The ForSyDe Standard Library\(^1\)

I. Sander  A. Jantsch  A. K. Singh
T. Raudvere
Royal Institute of Technology, Sweden

November 19, 2001

\(^1\)Version 2.0

## Contents

1 Introduction 4  
  1.1 Overview .................................. 4  

2 The module ForSyDeStdLib 5  
  2.1 Overview .................................. 5  

3 The module AbsentExt 6  
  3.1 Overview .................................. 6  
  3.2 Functions on the data type AbsentExt ............. 6  
  3.3 Implementation of library functions ............... 6  

4 The module Signal 8  
  4.1 Overview .................................. 8  
  4.2 The data type Signal ......................... 8  
  4.3 Functions on the data type Signal ............... 8  
  4.4 Implementation of the library functions .......... 9  
      4.4.1 Overloading of Show and Read for the data type Signal 9  
      4.4.2 Functions on the data type Signal ........ 10  

5 The module SynchronousLib 11  
  5.1 Overview .................................. 11  
  5.2 Skeletons for Combinatorial Processes .......... 11  
  5.3 Skeletons for Sequential Processes .......... 12  
  5.4 Processes .................................. 13  
  5.5 Implementation of Library Functions .......... 13  
      5.5.1 Skeletons for Combinatorial Processes .......... 13  
      5.5.2 Skeletons for Sequential Processes .......... 14  
      5.5.3 Processes .................................. 15  

6 The module SynchronousProcessLib 17  
  6.1 Overview .................................. 17  
  6.2 Processes .................................. 17  
  6.3 Implementation of Processes .............. 18  

7 The module DataflowLib 19  
  7.1 Overview .................................. 19  
  7.2 Data Types .................................. 19  
  7.3 Skeletons .................................. 20  
      7.3.1 Skeletons for Combinatorial Processes .......... 20  
      7.3.2 Skeletons for Sequential Processes .......... 20  
  7.4 Implementation .................................. 21  
      7.4.1 Combinatorial Skeletons .......... 21  
      7.4.2 Sequential Skeletons ............ 22
7.4.3 Supporting Functions .................................. 23

8 The module UntimedLib ...................................... 26
  8.1 Overview .................................................. 26
  8.2 Skeletons .................................................. 26
    8.2.1 Skeletons for Combinatorial Processes .......... 26
    8.2.2 Skeletons for Sequential Processes .......... 26
    8.2.3 Zip and Unzip based Skeletons ................. 27
    8.2.4 Source and Sink Skeletons ....................... 28
  8.3 Helper Functions ....................................... 28

9 The module DiscreteEventLib ................................ 29
  9.1 Overview ................................................ 29
  9.2 Data Types ............................................. 29
  9.3 Skeletons .............................................. 30
  9.4 Skeletons for combinatorial processes ............. 30
  9.5 Skeletons for sequential processes ................. 31
  9.6 implementation ....................................... 33
    9.6.1 Combinatorial Skeletons ......................... 33
    9.6.2 Sequential Skeletons ............................ 34
    9.6.3 supporting function .............................. 36

10 The module StochasticLib .................................. 40

11 The Stochastic Library .................................... 41
  11.1 Overview ............................................. 41
  11.2 Skeletons for Stochastic Processes ............... 41
  11.3 Implementation of Skeletons ...................... 41
    11.3.1 Skeletons for Stochastic Processes .......... 41
  11.4 Supporting Functions ................................ 42

12 The module FiringRules .................................... 44
  12.1 Overview ............................................. 44

13 The module Vector ......................................... 45
  13.1 Overview ............................................. 45
  13.2 The Data Type Vector ................................ 45
  13.3 Functions on the Data Type vector ............... 45
  13.4 Implementation ....................................... 48
    13.4.1 The Data Type vector .......................... 48
    13.4.2 Functions on the Data Type Vector .......... 48

14 The module Memory ......................................... 51
  14.1 Overview ............................................. 51
  14.2 Data Structure ....................................... 51
  14.3 Functions on the data type Memory ............... 51
  14.4 Implementation ....................................... 52

15 The module Queue ......................................... 53
  15.1 Overview ............................................. 53
  15.2 The data type Queue ................................ 53
  15.3 Functions on the data types Queue and FiniteQueue .... 53
  15.4 Implementation ....................................... 54
16 Possible Extensions to the ForSyDe Standard Library

16.1 Overview ............................................ 55
Chapter 1

Introduction

1.1 Overview

The ForSyDe standard library ForSyDeStdLib provides data types and functions to model systems according to the ForSyDe methodology.

The structure of the library is shown in Figure 1.1.

![Figure 1.1: The Structure of the ForSyDeStdLib](image.png)

The document presents version 2.0, but the library is under development. However, we try to keep new versions of the library as compatible with older versions as possible.

Additional application libraries can be built on top of this library.
Chapter 2

The module **ForSyDeStdLib**

2.1 Overview

The ForSyDe Standard Library contains the data types and functions for the ForSyDe design methodology.

The module `ForSyDeStdLib` works as a container and exports the other libraries.

```haskell
module ForSyDeStdLib(  
  module StochasticLib,  
  module SynchronousProcessLib,  
  module SynchronousLib,  
  module DiscreteEventLib,  
  module UntimedLib,  
  module Vector,  
  module Signal,  
  module Memory,  
  module AbsentExt,  
  module Queue)  
where

import StochasticLib  
import UntimedLib  
import DataflowLib  
import SynchronousProcessLib  
import SynchronousLib  
import DiscreteEventLib  
import Vector  
import Signal  
import Memory  
import AbsentExt  
import Queue  
import FiringRules
```
Chapter 3

The module \textbf{AbsentExt}

3.1 Overview

The module \texttt{AbsentExt} is used to extend existing data types with the value "absent" ($\bot$).

\begin{verbatim}
module AbsentExt(
  AbsExt (Abst, Prst). fromAbstExt, abstExt,
  isAbsent, isPresent)

where

The data type \texttt{AbsExt} has two constructors. The constructor \texttt{Abst} is used to model the absence of a value, while the constructor \texttt{Prst} is used to model present values.

\begin{verbatim}
data AbsExt a = Abst
  | Prst a deriving (Eq)
\end{verbatim}

The data type \texttt{AbsExt} is defined as an instance of \texttt{Show} and \texttt{Read}. '@' represents the value \texttt{Abst} while a present value is represented with its value, e.g. \texttt{Prst 1} is represented as '\texttt{1}'.

3.2 Functions on the data type \textbf{AbsentExt}

The module defines the following functions:

\begin{verbatim}
abstExt :: a \rightarrow AbsExt a
fromAbstExt :: a \rightarrow AbsExt a \rightarrow a
isPresent :: AbsExt a \rightarrow Bool
isAbsent :: AbsExt a \rightarrow Bool
\end{verbatim}

The function \texttt{abstExt} converts a value into an extended value. The function \texttt{fromAbsExt} converts a value from a extended value.

The functions \texttt{isPresent} and \texttt{isAbsent} check for the presence or absence of a value.

3.3 Implementation of library functions

\begin{verbatim}
instance Show a \Rightarrow Show (AbsExt a) where
  showsPrec _ x = showsAbstExt x

showsAbstExt Abst = (+) "\bot"

showsAbstExt (Prst x) = (+) (show x)
\end{verbatim}
instance Read a ⇒ Read (AbstExt a) where
    readsPrec _ x = readsAbstExt x

readsAbstExt :: (Read a) ⇒ ReadS (AbstExt a)
readsAbstExt s = [(Abst, r1), ("\", r1) ← lex s]
    ++ [(Pnst x, r2), (x, r2) ← reads s]

abstExt v = Pnst v

fromAbstExt x Abst = x
fromAbstExt _ (Pnst y) = y

isPresent Abst = False
isPresent (Pnst _) = True

isAbsent = not . isPresent
Chapter 4

The module Signal

4.1 Overview

The module Signal defines the data type Signal and functions operating on this data type.

```haskell
module Signal (
    Signal (NullS, (:-)). (-:). (:+). (!-).
    signal, fromSignal,
    unitS, nullS, headS, tailS, atS, takeS, dropS,
    lengthS
)

where

    infixr 5 :-
    infixr 5 :-:
    infixr 5 :+:
    infixr 5 !:-
```

4.2 The data type Signal

A signal is defined as a list of events. An event has a tag and a value. The tag of an event is defined by the position in the list.

```haskell
data Signal a = NullS
    | a :- Signal a deriving (Eq)
```

A signal is defined as an instance of the classes Read and Show. The signal 1 :- 2 :- NullS is represented as \{1,2\}.

4.3 Functions on the data type Signal

The module defines the following on the data type Signal:

```haskell
signal :: [a] -> Signal a
fromSignal :: Signal a -> [a]
unitS :: a -> Signal a
nullS :: Signal a -> Bool
headS :: Signal a -> a
tailS :: Signal a -> Signal a
atS :: Int -> Signal a -> a
takeS :: Int -> Signal a -> Signal a
dropS :: Int -> Signal a -> Signal a
```
The functions signal and fromSignal convert a list into a signal and vice versa. The function unitS creates a signal with one value. The function nullS checks if a signal is empty. The function headS gives the first value - the head of a signal, while tailS gives the rest of the signal - the tail. The function atS gives returns the n-th event in a signal. The numbering of events in a signal starts with 0. There is also an operator version of this function, (!). The function takeS returns the first n values of a signal, while the function dropS drops the first n values from a signal. New signals can be created by means of the following functions. The data constructor (::) adds an element to the signal at the head of the signal. The function lengthS returns the length of a finite stream. The operator (-) adds at an element to a signal at the tail. Finally the operator (+++) concatenates two signals into one signal.

