem: Every sink registers its own handlers.
  - state is stored once per sink (space penalty)
  - occurences/non-occurences are generated and state is updated once per sink (time penalty)

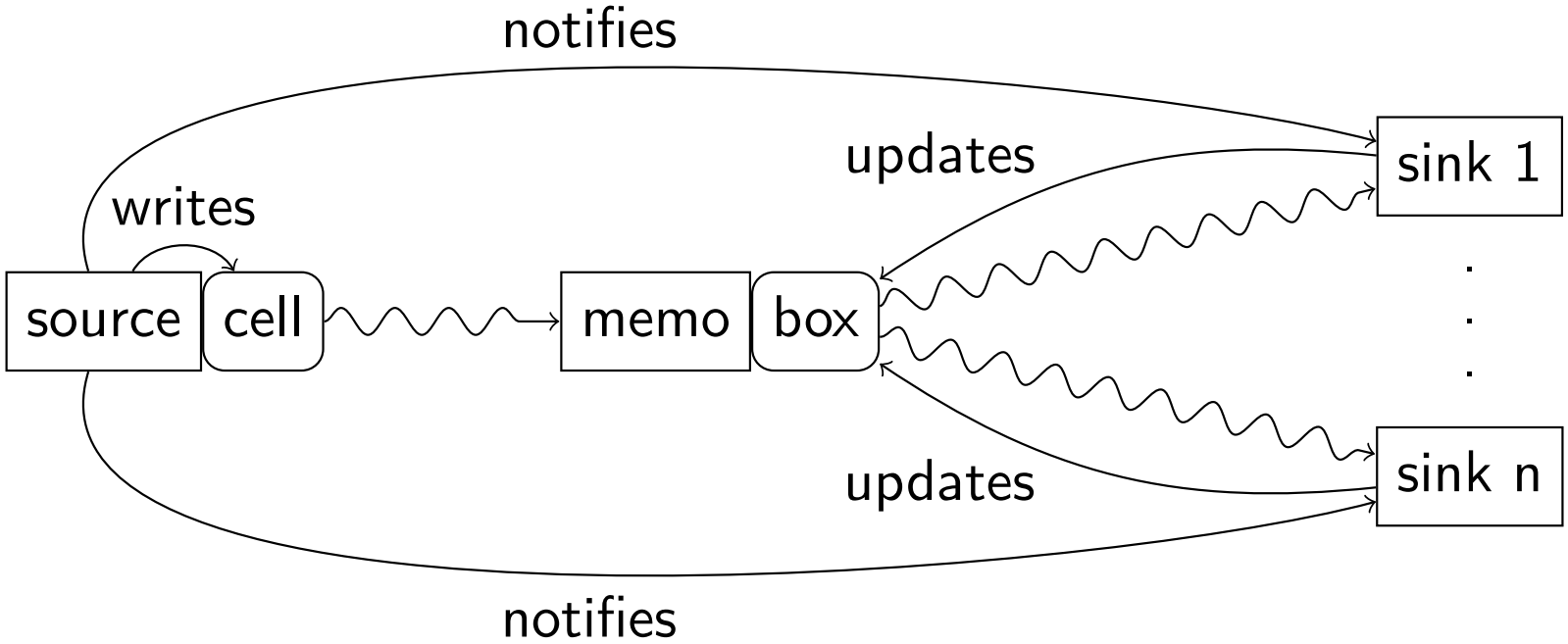
Memoization



- Solution: memoization circuit

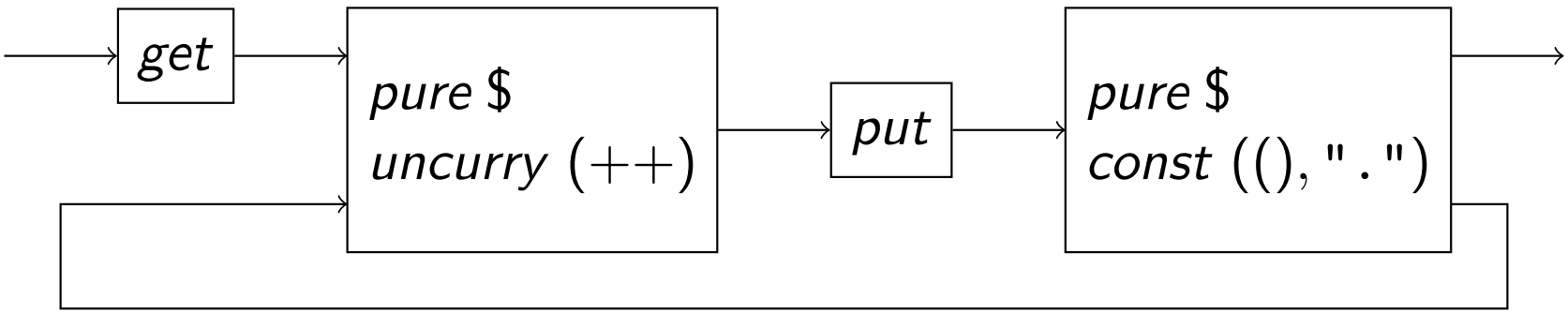
$$memo :: Circuit\ (DSignal\ val)\ (DSignal\ val)$$