The combinator fanS takes two processes p1 and p2 and and generates a process network, where a signal is split and processed by the processes p1 and p2.

fanS :: (Signal a → Signal b) → (Signal a → Signal c) → Signal a → (Signal b. Signal c)

4.4 Implementation of the library functions

4.4.1 Overloading of Show and Read for the data type Signal

instance (Show a) ⇒ Show (Signal a) where
  showPrec p nullS = showParent (p > 9) (showString "[]")
  showPrec p xs = showParent (p > 9) (showChar ' '. showSignal1 xs)
    where
      showSignal1 nullS = showChar '.
      showSignal1 (x: nullS) = shows x . showChar ' '.
      showSignal1 (x: xs) = shows x . showChar '.'
      . showSignal1 xs

instance Read a ⇒ Read (Signal a) where
  readsPrec _ s = readsSignal s

readsSignal :: (Read a) ⇒ Reads (Signal a)
readsSignal s = [(x: nullS), rest] |
  ("\[", r2) ← lex s,
  (x, r3) ← reads r2,
  ("\]", rest) ← lex r3
++ [(nullS, r4) |
  ("\[", r5) ← lex s,
  ("\]", r4) ← lex r5]
++ [(x:=-xs), r6] |
  ("\[", r7) ← lex s,
  (x, r8) ← reads r7,
  ("\]", r9) ← lex r8,
  (xs, r6) ← readsValues r9]
readsValues :: (Read a) => Signal a
readsValues s = [((x: NullS), r1)
| (x, r2) <- reads s,
(".", r1) <- lex r2]
++ [((x:xs), r3)
| (x, r4) <- reads s,
(".", r5) <- lex r4,
(xs, r3) <- readsValues r5]

4.4.2 Functions on the data type Signal

signal [] = NullS
signal (x:xs) = x := signal xs

fromSignal NullS = []
fromSignal (x:xs) = x : fromSignal xs

unitS x = x := NullS

nullS NullS = True
nullS _ = False

headS NullS = error "headS Signal is empty"
headS (x:_) = x

tailS NullS = error "tailS Signal is empty"
tailS (_:xs) = xs

atS _ NullS = error "atS Signal has not enough elements"

atS 0 (x:_) = x
atS n (_:xs) = atS (n-1) xs

(!-) xss n = atS n xs

takeS 0 _ = NullS
takeS _ NullS = NullS
takeS n (x:xs) n <= 0 = NullS
| otherwise = x := takeS (n-1) xs

dropS 0 NullS = NullS
dropS _ NullS = NullS
dropS n (x:xs) n <= 0 = x:xs
| otherwise = dropS (n-1) xs

(->) xs x = xs ++ (x := NullS)

(->) NullS ys = ys
(->) (x:xs) ys = x := (xs ++ ys)

lengthS NullS = 0
lengthS (_:xs) = 1 + lengthS xs

fanS p1 p2 xs = (p1 xs, p2 xs)
Chapter 5

The module SynchronousLib

5.1 Overview

The synchronous library SynchronousLib defines skeletons and processes for the synchronous computational model. A skeleton is a higher order function which together with combinatorial function(s) and values as arguments constructs a process. Thus a skeleton can also be viewed as a process constructor.

```haskell
module SynchronousLib(
    module Vector, module Signal, module AbsentExt,
    module Memory, unzipSY, zipWithSY, zipWith3SY, zipWith4SY, scanlSY, scanl2SY, scanl3SY, scanlDelaySY, scanlDelay2SY, scanlDelay3SY, delaySY, delaynSY, whenSY, fillSY, hold5SY, zipSY, unzipSY, unzip3SY, zipxSY, unzipxSY, mapSY, mooreSY, moore2SY, moore3SY, mealySY, mealy2SY, mealy3SY, fstSY, sndSY, groupSY
) where

import Signal
import Vector
import AbsentExt
import Memory
```

5.2 Skeletons for Combinatorial Processes

Combinatorial processes do not possess an internal state, so that the output only depends on input signals. The module includes the following skeletons for combinatorial processes:

- `mapSY :: (a -> b) -> Signal a -> Signal b`
- `zipWithSY :: (a -> b -> c) -> Signal a -> Signal b -> Signal c`
- `zipWith3SY :: (a -> b -> c -> d) -> Signal a -> Signal b -> Signal c -> Signal d`
- `zipWith4SY :: (a -> b -> c -> d -> e) -> Signal a -> Signal b -> Signal c -> Signal d -> Signal e`
- `mapSY :: (a -> b) -> Vector (Signal a) -> Vector (Signal b)`

The skeleton `mapSY` takes a combinatorial function as argument and returns a process with one input signal and one output signal. This is shown in the following, where `mapSY (+1)` is a process which increments all values of an input signal.
In a similar way zipWithSY, zipWith3SY and zipWith4SY apply a combinatorial function on a number of input signals.

The skeleton mapxSY creates a process network that maps a function onto all signals in a vector of signals.

5.3 Skeletons for Sequential Processes

Sequential processes have a local state. Skeletons that construct such processes take not only functions but also values as arguments to express the value of the local state of the process. The output of sequential processes is deterministic and depends on the initial state and the input signals.

The module includes the following skeletons for sequential processes:

- **scanlSY** :: (a → b → a) → a → Signal b → Signal a
- **scanl2SY** :: (a → b → c → a) → a → Signal b → Signal c → Signal a
- **scan3SY** :: (a → b → c → d → a) → a → Signal b → Signal c → Signal d → Signal a
- **scanDelaySY** :: (a → b → a) → a → Signal b → Signal a
- **scanDelay2SY** :: (a → b → c → a) → a → Signal b → Signal c → Signal a
- **scanDelay3SY** :: (a → b → c → d → a) → a → Signal b → Signal c → Signal d → Signal a
- **mooreSY** :: (a → b → a) → (a → c) → a → Signal b → Signal c
- **moore2SY** :: (a → b → c → a) → (a → d) → a → Signal b → Signal c → Signal d
- **moore3SY** :: (a → b → c → d → a) → (a → e) → a → Signal b → Signal c → Signal d → Signal e
- **mealySY** :: (a → b → a) → (a → b → c) → a → Signal b → Signal c
- **mealy2SY** :: (a → b → c → a) → (a → b → c → d) → a → Signal b → Signal c → Signal d
- **mealy3SY** :: (a → b → c → d → a) → (a → b → c → d → e) → a → Signal b → Signal c → Signal d → Signal e
- **delaySY** :: a → Signal a → Signal a
- **delaynSY** :: a → Int → Signal a → Signal a
- **filterSY** :: (a → Bool) → Signal a → Signal (AbsExt a)

We define two different basic skeletons to construct sequential processes, scanlSY and scanDelaySY. Both skeletons take a function \( f \) and a state \( s \) as arguments. Both skeletons use the function \( f \) to calculate the next state, but calculate the output in a different way. scanlSY behaves like the Haskell prelude function scanl and has the value of the new state as its output value, while scanDelaySY has the current state value as output. The following example exemplifies this:

```haskell
SynchronousLib> scanlSY (+) 0 (signal [1,2,3,4])
{1,3,6,10} :: Signal Integer
SynchronousLib> scanDelaySY (+) 0 (signal [1,2,3,4])
{0,1,3,6} :: Signal Integer
```

Skeletons like scanl2SY, scanl2DelaySY are used in the same way for several input signals.

The skeletons mooreSY and mealySY are used to model state machines. These skeletons are based on the skeleton scanlDelaySY as is naturally for state machines in hardware, that the output operates on the current state and not on the next state.

These skeletons take a function \( f \) to calculate the next state, a function \( o \) to calculate the output and an value \( m \) for the initial state. In a process based on
the mooreSY skeleton the output function \( o \) operates only on the current state of the process. In contrast the output function of a process based on the mealySY skeleton operates on both the current state and the input.

The skeleton delaySY delays the signal one event cycle by introducing an initial value at the beginning of the output signal. The skeleton delaynSY delays the signal \( n \) events by introducing \( n \) identical default values.

The skeleton filterSY takes a predicate \( p \) and produces a process, that discards all values that do not fullfil the predicate \( p \). In this case the output is \( \bot \).

### 5.4 Processes

The module also contains the following synchronous processes:

- **whenSY** :: \( \text{Signal} \ (\text{AbstExt \ a}) \rightarrow \text{Signal} \ (\text{AbstExt \ b}) \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \)
- **fillSY** :: \( \text{AbstExt \ a} \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \)
- **holdSY** :: \( \text{AbstExt \ a} \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \)
- **zipSY** :: \( \text{Signal \ a} \rightarrow \text{Signal \ b} \rightarrow \text{Signal \ (a,b)} \)
- **zip3SY** :: \( \text{Signal \ a} \rightarrow \text{Signal \ b} \rightarrow \text{Signal \ c} \rightarrow \text{Signal \ (a,b,c)} \)
- **unzipSY** :: \( \text{Signal \ (a,b)} \rightarrow (\text{Signal \ a, Signal \ b}) \)
- **unzip3SY** :: \( \text{Signal \ (a,b,c)} \rightarrow (\text{Signal \ a, Signal \ b, Signal \ c}) \)
- **zipxSY** :: \( \text{Vector} \ (\text{Signal \ a}) \rightarrow \text{Signal} \ (\text{Vector \ a}) \)
- **unzipxSY** :: \( \text{Signal} \ (\text{Vector \ a}) \rightarrow \text{Vector} \ (\text{Signal \ a}) \)
- **groupSY** :: \( \text{Int} \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ (\text{AbstExt \ (Vector \ a)}) \)
- **fstSY** :: \( \text{Signal} \ (\text{AbstExt \ (a,b)}) \rightarrow \text{Signal} \ (\text{AbstExt \ a}) \)
- **sndSY** :: \( \text{Signal} \ (\text{AbstExt \ (a,b)}) \rightarrow \text{Signal} \ (\text{AbstExt \ b}) \)

The skeleton **whenSY** creates a process that synchronizes a signal of timed values with another signal of timed values. The output signal has the value of the first signal whenever an event has a present value and \( \bot \) when the event has an absent value.

The skeleton **fillSY** creates a process that 'fills' a signal with timed values by replacing absent values with a given present value.

The skeleton **holdSY** creates a process that 'fills' a signal with timed values by replacing absent values by the preceding present value. Only in cases, where no preceding value exists, the absent value is replaced by a supplied present value.

The process **zipSY** 'zips' two incoming signals into one signal of tuples, while the process **unzipSY** 'unzips' a signal of tuples into two signals. The functions **zip3SY** and **unzip3SY** perform the corresponding function for three signals.

The process **zipxSY** 'zips' a signal of vectors into a vector of signals. The process **unzipxSY** 'unzips' a vector of signals into a signal of vectors.

The function **groupSY** groups values into a vector of size \( n \), which takes \( n \) cycles. While the grouping takes place the output from this process consists of absent values.

The processes **fstSY** and **sndSY** select the always the first or second value from a signal of timed values of pairs.

### 5.5 Implementation of Library Functions

#### 5.5.1 Skeletons for Combinatorial Processes

- **mapSY** :: \( \text{NullS} = \text{NullS} \)
- **mapSY** :: \( f \ (x:xs) = f \ x \ : \ : \ (\text{mapSY} \ f \ xs) \)

- **zipWithSY** :: \( \text{NullS} \ _ = \text{NullS} \)
- **zipWithSY** :: \( \_ \ _ = \text{NullS} \)
zipWithSY f (x:-xs) (y:-ys) = f x y (zipWithSY f xs ys)
zipWith3SY _ NullIS _ _ = NullIS
zipWith3SY _ _ NullIS _ = NullIS
zipWith3SY _ _ _ _ = NullIS = NullIS
zipWith3SY f (x:-xs) (y:-ys) (z:-zs) = f x y z (zipWith3SY f xs ys zs)
zipWith4SY _ NullIS _ _ _ = NullIS
zipWith4SY _ _ NullIS _ = NullIS
zipWith4SY _ _ _ _ _ = NullIS = NullIS
zipWith4SY _ _ _ _ _ NullIS = NullIS
zipWith4SY f (w:-ws) (x:-xs) (y:-ys) (z:-zs) = f w x y z (zipWith4SY f ws xs ys zs)
mapSY f = mapV (mapSY f)

5.5.2 Skeletons for Sequential Processes

scanISY _ _ NullIS = NullIS
scanISY f mem (x:-xs) = f mem x (scanISY f newmem xs)
  where newmem = f mem x
scan2SY _ _ NullIS _ = NullIS
scan2SY _ _ _ _ NullIS = NullIS
scan2SY f mem (x:-xs) (y:-ys) = f mem x y (scan2SY f newmem xs ys)
  where newmem = f mem x y
scan3SY _ _ NullIS _ _ = NullIS
scan3SY _ _ _ _ NullIS = NullIS
scan3SY f mem (x:-xs) (y:-ys) (z:-zs) = f mem x y z (scan3SY f newmem xs ys zs)
  where newmem = f mem x y z
scanDelaySY _ _ NullIS = NullIS
scanDelaySY f mem (x:-xs) = mem (scanDelaySY f newmem xs)
  where newmem = f mem x
scanDelay2SY _ _ NullIS _ = NullIS
scanDelay2SY _ _ _ _ NullIS = NullIS
scanDelay2SY f mem (x:-xs) (y:-ys) = mem (scanDelay2SY f newmem xs ys)
  where newmem = f mem x y
scanDelay3SY _ _ NullIS _ _ = NullIS
scanDelay3SY _ _ _ _ NullIS = NullIS
scanDelay3SY f mem (x:-xs) (y:-ys) (z:-zs) = mem (scanDelay3SY f newmem xs ys zs)
  where newmem = f mem x y z
delaySY e es = e:-es
delaynSY e n xs | n <= 0 = xs
               | otherwise = e (delaynSY e (n-1) xs
mooeSY nextState output initial
     = mapSY output (scanDelaySY nextState initial)
moose2SY nextState output initial inp1 inp2 =
mapSY output (scanDelay2SY nextState initial inp1 inp2)

moose3SY nextState output initial inp1 inp2 inp3 =
mapSY output (scanDelay3SY nextState initial inp1 inp2 inp3)

mealySY nextState output initial signal =
zipWithSY output (scanDelaySY nextState initial signal) signal

mealy2SY nextState output initial inp1 inp2 =
zipWith3SY output (scanDelay2SY nextState initial inp1 inp2)
inp1 inp2

mealy3SY nextState output initial inp1 inp2 inp3 =
zipWith4SY output (scanDelay3SY nextState initial inp1 inp2 inp3)
inp1 inp2 inp3

filterSY p NullIS = NullIS
filterSY p (x:-xs) = if (p x == True) then
Prst x := filterSY p xs
else
Abst := filterSY p xs

5.5.3 Processes

whenSY NullIS _ = NullIS
whenSY _ NullIS = NullIS
whenSY (._:-xs) (Abst:-ys) = Abst :- (whenSY xs ys)
whenSY (x:-xs) (_:-ys) = x :- (whenSY xs ys)

fillSY a xs = mapSY (replaceAbst a) xs
where replaceAbst a Abst = a
   replaceAbst . (Prst x) = (Prst x)

holdSY a xs = scanSY hold a xs
where hold a Abst = a
   hold . (Prst x) = Prst x

zipSY (x:-xs) (y:-ys) = (x, y) :- zipSY xs ys
zipSY _ _ = NullIS

zip3SY (x:-xs) (y:-ys) (z:-zs) = (x, y, z) :- zip3SY xs ys zs
zip3SY _ _ _ = NullIS

unzip3SY NullIS = (NullIS, NullIS)
unzip3SY ((x, y):-xs) = (x:-xs, y:-ys) where (xs, ys) = unzip3SY xys
unzip3SY _ = (NullIS, NullIS, NullIS)
unzip3SY ((x, y, z):-xs) = (x:-xs, y:-ys, z:-zs) where (xs, ys, zs) = unzip3SY xys zs

zipxSY NullIV = NullIS
zipxSY (NullIS :> xss) = zipxSY xss
zipxSY ((x:-xs) :> xss) = (x :> (mapV headS xss))
   :- (zipxSY (xs := (mapV tailS xss)))
unzipxSY NullIS = NullIV
unzipxSY (NullV :- vss) = unzipxSY vss
unzipxSY ((v :- vs) :- vss) = (v :- (mapSY headV vs))
    :- (unzipxSY (vs :- (mapSY tailV vs)))

groupId n xs = mooreSY (addElement n) (output n) (NullV, 0) xs
    where addElement m (vs, n) x | n < m = (vs <; x, n+1)
       | n == m = (unitV x, 1)
       output m (vs, n) | m == n = Prst vs
         | m /= n = Abst

fstSY = mapSY fst1
    where fst1 Abst = Abst
               fst1 (Prst (a, _)) = Prst a

sndSY = mapSY snd1
    where snd1 Abst = Abst
               snd1 (Prst (_, b)) = Prst b
Chapter 6

The module
SynchronousProcessLib

6.1 Overview

The synchronous process library SynchronousProcessLib defines processes for the synchronous computational model. It is based on the synchronous library SynchronousLib.

```haskell
module SynchronousProcessLib (
    module SynchronousLib,
    module Signal,
    module AbsentExt,
    fifoDelaySY, finiteFifoDelaySY,
    memorySY, mergeSY, counterSY
  ) where

import SynchronousLib
import Signal
import AbsentExt
import Queue
```

6.2 Processes

The library defines the following processes:

- `fifoDelaySY` :: `Signal [a]` → `Signal (AbsentExt a)`
- `finiteFifoDelaySY` :: `Int` → `Signal [a]` → `Signal (AbsentExt a)`
- `memorySY` :: `Int` → `Signal (Access a)` → `Signal (AbsentExt a)`
- `mergeSY` :: `Signal (AbsentExt a)` → `Signal (AbsentExt a)`
  → `Signal (AbsentExt a)`
- `counterSY` :: `(Enum a, Ord a) => a → a → Signal a`

The process `fifoDelaySY` implements a synchronous model of a FIFO with infinite size, while the process `finiteFifoDelaySY` implements a FIFO with finite size. Both FIFOs take a list of values at each event cycle and output one value. There is a delay of one cycle. The process `memorySY` implements a synchronous memory. It uses access functions of the type `Read adr` and `Write adr value`. The process `mergeSY` merges two input signals into a single signal. The process has an internal buffer in order to prevent loss of data. The process is deterministic and outputs events according to their time tag. If there are two valid values at on both signals. The value of the first signal is output first. The process `counter` implements a counter,
that counts from min to max. The process counterS has no input and its output is an infinite signal.