Memoization breaking feedbacks



- Problem: Memoization breaks feedbacks.
- Interesting: Locals creation does not depend on I/O.

No earlier use of pure values in the *IO* monad



- example *Kleisli IO* arrow for illustrating the problem:

$$
\begin{aligned}
&\mathit{loop}\ \$\ \mathit{first}\ \mathit{get} \ggg \\
&\qquad \mathit{pure}\ (\mathit{uncurry}\ (++)) \ggg \\
&\qquad \mathit{put} \ggg \\
&\qquad \mathit{pure}\ (\mathit{const}\ ((), "."))
\end{aligned}
$$

- *put* will fail.

Introducing lax arrows

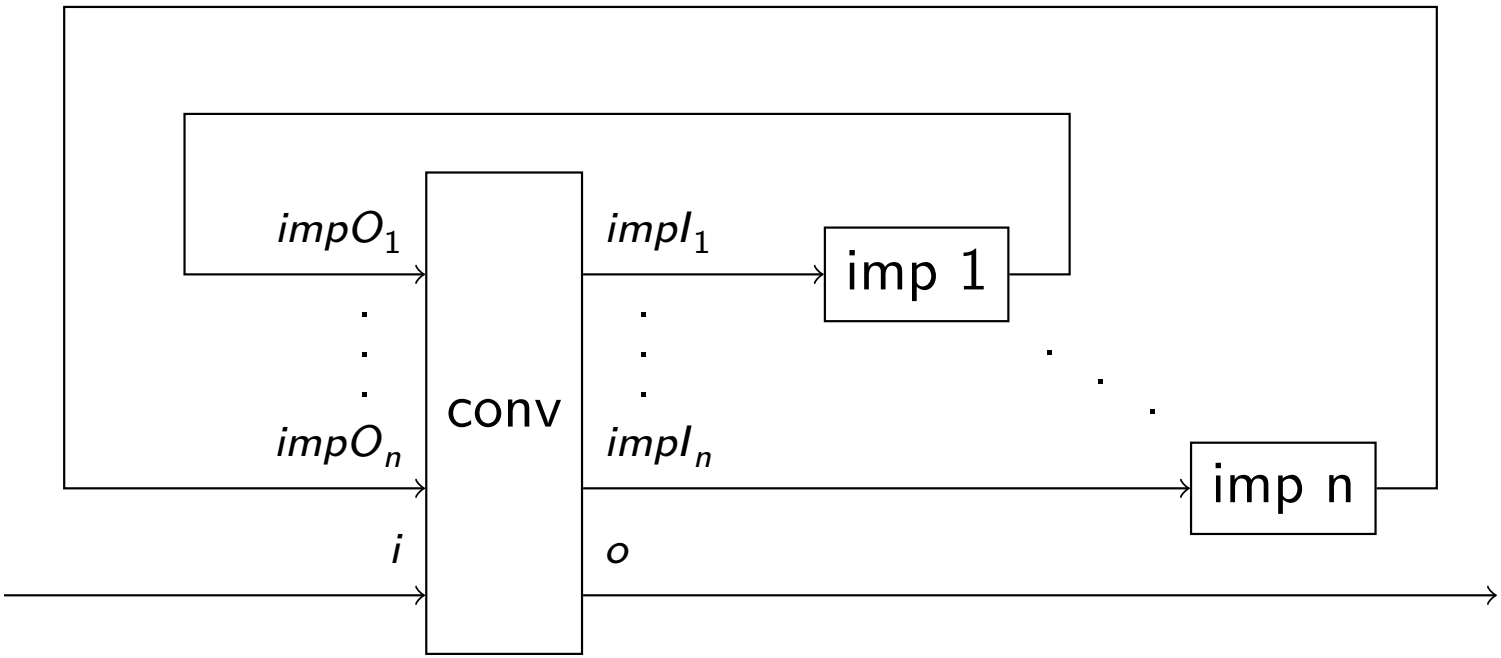
- Solution: moving all pure parts to the beginning of the arrow automatically
- done with lax arrows
- approach:
  1. create impure particles

$$impure :: (ArrowLoop\ base) \Rightarrow \\ base\ i\ o \rightarrow LaxArrow\ base\ i\ o$$

  2. compose them using *LaxArrow* implementations of *Arrow* and *ArrowLoop* methods
  3. convert the resulting lax arrow value into a base arrow value

$$runLax :: (Arrow\ base) \Rightarrow \\ LaxArrow\ base\ i\ o \rightarrow base\ i\ o$$

Structure of *runLax* results



- Purly functional computations are done at the beginning.
- Impure particles follow.
- The outputs of impure particles are immediately fed back into the converter.

IO feedback example using lax arrows

- modification of our feedback example using lax arrows:

$$
\begin{aligned}
&\mathit{runLax}\ \$\\
&\mathit{loop}\ \$\ \mathit{first}\ (\mathit{impure}\ \mathit{get}) \ggg\\
&\qquad \mathit{pure}\ (\mathit{uncurry}\ (++)) \ggg\\
&\qquad \mathit{impure}\ \mathit{put} \ggg\\
&\qquad \mathit{pure}\ (\mathit{const}\ ((), \text{"."}))
\end{aligned}
$$

- This is equivalent to this:

$$\mathit{loop}\ (\mathit{loop}\ (\mathit{pure}\ \mathit{conv} \ggg \mathit{second}\ \mathit{get}) \ggg \mathit{second}\ \mathit{put})$$

with *conv* defined like this:

$$\mathit{conv}\ \sim(\sim(i, \mathit{putO}), \mathit{getO}) = (((), \mathit{getO} ++ \text{"."}), i)$$

Improving Push-based FRP

Wolfgang Jeltsch

Brandenburgische Technische Universität Cottbus
Cottbus, Germany

Ninth Symposium on Trends in Functional Programming
May 26–28, 2008

Signal functions

- some primitive signal functions:

$$
\begin{aligned}
&\mathit{empty} \;::\; \mathit{DSignal}\ \mathit{val}\\
&\mathit{scan} \;::\; (\mathit{val}' \to \mathit{val} \to \mathit{val}') \to \mathit{val}'\\
&\qquad \to (\mathit{DSignal}\ \mathit{val} \to \mathit{DSignal}\ \mathit{val}')\\
&\mathit{filter} \;::\; (\mathit{val} \to \mathit{Bool}) \to (\mathit{DSignal}\ \mathit{val} \to \mathit{DSignal}\ \mathit{val})\\
&\mathit{merge} \;::\; \mathit{DSignal}\ \mathit{val}_1 \to \mathit{DSignal}\ \mathit{val}_2\\
&\qquad \to \mathit{DSignal}\ (\mathit{MergeVal}\ \mathit{val}_1\ \mathit{val}_2)\\
&\textbf{data}\ \mathit{MergeVal}\ \mathit{val}_1\ \mathit{val}_2 = \mathit{First}\ \mathit{val}_1\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathit{Second}\ \mathit{val}_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid\ \mathit{Both}\ \mathit{val}_1\ \mathit{val}_2
\end{aligned}
$$

Discrete sources

- type *DSource* (discrete source) for describing event sources

$$\textbf{data}\ \mathit{DSource} = \mathit{DSource}\ \mathit{Unique}\ \mathit{Notifier}$$
$$\textbf{type}\ \mathit{Notifier} = \mathit{IO}\ () \rightarrow \mathit{IO}\ :\$\ \mathit{IO}\ :\$\ ()$$

  - unique identifier for equality test and use as map keys
  - notifier being an action registering event handlers

LaxArrow implementation (basic idea)