6.3 Implementation of Processes

\[
\begin{align*}
\text{fifoDelaySY} \; xs & \quad = \text{moemSY} \; \text{fifoState} \; \text{fifoOutput} (\text{queue} \; []) \; xs \\
\text{fifoState} \quad &::\; \text{Queue} \; a \rightarrow [a] \rightarrow \text{Queue} \; a \\
\text{fifoState} \; (Q \; []) \; xs & \quad = (Q \; xs) \\
\text{fifoState} \; q \; xs & \quad = \text{fst} \; (\text{popQ} \; (\text{pushListQ} \; q \; xs)) \\
\text{fifoOutput} \quad &::\; \text{Queue} \; a \rightarrow \text{AbstExt} \; a \\
\text{fifoOutput} \; (Q \; []) & \quad = \text{Abst} \\
\text{fifoOutput} \; (Q \; (x:xs)) & \quad = \text{Prst} \; x \\
\text{finiteFifoDelaySY} \; n \; xs & \quad = \text{moemSY} \; \text{fifoStateFQ} \; \text{fifoOutputFQ} \; (\text{finiteQueue} \; n \; []) \; xs \\
\text{fifoStateFQ} \quad &::\; \text{FiniteQueue} \; a \rightarrow [a] \rightarrow \text{FiniteQueue} \; a \\
\text{fifoStateFQ} \; (FQ \; n \; []) \; xs & \quad = (FQ \; n \; xs) \\
\text{fifoStateFQ} \; q \; xs & \quad = \text{fst} \; (\text{popFQ} \; (\text{pushListFQ} \; q \; xs)) \\
\text{fifoOutputFQ} \quad &::\; \text{FiniteQueue} \; a \rightarrow \text{AbstExt} \; a \\
\text{fifoOutputFQ} \; (FQ \; n \; []) & \quad = \text{Abst} \\
\text{fifoOutputFQ} \; (FQ \; n \; (x:xs)) & \quad = \text{Prst} \; x \\
\text{memorySY} \; \text{size} \; xs & \quad = \text{moemSY} \; ns \; o \; (\text{newMem} \; \text{size}) \; xs \\
\text{where} \\
\quad \text{ns} \; \text{mem} \; (\text{Read} \; x) & \quad = \text{memState} \; \text{mem} \; (\text{Read} \; x) \\
\quad \text{ns} \; \text{mem} \; (\text{Write} \; x \; v) & \quad = \text{memState} \; \text{mem} \; (\text{Write} \; x \; v) \\
\quad \text{o} \; \text{mem} \; (\text{Read} \; x) & \quad = \text{memOutput} \; \text{mem} \; (\text{Read} \; x) \\
\quad \text{o} \; \text{mem} \; (\text{Write} \; x \; v) & \quad = \text{memOutput} \; \text{mem} \; (\text{Write} \; x \; v) \\
\text{mergeSY} \; xs \; ys & \quad = \text{moemSY} \; \text{mergeState} \; \text{mergeOutput} \; [] \; xs \; ys \\
\text{where} \\
\quad \text{mergeState} \; [] \; \text{Abst} \; \text{Abst} & \quad = [] \\
\quad \text{mergeState} \; [] \; \text{Abst} \; (\text{Prst} \; y) & \quad = [y] \\
\quad \text{mergeState} \; [] \; (\text{Prst} \; x) \; \text{Abst} & \quad = [x] \\
\quad \text{mergeState} \; [] \; (\text{Prst} \; x) \; (\text{Prst} \; y) & \quad = [x, y] \\
\quad \text{mergeState} \; (u:us) \; \text{Abst} \; \text{Abst} & \quad = us \\
\quad \text{mergeState} \; (u:us) \; \text{Abst} \; (\text{Prst} \; y) & \quad = us \; ++ \; [y] \\
\quad \text{mergeState} \; (u:us) \; (\text{Prst} \; x) \; \text{Abst} & \quad = us \; ++ \; [x] \\
\quad \text{mergeState} \; (u:us) \; (\text{Prst} \; x) \; (\text{Prst} \; y) & \quad = us \; ++ \; [x, y] \\
\quad \text{mergeOutput} \; [] & \quad = \text{Abst} \\
\quad \text{mergeOutput} \; (u:us) & \quad = \text{Prst} \; u \\
\text{counterSY} \; \text{min} \; \text{max} & \quad = \text{counterSY}' \; \text{min} \; \text{max} \\
\text{where} \; \text{counterSY}' \; x \; \text{min} \; \text{max} & \quad | \quad x \; /= \; \text{max} \; = \; x \quad ::= \; \text{counterSY}' \; (\text{succ} \; x) \; \text{min} \; \text{max} \\
\quad & \quad | \quad x \; == \; \text{max} \; = \; x \quad ::= \; \text{counterSY}' \; \text{min} \; \text{max} 
\end{align*}
\]
Chapter 7

The module **DataflowLib**

7.1 Overview

This library defines data types, skeletons and functions to model dataflow process networks, as they are described by Lee and Parks in [2]).

Each process is defined by a set of firing rules and corresponding actions. A process fires, if the incoming signals match a firing rule. Then the process consumes the matched tokens and executes the action corresponding to the firing rule.

We follow their definition of data flow process networks for the combinatorial processes:

```
module DataflowLib(
    module Signal, FiringToken(Wild, Value), mapDF,
    zipWithDF, zipWith3DF, mealyDF, mooseDF, scanDF
)

where

import FiringRules
import Signal
```

7.2 Data Types

The data type FiringToken (defined in the module FiringRules) defines the data type for tokens. The constructor `Wild` constructs a token wildcard, the constructor `Value` a constructs a token with value `a`.

A sequence (pattern) matches a signal, if the sequence is a prefix of the signal. Table 12.1 illustrates the use of tokens in firing rules:

<table>
<thead>
<tr>
<th>Token</th>
<th>ForSyDe Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>NullS</td>
<td>matches always</td>
</tr>
<tr>
<td>[*]</td>
<td>[Wild]</td>
<td>matches signal with at least one token</td>
</tr>
<tr>
<td>[v]</td>
<td>[Value v]</td>
<td>matches signal with <code>v</code> as its first value</td>
</tr>
<tr>
<td>[*]</td>
<td>[Wild, Wild]</td>
<td>matches signals with at least two tokens</td>
</tr>
</tbody>
</table>

Table 7.1: Matching of Tokens
7.3 Skeletons

7.3.1 Skeletons for Combinatorial Processes

Combinatorial processes do not have an internal state. This means, that the output signal only depends on the input signals. The library contains the following combinatorial skeletons:

\[
\begin{align*}
\text{mapDF} & \quad : \text{Eq a} \Rightarrow \{ [[\text{FiringToken a}]] \} \\
& \quad \quad \rightarrow (\text{Signal} a \rightarrow [[b]]) \rightarrow \text{Signal} a \rightarrow \text{Signal} b \\
\text{zipWithDF} & \quad : (\text{Eq a}, \text{Eq b}) \Rightarrow \\
& \quad \quad \{ [[\text{FiringToken b}]. [[\text{FiringToken a}]]] \} \\
& \quad \quad \rightarrow (\text{Signal} b \rightarrow \text{Signal} a \rightarrow [[c]]) \rightarrow \text{Signal} b \rightarrow \text{Signal} c \\
\text{zipWith3DF} & \quad : (\text{Eq a}, \text{Eq b}, \text{Eq c}) \Rightarrow \\
& \quad \quad \{ [[\text{FiringToken a}]. [[\text{FiringToken b}]. [[\text{FiringToken c}]]]] \} \\
& \quad \quad \rightarrow (\text{Signal} a \rightarrow \text{Signal} b \rightarrow \text{Signal} c \rightarrow [[d]]) \rightarrow \text{Signal} a \rightarrow \text{Signal} b \rightarrow \text{Signal} c \rightarrow \text{Signal} d
\end{align*}
\]

The skeleton \text{mapDF} takes a list of firing rules, a list of corresponding output functions to generate a data flow process with one input and one output signal. The skeleton \text{zipWithDF} and \text{zipWith3DF} work in the same way, but have two and three input signals. To illustrate the concept of data flow processes, we create a process that selects tokens from two inputs according to a control signal. The process has the following firing rules [2]:

\[
\begin{align*}
\text{R}_1 & = \{ [[*], \perp, [T]] \} \\
\text{R}_2 & = \{ \perp, [[*], [F]] \}
\end{align*}
\]

(7.1) (7.2)

The corresponding ForSyDe formulation of the firing rules is:

\[
\text{selectRules} = \{ [[\text{Wild}]. [[.]], \{ \text{Value True} \}].
\quad \{ \text{Value False} \} \} )
\]

For the output we formulate the following set of output functions:

\[
\text{selectOutput xs ys \_} = \{ \text{headS xs}. \quad
\quad \text{headS ys} \}
\]

The select process \text{selectDF} is then defined by:

\[
\begin{align*}
\text{selectDF} & \quad : \text{Eq a} \Rightarrow \text{Signal} a \rightarrow \text{Signal} a \\
& \quad \quad \rightarrow \text{Signal} \text{Bool} \rightarrow \text{Signal} a \\
\text{selectDF} & \quad = \text{zipWith3DF selectRules selectOutput}
\end{align*}
\]

Given the signals s1, s2 and s3,

\[
\begin{align*}
s_1 & = \text{signal} [1,2,3,4,5,6] \\
s_2 & = \text{signal} [7,8,9,10,11,12] \\
s_3 & = \text{signal} [\text{True, True, False, False, True, True}]
\end{align*}
\]

the executed process gives the following results:

DataflowLib> selectDF s1 s2 s3
\{1,2,7,8,3,4\} :: Signal Integer

7.3.2 Skeletons for Sequential Processes

The \text{mealyDF} skeleton implements the most general state machine in the ForSyDe methodology. It takes a set of firing rules, a set of corresponding next state functions and a set of output functions as argument. A firing rule is a tuple. The first value is a pattern for the state, the second value corresponds to an input pattern.
When a pattern matches, the process fires, the corresponding next state and output functions are executed, and the tokens matching the pattern are consumed.

As an example we can view a process calculating the running sum of the input tokens. It has only one firing rule, which is illustrated in Table 7.3.2. A dataflow process using these firing rules and the initial state 0 can be formulated in ForSyDe as

\[
\begin{align*}
rs \ times \ xs & = \text{mealyDF} \ firingRule \ \text{nextState} \ output \ \text{initState} \ xs \\
\text{where} \\
\text{nextState} \ state \ xs & = [[\text{Wild}, \ [\text{Wild}]]] \\
\text{output} \ state & = [[\text{state}]] \\
\text{initState} & = 0
\end{align*}
\]

Execution of the process gives:

\[
\text{DataflowLib} > rs \ (\text{signal}[1,2,3,4,5,6]) \ \\
\{0,1,3,6,10,15\} :: \text{Signal Integer}
\]

Another 'running sum' process rs2 takes two tokens, pushes them into a queue of five elements and calculates the sum as output.

\[
\begin{align*}
rs2 & = \text{mealyDF} \ fs \ ns \ o \ init \\
\text{where} \\
fs & = [[\text{Wild}, \ [\text{Wild}, \ \text{Wild}]]] \\
ns \ state \ xs & = \text{drop 2 state} \ \text{++ fromSignal} \ (\text{takeS} \ 2 \ xs) \\
o \ state \ - & = [[\text{sum state}]]
\end{align*}
\]

\[
\text{DataflowLib} > rs2 \ (\text{signal [1,2,3,4,5,6,7,8,9,10]}) \\
\{0,3,10,20,30\} :: \text{Signal Integer}
\]

Besides the skeleton mealyDF there are two more skeletons mooreDF, which models a Moore automaton and scanlDF, which models an automaton without an output decoder.

\[
\begin{align*}
\text{mealyDF} & :: (\text{Eq} \ a, \ \text{Eq} \ b) \Rightarrow ([\text{FiringToken} \ b, [\text{FiringToken} \ a]]) \\
& \rightarrow (b \rightarrow \text{Signal} \ a \rightarrow [b]) \rightarrow (b \rightarrow \text{Signal} \ a \rightarrow [c]) \\
& \rightarrow b \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ c \\
\text{mooreDF} & :: (\text{Eq} \ a, \ \text{Eq} \ b) \Rightarrow ([\text{FiringToken} \ b, [\text{FiringToken} \ a]]) \\
& \rightarrow (b \rightarrow \text{Signal} \ a \rightarrow [b]) \rightarrow (b \rightarrow [c]) \\
& \rightarrow b \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ c \\
\text{scanlDF} & :: (\text{Eq} \ a, \ \text{Eq} \ b) \Rightarrow ([\text{FiringToken} \ b, [\text{FiringToken} \ a]]) \\
& \rightarrow (b \rightarrow \text{Signal} \ a \rightarrow [b]) \\
& \rightarrow b \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ b
\end{align*}
\]