- Idea: lax arrow internally consists of
  - the converter function

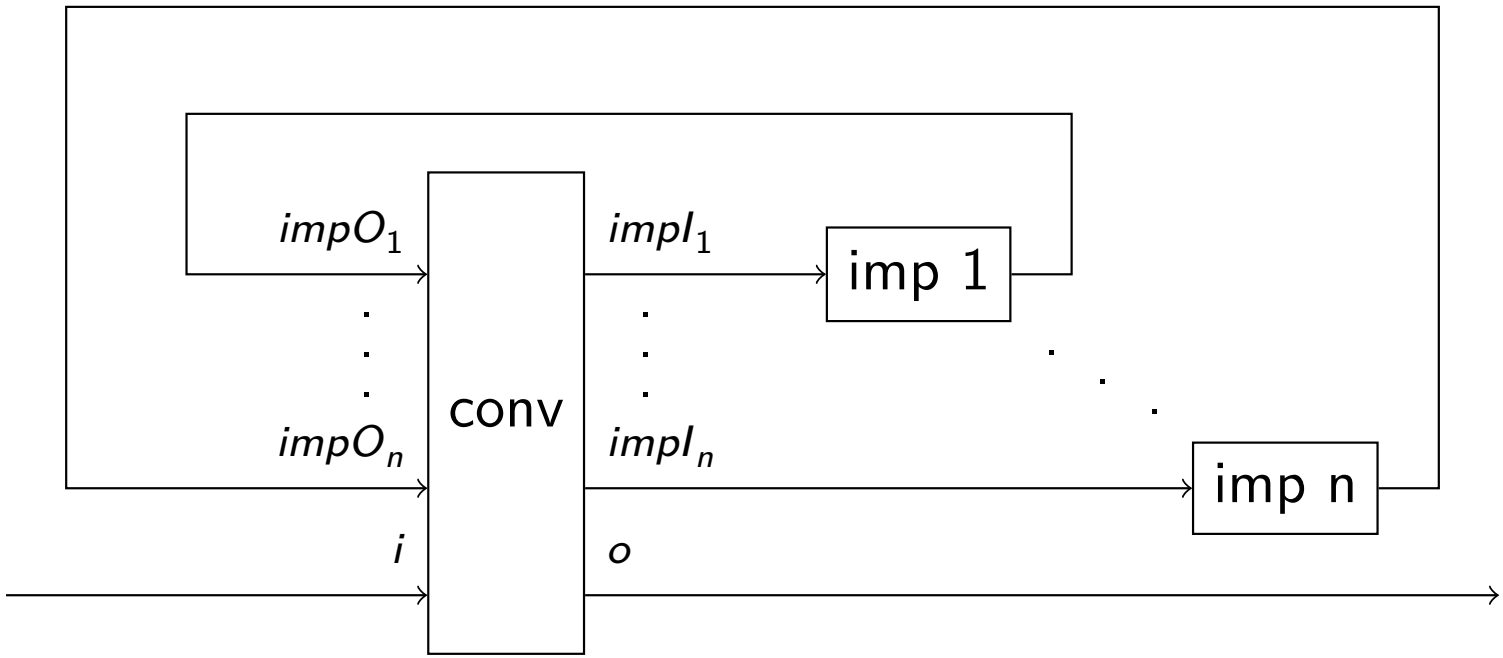
$$(impO, i) \longrightarrow (impI, o)$$

  - a function transforming the converter into the *runLax* result:

$$base\ (impO, i)\ (impI, o) \longrightarrow base\ i\ o$$

- makes implementing *runLax* very easy
- has to be generalized a bit to make $(\ggg)$ implementation possible