### 7.4 Implementation

#### 7.4.1 Combinatorial Skeletons

\[
\begin{align*}
\text{mapDF} \ _ - _ & = \text{NullS} \\
\text{mapDF} \ rs \ as \ xs & = \text{output} \leftarrow \text{mapDF} \ rs \ as \ xs' \\
\text{where} \\
xs' & = \text{if matchedRule} < 0 \text{ then}
\end{align*}
\]
NullS
else
    consumeDF rule xs
matchedRule = (matchDF rs xs)
rule = rs !! matchedRule
output = if matchedRule < 0 then
    NullS
else
    signal ((as xs) !! matchedRule)

zipWithDF = NullS NullS NullS = NullS
zipWithDF rs as xs ys = output ←→ zipWithDF rs as xs' ys'
    (xs', ys') = if matchedRule < 0 then
        (NullS, NullS)
    else
        consume2DF rule xs ys
matchedRule = (match2DF rs xs ys)
rule = rs !! matchedRule
output = if matchedRule < 0 then
    NullS
else
    signal ((as xs ys) !! matchedRule)

zipWith3DF = NullS NullS NullS NullS = NullS
zipWith3DF rs as xs ys zs = output ←→ zipWith3DF rs as xs' ys' zs'
    (xs', ys', zs') = if matchedRule < 0 then
        (NullS, NullS, NullS)
    else
        consume3DF rule xs ys zs
matchedRule = (match3DF rs xs ys zs)
rule = rs !! matchedRule
output = if matchedRule < 0 then
    NullS
else
    signal ((as xs ys zs) !! matchedRule)

7.4.2 Sequential Skeletons

mealyDF = NullS
mealyDF fs ns o state xs = output ←→ mealyDF fs ns o state' xs'
    xs' = if matchedRule < 0 then
        NullS
    else
        consumeDF rule xs
matchedRule = matchStDF fs state xs
rule = snd (fs !! matchedRule)
output = signal ((o state xs) !! matchedRule)
state' = if matchedRule < 0 then
    error "Neither rule matches the pattern!"
else
    (ns state xs) !! matchedRule

mooreDF = NullS
mooreDF fs ns o state xs = output ←→ mooreDF fs ns o state' xs'
    xs' = if matchedRule < 0 then
nulls
else
  consumeDF rule xs
matchedRule = matchStDF fs state xs
rule = snd (fs !! matchedRule)
output = signal (o state)
state' = if matchedRule < 0 then
  error "No rule matches the pattern!"
else
  (ns state xs) !! matchedRule

scanDF . . . nulls = nulls
scanDF fs ns state xs = (units state)
  (i+i scanDF fs ns state' xs')
where
xs' = if matchedRule < 0 then
  nulls
else
  consumeDF rule xs
matchedRule = matchStDF fs state xs
rule = snd (fs !! matchedRule)
state' = if matchedRule < 0 then
  error "No rule matches the pattern!"
else
  (ns state xs) !! matchedRule

7.4.3 Supporting Functions

The function prefixDF takes a pattern and a signal and returns True, if the pattern
is a prefix from the signal.

prefixDF :: Eq a => [FiringToken a] -> Signal a -> Bool
prefixDF [] = True
prefixDF _ nulls = False
prefixDF (Wild:ps) (x:xs) = prefixDF ps xs
prefixDF ((Value p):ps) (x:xs) = if p == x then
  prefixDF ps xs
else
  False

The function consumeDF takes a pattern and a signal and 'consumes' the pattern
from the signal. The functions consume2DF and consume3DF work in the same
way as consumeDF, but with two and three input signals.

consumeDF :: Eq a => [FiringToken a]
  -> Signal a -> Signal a
consumeDF _ nulls = nulls
consumeDF [] xs = xs
consumeDF (Wild:ts) (x:xs) = consumeDF ts xs
consumeDF (Value t:ts) (x:xs) = if t == x then
  consumeDF ts xs
else
  error "Tokens not correct"

consume2DF :: (Eq a, Eq b) =>
  ([FiringToken a], [FiringToken b])
  -> Signal a -> Signal b -> (Signal a, Signal b)
consume2DF (px, py) xs ys = (consumeDF px xs,
  consumeDF py ys)

consume3DF :: (Eq a, Eq b, Eq c) =>
consume3DF \((px, py, pz)\) \(xs\) \(ys\) \(zs\) = (consumeDF \(px\) \(xs\), consumeDF \(py\) \(ys\), consumeDF \(pz\) \(zs\))

The function `matchDF` checks, which firing rule, starting from 0, is matched by the input signal. If no firing rule matches, the output is ‘-1’. The functions `match2DF` and `match3DF` work in the same way for two and three inputs.

```
matchDF :: (Num a, Eq b) =>
            [[FiringToken b]] \rightarrow Signal b \rightarrow a
matchDF rs xs = matchDF' 0 rs xs
  where matchDF' = []
        matchDF' n (t:rs) xs = if prefixDF r xs then
                                 n
                                else
                                 matchDF' (n+1) rs xs
```

```
match2DF :: (Num a, Eq b, Eq c) =>
            [([FiringToken b], [FiringToken c])] \rightarrow Signal b \rightarrow Signal c \rightarrow a
match2DF rs xs ys = match2DF' 0 rs xs ys
  where match2DF' = []
        match2DF' n (t:rs) xs ys = if prefixDF r xs&&
                                   prefixDF ry ys
                                   then
                                   n
                                   else
                                   match2DF' (n+1) rs xs ys
```

```
match3DF :: (Num a, Eq b, Eq c, Eq d) =>
            [([FiringToken b], [FiringToken d], [FiringToken c])] \rightarrow Signal b \rightarrow Signal d \rightarrow Signal c \rightarrow a
match3DF rs xs ys zs = match3DF' 0 rs xs ys zs
  where match3DF' = []
        match3DF' n (t:rs) xs ys zs = if prefixDF r xs&&
                                     prefixDF ry ys&&
                                     prefixDF rz zs
                                     then
                                     n
                                     else
                                     match3DF' (n+1) rs xs ys zs
```

The function `matchStDF` works in the same way as `matchDF`, but it looks on patterns that include the state.

```
matchStDF :: (Num a, Eq b, Eq c) =>
            [[FiringToken c],[FiringToken b]] \rightarrow c \rightarrow Signal b \rightarrow a
matchStDF rs state xs = matchStDF' 0 rs state xs
  where matchStDF' = []
        matchStDF' n (t:rs) state xs = if prefixDF (snd r) xs&&
                                      matchState (fst r) state
                                      then
                                      n
                                      else
```

24
matchStDF' (n+1) rs state xs

matchState :: Eq a ⇒ FiringToken a → a → Bool
matchState Wild = True
matchState (Value v) x = x == v
Chapter 8

The module UntimedLib

8.1 Overview

The untimed library follows the definition of the process constructors in the paper [].

We only need the signal modele from the ForSyDe libraries.

    module UntimedLib(
      module Signal, mapU, scanU, zipU, zipUs,  
      zipWithU, mealyU, mooreU, sourceU, sinkU, initU,  
      unzipU
    )

    where

    import Signal

8.2 Skeletons

8.2.1 Skeletons for Combinatorial Processes

    mapU :: Int → ([a] → [b]) → Signal a → Signal b
    mapU c f NullS = NullS
    mapU c f xs | lenS (takeSp c xs) < c = NullS
                 | lenS (takeSp c xs) == c
                   = signal (f (takeL c xs)) ⨿ (mapU c f (dropSp c xs))

The first parameter of mapU is a constant integer defining the number of tokens consumed in every evaluation cycle. The second argument is a function on lists of the input type and returning a list of the output type. For instance,

    r2 = mapU 1 f
    where f :: [Int] → [Int]
          f [x] = [2*x]

defines a process r2 which consumes one token in each evaluation cycle and multiplies it by two.

8.2.2 Skeletons for Sequential Processes

scanU has an internal state which is visible at the output. The first argument is a function γ which, given the state returns the number of tokens consumed next. The second argument is the next state function and the third is the initial state.
scanU :: (b->Int) → (b->[a]→b) → b → Signal a → Signal b
scanU gamma g state NullS = NullS
scanU gamma g state xs
  | lenS (takeSp c xs) == c = newstate
  | _ = scanU gamma g newstate (dropSp c xs)
where c = gamma state
  newstate = g state (takeL c xs)

The next two skeletons create Moore and Mealy based state machines. In
addition to the next state function they also have an output encoding function.

mooreU :: (b->Int) → (b->[a]→b) → (b → [c]) → b
  → Signal a → Signal c
mooreU (b->Int) (b-[a]→b) (b → [c]) → b
mooreU gamma g f state xs
  | length as == c = signal (f state)
  | _ = mooreU gamma g newstate (dropSp c xs)
where c = gamma state
  as = takeL c xs
  newstate = g state as

mealyU :: (b->Int) → (b->[a]→b) → (b → [a] → [c]) → b
  → Signal a → Signal c
mealyU (b->Int) (b-[a]→b) (b → [a] → [c]) → b
mealyU gamma g f state xs
  | length as == c = signal (f state as)
  | _ = mealyU gamma g newstate (dropSp c xs)
where c = gamma state
  as = takeL c xs
  newstate = g state as