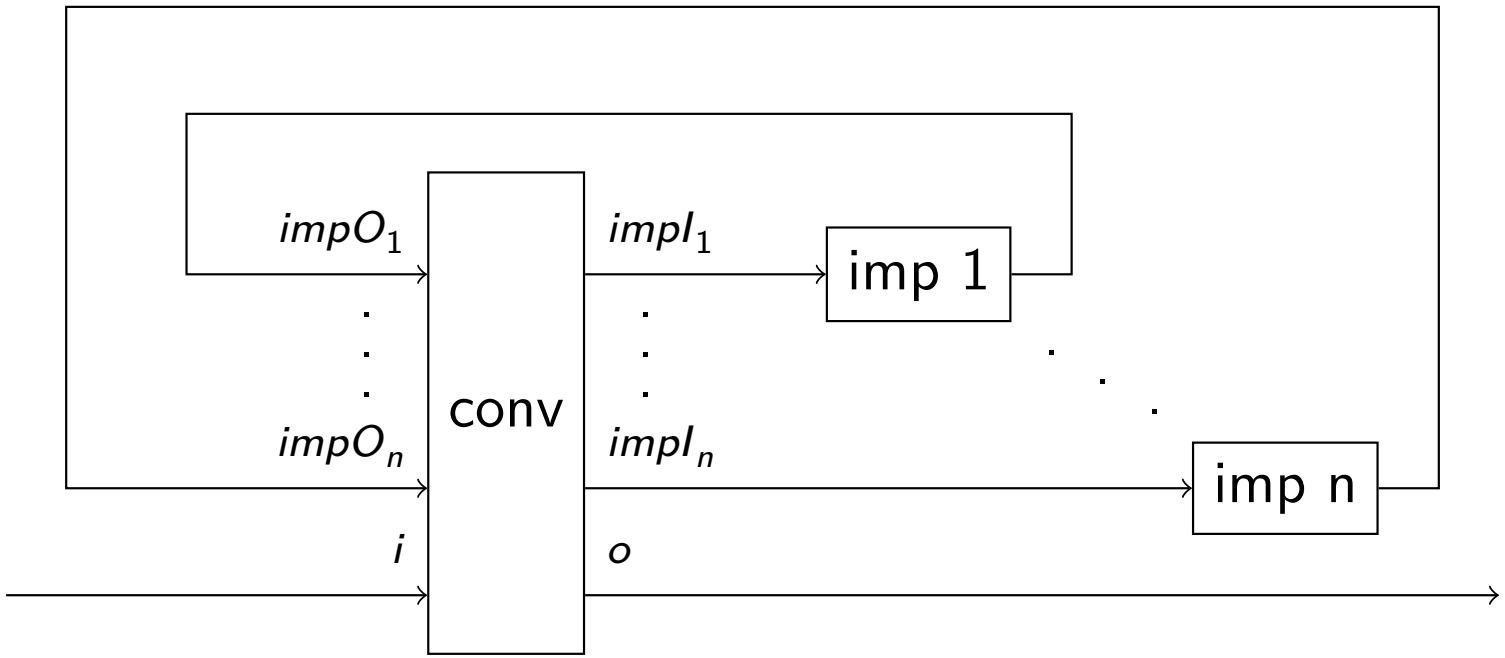
Implementation of *pure* (modulo retupling)



converter function is the argument of *pure*

base generator is the identity

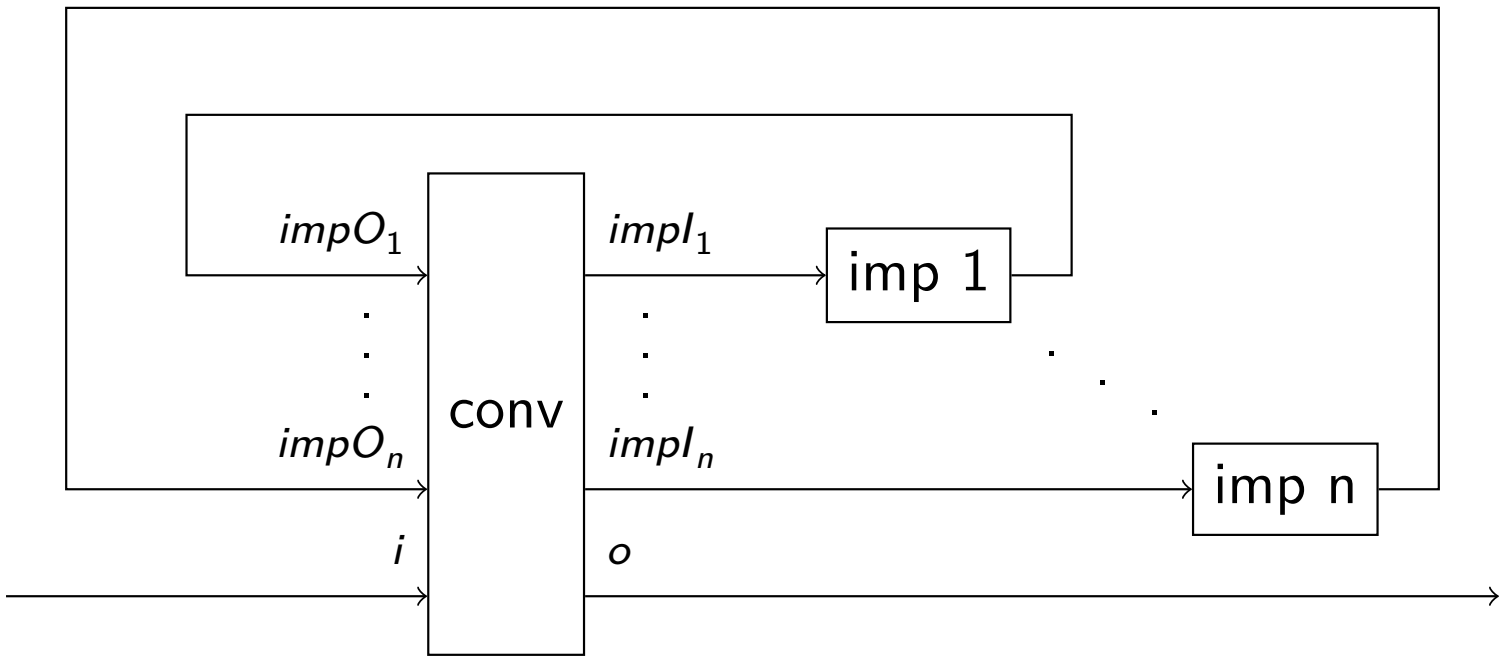
Implementation of (⋙) (modulo retupling)