8.2.3 Zip and Unzip based Skeletons

zipU :: Signal (Int,Int) → Signal a → Signal b → Signal ([a],[b])
zipU NullS _ _ = NullS
zipU _ NullS _ = NullS
zipU _ _ NullS = NullS
zipu ((c1,c2):_:_:cs) xs ys
  | lenS (takeSp c1 xs) == c1 && lenS (takeSp c2 ys) == c2
  | = (takeL c1 xs,takeL c2 ys) zipu (cs (dropSp c1 xs) (dropSp c2 ys))
  | (lenS (takeSp c1 xs)) < c1 || lenS (takeSp c2 ys) < c2
  | = NullS

zipUs :: Int → Int → Signal a → Signal b → Signal ([a],[b])
zipUs _ NullS _ _ = NullS
zipUs _ _ NullS _ = NullS
zipUs c1 c2 _ _ =
  | lenS (takeSp c1 xs) == c1 && lenS (takeSp c2 ys) == c2
  | = (takeL c1 xs,takeL c2 ys)
  | _ = zipUs c1 c2 (dropSp c1 xs) (dropSp c2 ys)
  | otherwise = NullS

zipWithU :: Int → Int → ([a]->[b]->[c]) → Signal a → Signal b → Signal c
zipWithU _ _ f NullS _ _ = NullS
zipWithU _ _ _ NullS = NullS
zipWithU c1 c2 f _ _ =
\[
\begin{align*}
| \text{lenS} \ (\text{takeSp} \ c1 \ xs) & = c1 & \text{lenS} \ (\text{takeSp} \ c2 \ ys) & = c2 \\
& = \text{signal} \ (f \ (\text{takeL} \ c1 \ xs) \ (\text{takeL} \ c2 \ ys)) \\
& \quad \mapsto \text{zipWithU} \ c1 \ c2 \ f \ (\text{dropSp} \ c1 \ xs) \ (\text{dropSp} \ c2 \ ys) \\
& \text{otherwise} = \text{NullS}
\end{align*}
\]

\[\text{unzipU} :: \text{Signal} \ ([a],[b]) \rightarrow (\text{Signal} \ a, \text{Signal} \ b)\]
\[\text{unzipU} \ \text{NullS} = (\text{NullS}, \text{NullS})\]
\[\text{unzipU} \ ((\text{as}, \text{bs}) \rightarrow x) = (\text{signal} \ \text{as} \mapsto \text{as},\]
\[\quad \text{signal} \ \text{bs} \mapsto \text{bs})\]
\[\text{where} \ (\text{as}, \text{bs}) = \text{unzipU} \ x\]

### 8.2.4 Source and Sink Skeletons

\[\text{sourceU} :: (a \rightarrow a) \rightarrow a \rightarrow \text{Signal} \ a\]
\[\text{sourceU} \ g \ \text{state} = \text{newstate} \leftarrow \text{sourceU} \ g \ \text{newstate}\]
\[\text{where} \ \text{newstate} = g \ \text{state}\]

\[\text{sinkU} :: (a \rightarrow \text{Int}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow \text{Signal} \ b \rightarrow \text{Signal} \ b\]
\[\text{sinkU} \ _ \ _ \ _ \ \text{NullS} = \text{NullS}\]
\[\text{sinkU} \ \gamma \ \text{g} \ \text{state} \ \text{xs}\]
\[\quad | \ \text{length} \ \text{as} = c = \text{sinkU} \ \gamma \ \text{g} \ \text{newstate} \ (\text{dropSp} \ c \ \text{xs})\]
\[\quad | \ \text{otherwise} = \text{NullS}\]
\[\text{where} \ \text{as} = \text{takeL} \ c \ \text{xs}\]
\[\quad c = \gamma \ \text{state}\]
\[\quad \text{newstate} = \text{g} \ \text{state}\]

\[\text{initU} \ \text{is used to initialise a signal. Its first argument is prepended to its second argument, a signal.}\]
\[\text{initU} :: [a] \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ a\]
\[\text{initU} \ \text{init} \ s = (\text{signal} \ \text{init}) \mapsto \ s\]

### 8.3 Helper Functions

\[\text{lenS} :: \text{Signal} \ a \rightarrow \text{Int}\]
\[\text{lenS} \ \text{NullS} = 0\]
\[\text{lenS} \ (x:\rightarrow \text{xs}) = 1 + \text{lenS} \ \text{xs}\]

\[\text{takeSp} \ 0 \ \text{null} = \text{null}\]
\[\text{takeSp} \ _ \ \text{null} = \text{null}\]
\[\text{takeSp} \ \text{n} \ (x:\rightarrow \text{xs}) = x : \text{takeSp} \ (\text{n}-1) \ \text{xs}\]

\[\text{dropSp} \ 0 \ \text{xs} = \text{xs}\]
\[\text{dropSp} \ _ \ \text{null} = \text{null}\]
\[\text{dropSp} \ \text{n} \ (x:\rightarrow \text{xs}) = \text{dropSp} \ (\text{n}-1) \ \text{xs}\]

\[\text{takeL} \ c = \text{fromSignal} \ . \ (\text{takeSp} \ c)\]
Chapter 9

The module **DiscreteEventLib**

9.1 Overview

This library defines data types, skeletons and functions to model dataflow process networks, as they are described by Lee and Parks in [2]).

Each process is defined by a set of firing rules and corresponding actions. A process fires, if the incoming signals match a firing rule. Then the process consumes the matched tokens and executes the action corresponding to the firing rule. However it outputs only those events which have time\_tag smaller than or equal to the maximum consumed input time\_tag. The rest is stored in memory and remaining there until some input time tag is consumed which is greater than that.

```haskell
module DiscreteEventLib (
    module Signal, DiscreteEvent(DE),
    mapDE, zipWithDE, zipWith3DE, mealyDE, mooreDE,
    scanDE)

where
import Signal
import FiringRules
```

9.2 Data Types

The data type **DiscreteEvent** defines the data type for events of a **DiscreteEvent** Signal.

Two functions are defined on this type as well, give\_val returns the value and give\_tag returns the timing tag of **DiscreteEvent**

```haskell
type Time\_tag = Int

data DiscreteEvent a = DE a Time\_tag deriving (Eq, Show)
give\_val :: DiscreteEvent a \to a
give\_val (DE x _) = x

give\_tag :: DiscreteEvent a \to Time\_tag
give\_tag (DE _ i) = i
```
9.3 Skeltons

9.4 Skeltons for combinatorial processes

The library contains the following combinatorial skeltons.

\[
\text{mapDE} :: \text{Eq } a \Rightarrow [[\text{FiringToken } a]] \rightarrow (\text{Signal } (\text{DiscreteEvent } a) \rightarrow \\
[\text{Signal } (\text{DiscreteEvent } b)]) \rightarrow \text{Signal } (\text{DiscreteEvent } a) \\
\rightarrow \text{Signal } (\text{DiscreteEvent } b)
\]

\[
\text{zipWithDE} :: (\text{Eq } a, \text{ Eq } b) \Rightarrow [[[\text{FiringToken } a],[\text{FiringToken } b]]] \\
\rightarrow (\text{Signal } (\text{DiscreteEvent } a) \rightarrow \text{Signal } (\text{DiscreteEvent } b) \\
\rightarrow [[\text{DiscreteEvent } c]]) \rightarrow \text{Signal } (\text{DiscreteEvent } a) \\
\rightarrow \text{Signal } (\text{DiscreteEvent } b) \rightarrow \text{Signal } (\text{DiscreteEvent } c)
\]

The skeleton \text{mapDE} takes a list of firing rules, a list of corresponding output functions to generate a DiscreteEvent process with one input and one output signal. The firing rules matches only the value of the event and not the time tag. The skeleton \text{zipWithDE} and \text{zipWith3DE} work in the same way, but have two and three input signals. To illustrate the concept of data flow processes, we create a process that selects tokens from two inputs according to a control signal. The process has the following firing rules

\[
\mathbf{R} = \{[[a, a', a', b'], [a', b', b'], [a, a', a']])
\]

(9.1)

The corresponding \text{ForSyDe} formulation of the firing rules is:

\[
\text{selectRules} = [[[\text{Wild, Value 'a'}, \text{Wild, Value 'b'}], \\
[\text{Wild, Value 'b'}, \text{Value 'b'}], \\
[\text{Wild, Value 'a'}]]
\]

For the output we formulate the following set of output functions: The function \text{maxTag i s}, returns the maximum timing tag of first \text{i} event of signal \text{s}.

\[
\text{selectOutput } xs = [f1 \; xs, \; f2 \; xs, \; f3 \; xs]
\]

\[
f1 \; x = \text{DE 'c' (t+1):-DE 'c' (t+2):-DE 'c' (t+3):--NullIS} \\
\text{where} \\
t = \text{maxTag 4} \; x
\]

\[
f3 \; x = \text{DE 'c' (2*\; t):--DE 'c' (4*\; t):--DE 'c' (4*\; t+2):--NullIS} \\
\text{where} \\
t = \text{maxTag 3} \; x
\]

\[
f2 \; x = \text{DE 'c' (t+1):-DE 'c' (t+2):-DE 'c' (t+4):--NullIS} \\
\text{where} \\
t = \text{maxTag 3} \; x
\]

\[
\text{maxTag } i \; \text{NullIS} = -1 \\
\text{maxTag 1 (DE } \text{ p j:--xs } = j \\
\text{maxTag } i \; (\text{DE } \text{ p j:--xs }) = \text{ max } j \; (\text{maxTag } (i-1) \; \text{xs})
\]

Now the process proc is defined as follows.
proc :: Signal (DiscreteEvent Char) → Signal (DiscreteEvent Char)
proc = mapDE selectRules selectOutput

Given the input signal s

\[
\begin{align*}
\text{DE } 'a' & : 1;\text{ DE } 'b' : 2;\text{ DE } 'a' : 3;\text{ DE } 'a' : 4; \\
\text{DE } 'a' : 5;\text{ DE } 'b' : 6;\text{ DE } 'b' : 7;\text{DE } 'a' : 8; \\
\text{DE } 'b' : 9;\text{DE } 'b' : .NullIS \\
\end{align*}
\]

The executed process gives the following result:

DiscreteEvent> proc s
\{DE 'c' 6,DE 'c' 8,DE 'c' 9,DE 'c' 10,DE 'c' 11,DE 'c' 12,DE 'c' 13,DE 'c' 14\} :: Signal DiscreteEvent Char

9.5 Skeltons for sequential processes

The mealyDF skeleton implements the most general state machine in the ForSyDe methodology. It takes a set of firing rules, a set of corresponding next state functions and a set of output functions as argument. A firing rule is a tuple. The first value is a pattern for the state, the second value corresponds to an input pattern. When a pattern matches, the process fires, the corresponding next state and output functions are executed, and the tokens matching the pattern are consumed. However the actual output in that firing cycle is of only those output events for which the timing tag is smaller than the maximal consumed input token so far. This has to be done, so that the final output signal have tags in globally increasing order.

The library contains the following sequential skeltons.

mealyDE :: (Eq a, Eq b) ⇒ [[FiringToken b.[FiringToken a]]]
→ (b → Signal (DiscreteEvent a) → [b])
→ (b → Signal (DiscreteEvent a) → [[DiscreteEvent c]])
→ b → Signal (DiscreteEvent a) → Signal (DiscreteEvent c)

mooreDE :: (Eq a, Eq b) ⇒ [[FiringToken b.[FiringToken a]]]
→ (b → Signal (DiscreteEvent a) → [b])
→ (b → [DiscreteEvent c]) → b → Signal (DiscreteEvent a)
→ Signal (DiscreteEvent c)

scanIDE :: (Eq a, Eq b)
⇒ [[(FiringToken (DiscreteEvent b).[FiringToken a])]]
→ (DiscreteEvent b → Signal (DiscreteEvent a)
→ [DiscreteEvent b]) → DiscreteEvent b
→ Signal (DiscreteEvent a) → Signal (DiscreteEvent b)

As an example consider a Mealy machine defined by following -

outputFun state xs = [ff1 state xs, ff2 state xs, ff3 state xs, ff4 state xs]
ff1 state NullIS = []
ff1 1 (DE x i:-xs) = (DE x (i\*t):ff1 1 xs)
where
  t = max_tag 3 (DE x i:-xs)
ff1 0 _ = []

ff2 0 x = [DE 'c' (t+2), DE 'c' (t+3), DE 'c' (t+11)]
where
  t = max_tag 3 x
ff2 1 NullIS = []
ff3 \_ x = [DE \ 'c' \ (4+t)]
where
t = max Jag 1 x

ff4 \_ x = [DE \ 'c' \ (t+7)]
where
t = max Jag 1 x

def nextstate fun state xs = [o1 state xs, o2 state xs, o1 state xs, o2 state xs] o1 state NullS = 0 o1 1 (DE \ 'b' \ _ := xs) = 0 o1 _ _ = 1 o2 0 (DE \ 'a' \ _ := xs) = 1 o2 _ _ = 0
defiring rules = [(Value 1, [Wild, Wild, Value \ 'a'])],
(Value 0, [Wild, Wild, Value \ 'a'])],
(Value 1, [Wild]), (Value 0, [Wild])]

inputt =\{DE \ 'b' 1:-DE \ 'b' 2:-DE \ 'a' 3:-DE \ 'a' 4:-DE \ 'b' 5:-DE \ 'a' 8:-DE \ 'b' 10
:-DE \ 'a' 12:-NullS\}

Now we can define a process \texttt{pro}, by using the \texttt{mealyDE} skeleton

\texttt{pro = mealyDE firing rules nextstate fun output fun 1}

Execution of the process gives:

\texttt{DiscreteEvent> pro inputt}
\{DE \ 'b' 3,DE \ 'b' 8,DE \ 'c' 10,DE \ 'c' 11,DE \ 'c' 14,DE \ 'a' 15,
DE \ 'c' 19,DE \ 'c' 19,DE \ 'a' 32,DE \ 'b' 50,DE \ 'a' 96,DE \ 'b' 120,
DE \ 'a' 144\};:: Signal (DiscreteEvent Char)

The figure explains how this is happening...
9.6 implementation

9.6.1 Combinatorial Skeltons

mapDE rs as xs = mapDD rs as xs NullS
zipWithDE rs as xs ys = zipWithDD rs as xs ys NullS
zipWith3DE rs as xs ys zs = zipWith3DD rs as xs ys zs NullS

The function mapDE is implemented by calling another function mapDD. The functionality of mapDD can be explained in following steps -

1. The local variable matchedRule carries the index of the rule which matches the current input signal.

2. using the output function as, and the index matchedRule, we get the value of output in the current cycle.

3. The variable s is used to carry information about the output produced in old cycles which is not yet outputted.

4. Combining the signal outputted in the present cycle and that which is coming from the old cycles, those events are selected which have tag less than the largest consumed input tags and are actually outputted.

5. The largest consumed input tag is given by variable i

6. Those events which are not outputted are send to the next cycle by the variable haskellremaining.

All the functions in DiscreteEvent Library essentially follows the theme described above.

mapDD _ _ NullS s = s
mapDD rs as xs s = put (+) mapDD rs as xs' remaining
where
  xs' = if matchedRule < 0 then
        NullS
    else
        fst l
  i = snd l
  l = consumeD rule xs
  matchedRule = (matchD rs xs)
  rule = rs !! matchedRule
  put = if matchedRule < 0 then
        NullS
    else
        fst m
  remaining = if matchedRule < 0 then
               s
    else
        snd m
  m = cutting s ((as xs) !! matchedRule) i
zipWithDD _ _ NullS NullS s = s
zipWithDD rs as xs ys s = put ←+ zipWithDD rs as xs’ ys’ remaining
where
    matchedRule = match2D rs xs ys
    rule = rs !! matchedRule
    (1,m,n) = consume2D rule xs ys
    xs’ = if matchedRule < 0 then
        NullS
    else
        m
    ys’ = if matchedRule < 0 then
        NullS
    else
        n
    put = if matchedRule < 0 then
        NullS
    else
        fst k
    remaining = if matchedRule < 0 then
        s
    else
        snd k
    k = cutting s (signal ((as xs ys) !! matchedRule)) n

zipWith3DD _ _ NullS NullS NullS s = s
zipWith3DD rs as xs ys zs s = put ←+ zipWith3DD rs as xs’ ys’ zs’ remaining
where
    matchedRule = match3D rs xs ys zs
    rule = rs !! matchedRule
    (1,m,o,n) = consume3D rule xs ys zs
    xs’ = if matchedRule < 0 then
        NullS
    else
        m
    ys’ = if matchedRule < 0 then
        NullS
    else
        n
    zs’ = if matchedRule < 0 then
        NullS
    else
        o
    put = if matchedRule < 0 then
        NullS
    else
        fst k
    remaining = if matchedRule < 0 then
        s
    else
        snd k
    k = cutting s (signal ((as xs ys zs) !! matchedRule)) n

9.6.2 Sequential Skeltons

mealyDE fs ns o state xs = mealyDD fs ns o state xs NullS
mealyDD fs ns o state NullS s = s
mealyDD fs ns o state xs s = put mealyDD fs ns o state’ xs’ remaining
where
matchedRule = matchStD fs state xs
rule = snd (fs !! matchedRule)
state’ =
  if matchedRule < 0 then
    error "No rule matches the pattern!!"
  else
    (ns state xs) !! matchedRule
l = consumeD rule xs
xs’ = if matchedRule < 0 then
      NullS
    else
      fst l
i = snd l
m = cutting s (signal ((o state xs) !! matchedRule)) i
put = if matchedRule < 0 then
      NullS
    else
      fst m
remaining = if matchedRule < 0 then
      NullS
    else
      snd m

mooreDD fs ns o state xs = mooreDD fs ns o state xs NullS
mooreDD fs ns o state NullS s = s
mooreDD fs ns o state xs s = put mealyDD fs ns o state’ xs’ remaining
where
matchedRule = matchStD fs state xs
rule = snd (fs !! matchedRule)
state’ =
  if matchedRule < 0 then
    error "No rule matches the pattern!!"
  else
    (ns state xs) !! matchedRule
l = consumeD rule xs
xs’ = if matchedRule < 0 then
      NullS
    else
      fst l
i = snd l
m = cutting s (signal (o state)) i
put = if matchedRule < 0 then
      NullS
    else
      fst m
remaining = if matchedRule < 0 then
      NullS
    else
      snd m

scanIDE fs ns state xs = scanIDD fs ns state xs NullS
scanIDD fs ns state NullS s = s
scanIDD fs ns state xs s = put scanIDD fs ns state’ xs’ remaining
where
matchedRule = matchStD fs state xs
rule = snd (fs !! matchedRule)
state' =
    if matchedRule < 0 then
        error "No rule matches the pattern!!"
    else
        (ns state xs) !! matchedRule
l = consumeD rule xs
xs' = if matchedRule < 0 then
    NullS
    else
        fst l
i = snd l
m = cutting s (unitS state) i
put = if matchedRule < 0 then
    NullS
    else
        fst m
remaining = if matchedRule < 0 then
    NullS
    else
        snd m

9.6.3 supporting function

The function prefixD takes a pattern and a signal of discrete events and return true if the pattern matches the signal. The pattern have only values and not the time tag.

prefixD :: Eq a ⇒ [FiringToken a] → Signal (DiscreteEvent a) → Bool

prefixD [] = True
prefixD _ NullS = False
prefixD (Wild:ps) (x:xs) = prefixD ps xs
prefixD ((Value p):ps) (x:xs) = if p == (give_val x) then
    prefixD ps xs
    else
        False

The function cutting is taking two signals and an integer i as argument and it is returning two sorted signals put and remaining. The signal put is having those values which are less than i and the signal remaining has those values which are greater than i.

cutting :: Signal (DiscreteEvent a) → Signal (DiscreteEvent a) →
        Int → (Signal (DiscreteEvent a),Signal (DiscreteEvent a))

cutting s t i = (put,remaining)
where
    put = fst l
    l = (lessThan s i)
    remaining = sortt (snd l) t

sortt :: Signal (DiscreteEvent a) → Signal (DiscreteEvent a) → Signal (DiscreteEvent a)

sortt NullS t = t
sortt s NullS = s
sortt (x:xs) (y:ys)

36
\( (\text{give}_{\text{tag}} \ x) \leq (\text{give}_{\text{tag}} \ y) = (x:- (\text{sort} \ xs \ (y:-ys))) \)
\( \text{otherwise} = (y:- (\text{sort} \ (x:-xs) \ ys)) \)

\[
\text{lessThan} :: \text{Signal} \ (\text{DiscreteEvent} \ a) \rightarrow \text{Int} \\
\quad \rightarrow (\text{Signal} \ (\text{DiscreteEvent} \ a), \text{Signal} \ (\text{DiscreteEvent} \ a))
\]
\[
\text{lessThan} \text{ NullS} \ i = \text{NullS, NullS} \\
\text{lessThan} \ (x:-xs) \ i \\
\quad | (\text{give}_{\text{tag}} \ x) \leq i = (x:-p,q) \\
\quad | \text{otherwise} = (\text{NullS,} \ (x:-xs))
\]

where
\( (p,q) = \text{lessThan} \ xs \ i \)