converter function splits impure outputs, joins impure inputs and transfers first arrow's output to second arrow's input

base generator is the composition of both arrow's base generators

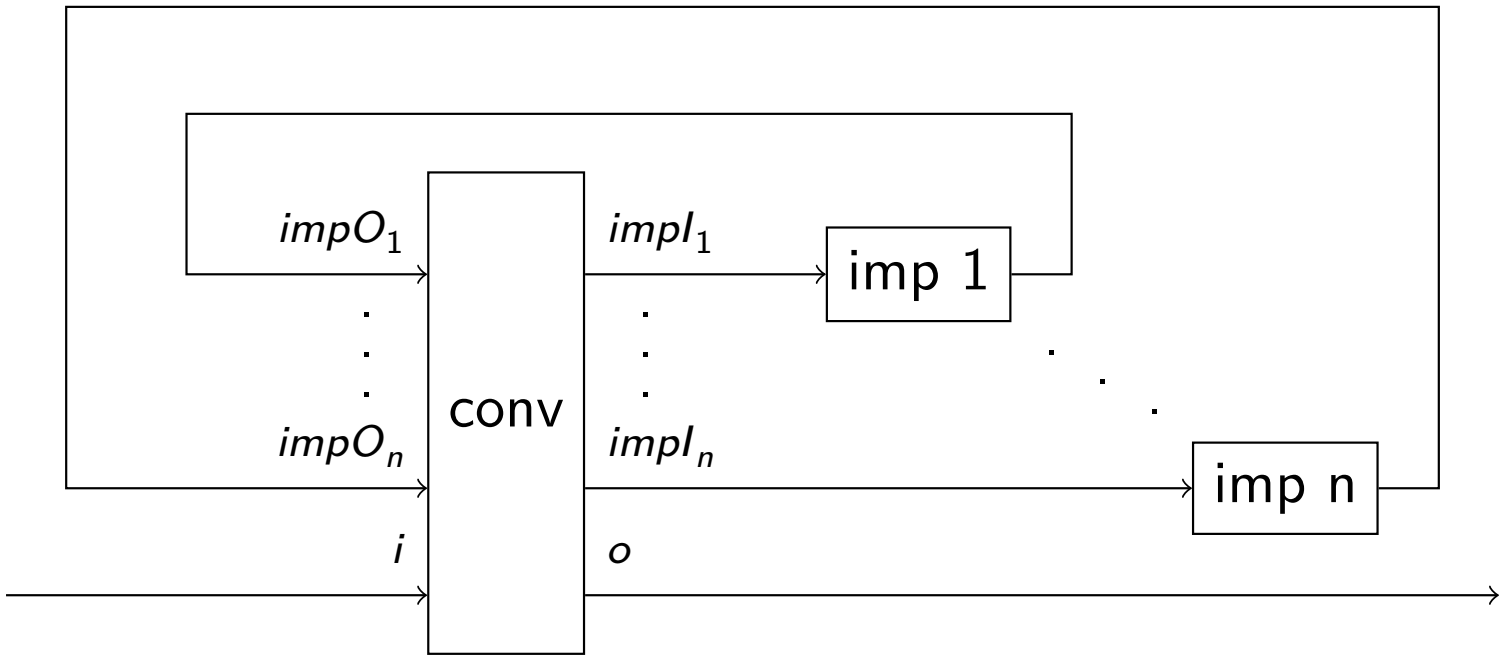
Implementation of *first* and *loop* (modulo retupling)



converter function does the respective wiring

base generator is the argument's base generator

Implementation of *impure* (modulo retupling)



converter function adds the cycle containing the impure particle

base generator feeds input to the impure input and the impure output to the output