The function \( \text{consumeT} \) takes a pattern and a signal and 'consumes' the pattern
from the signal. The functions \( \text{consume2T} \) and \( \text{consume3T} \) work in the same way
as \( \text{consumeT} \), but with two and three input signals.

\[
\text{consumeD} :: \text{Eq} \ a \Rightarrow ([\text{FiringToken} \ a] \rightarrow \text{Signal} \ (\text{DiscreteEvent} \ a) \\
\quad \rightarrow (\text{Signal} \ (\text{DiscreteEvent} \ a), \text{Int}))
\]
\[
\text{consumeD} \ \text{rule} \ \text{xs} = \text{consumeD'} \ 0 \ \text{rule} \ \text{xs}
\]
\[
\text{consumeD'} \ i = \text{NullS} = (\text{NullS, i})
\]
\[
\text{consumeD'} \ i \ [\] \ \text{xs} = (\text{xs, i})
\]
\[
\text{consumeD'} \ i \ (\text{[Wild:ts]} \ (a:-xs)) = \text{consumeD'} \ (\text{give}_{\text{tag}} \ a) \ ts \ \text{xs}
\]
\[
\text{consumeD'} \ i \ (\text{Value t:ts}) \ (a:-xs) = \text{if} \ t = (\text{give}_{\text{val}} \ a) \ \text{then}
\quad \text{consumeD'} \ (\text{give}_{\text{tag}} \ a) \ ts \ \text{xs}
\quad \text{else}
\quad \text{error} "\text{Token not correct!!}"\]

\[
\text{consume2D} :: (\text{Eq} \ a, \text{Eq} \ b) \Rightarrow ([\text{FiringToken} \ b], [\text{FiringToken} \ a]) \rightarrow \\
\quad \text{Signal} \ (\text{DiscreteEvent} \ b) \rightarrow \text{Signal} \ (\text{DiscreteEvent} \ a) \rightarrow \\
\quad (\text{Signal} \ (\text{DiscreteEvent} \ b), \text{Signal} \ (\text{DiscreteEvent} \ a), \text{Int})
\]
\[
\text{consume2D} \ \text{rule} \ \text{xs} \ \text{ys} = \text{consume2D'} \ 0 \ \text{rule} \ \text{xs} \ \text{ys}
\]
\[
\text{consume2D'} \ i = \text{NullS NullS} = (\text{NullS, NullS, i})
\]
\[
\text{consume2D'} \ i \ ([],[]) \ \text{xs} \ \text{ys} = (\text{xs, ys, i})
\]
\[
\text{consume2D'} \ i \ ([],(\text{Value a:ts})) \ (b:-ys) = \text{if} \ \text{give}_{\text{tag}} \ b < i \ \text{then}
\quad \text{consume2D'} \ i \ ([],ts) \ \text{xs} \ \text{ys}
\quad \text{else}
\quad \text{consume2D'} \ (\text{give}_{\text{tag}} \ b) \ ([],ts) \ \text{xs} \ \text{ys}
\quad \text{else}
\quad \text{error} "\text{Token not correct!!}"\]
\[
\text{consume2D'} \ i \ ((\text{Value a:ts}),[]) \ (b:-xs) \ \text{ys} = \text{if} \ a = (\text{give}_{\text{val}} \ b) \ \text{then}
\quad \text{if} \ \text{give}_{\text{tag}} \ b < i \ \text{then}
\quad \quad \text{consume2D'} \ i \ (ts,[]) \ \text{xs} \ \text{ys}
\quad \quad \quad \text{else}
\quad \quad \quad \text{consume2D'} \ (\text{give}_{\text{tag}} \ b) \ (ts,[]) \ \text{xs} \ \text{ys}
\quad \text{else}
\quad \text{error} "\text{Token not correct!!}"\]
\[
\text{consume2D'} \ i \ ((\text{Value a:ts}),\text{Value b:us}) \ (p:-xs) \ (q:-ys) = \text{if} \ a = (\text{give}_{\text{val}} \ p) \&\& b = (\text{give}_{\text{val}} \ q) \ \text{then}
\quad \text{if} \ \text{give}_{\text{tag}} \ p < (\text{give}_{\text{tag}} \ q) \ \text{then}
\quad \quad \text{consume2D'} \ (\text{give}_{\text{tag}} \ q) \ (ts,us) \ \text{xs} \ \text{ys}
\quad \quad \quad \text{else}
\quad \quad \quad \text{error} "\text{Token not correct!!}"\]

37
consume2D' (give tag p) (ts,us) xs ys
    else
      error "Token not correct!!!"

consume2D' i ((Wild:ts),(Value b:us)) (p:-xs) (q:-ys) =
  if (b == give_val q) then
    if (give_tag p) < (give_tag q) then
      consume2D' (give_tag q) (ts,us) xs ys
    else
      consume2D' (give_tag p) (ts,us) xs ys
  else
    error "Token not correct!!!"

consume2D' i ((Value a:ts),(Wild:us)) (p:-xs) (q:-ys) =
  if (a == give_val p) then
    if (give_tag p) < (give_tag q) then
      consume2D' (give_tag q) (ts,us) xs ys
    else
      consume2D' (give_tag p) (ts,us) xs ys
  else
    error "Token not correct!!!"

consume3D :: (Eq a, Eq b, Eq c) \Rightarrow ((FiringToken c)\cdot[FiringToken b]\cdot[FiringToken a])
  \rightarrow Signal (DiscreteEvent c) \rightarrow Signal (DiscreteEvent b)
  \rightarrow Signal (DiscreteEvent a)
  \rightarrow (Signal (DiscreteEvent c),Signal (DiscreteEvent b),Signal (DiscreteEvent a),Int)

consume3D (px,py,pz) xs ys zs = (xs',ys',zs',k)
where
  (xs'.p) = consumeD px xs
  (ys'.q) = consumeD py ys
  (zs'.r) = consumeD pz zs
  k = max (max p q) r

matchD :: (Num a, Eq b) \Rightarrow [[FiringToken b]] \rightarrow Signal (DiscreteEvent b) \rightarrow a
matchD rs xs = matchD' 0 rs xs
where
  matchD' = [] = \_ = -1
  matchD' n (r:rs) xs = if prefixD r xs then
    n
  else
    matchD' (n+1) rs xs

match2D :: (Num a, Eq b, Eq c) \Rightarrow [[[FiringToken b],[FiringToken c]]]
  \rightarrow Signal (DiscreteEvent b) \rightarrow Signal (DiscreteEvent c) \rightarrow a

match2D rs xs ys = match2D' 0 rs xs ys
where
  match2D' = [] = \_ = -1
  match2D' n ((r,ry):rs) xs ys
    = if prefixD r x && prefixD ry y then
      n
    else
      match2D' (n+1) rs xs ys

match3D :: (Num a, Eq b, Eq c, Eq d)
  \Rightarrow [[[FiringToken b],[FiringToken d],[FiringToken c]]]
  \rightarrow Signal (DiscreteEvent b) \rightarrow Signal (DiscreteEvent d)
  \rightarrow Signal (DiscreteEvent c) \rightarrow a

38
match3D rs xs ys zs = match3D' 0 rs xs ys zs
where
    match3D' _ _ _ _ = –1
    match3D' n ((rx, ry, rz):rs) xs ys zs =
        if prefixD rx xs&&
            prefixD ry ys&&
            prefixD rz zs
            then
            n
            else
            match3D' (n+1) rs xs ys zs

matchSttD :: (Num a, Eq b, Eq c) => [(FiringToken c,[FiringToken b])]
           -> c -> Signal (DiscreteEvent b) -> a

matchSttD rs state xs = matchSttD' 0 rs state xs
where
    matchSttD' _ _ _ _ = –1
    matchSttD' n (r:rs) state xs
        = if prefixD (snd r) xs&&
            matchSttD (fst r) state
            then
            n
            else
            matchSttD' (n+1) rs state xs

matchSttD :: Eq a => FiringToken a -> a -> Bool
matchSttD Wild _ = True
matchSttD (Value a) x = a==x
Chapter 10

The module StochasticLib
Chapter 11

The Stochastic Library

11.1 Overview

The stochastic library provides a few stochastic skeletons, which are relatives to the skeletons of the synchronous library. These skeletons are based on two elementary functions, `sigmaUn` and `sigmaGe` which provide stochastic signals. The background and motivation for this approach is described in the paper [1]. Unfortunately, not all of the suggested skeletons are implemented.

```haskell
module StochasticLib(module Signal, selMapSY, selScanSY, sigmaUn, sigmaGe)

import SynchronousLib
import Signal
import Random
```

11.2 Skeletons for Stochastic Processes

The module contains the following skeletons to construct stochastic processes:

```haskell
selMapSY :: Int → (a → b) → (a → b) → Signal a → Signal b
selScanSY :: Int → (a → b → a) → (a → b → a) → a → Signal b → Signal a
```

The skeleton `selMapSY` is a stochastic variant of `mapSY`. It has an internal stochastic process and selects one out of two combinatorial functions depending on the output of the stochastic process. The skeleton `selScanSY` is a stochastic variant of `scanSY`. `sigmaUn` generates a signal list of uniformly distributed Int within the given range and with a given seed.

11.3 Implementation of Skeletons

11.3.1 Skeletons for Stochastic Processes

```haskell
select1 :: Int → (a → b) → (a→b) → a → b
select1 0 f0 _ x = f0 x
select1 1 _ f1 x = f1 x

select2 :: Int → (a → b → c) → (a→b→c) → a → b → c
select2 0 f0 _ x y = f0 x y
```
11.4 Supporting Functions

sigmaGe is a more general stochastic process. The first argument is a function which describes the distribution. For each value \( v \) in the given range \( (r1, r2) \), \( f(v) \) is the probability that \( v \) is generated.

Note, that the user has to make sure that \( \text{sum}(f(v))=1 \) for \( v \) in \( (r1, r2) \).

\[
\text{sigmaGe} :: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Signal} \text{Int}
\]

\[
\text{sigmaGe} f \text{ seed } (r1.r2) = \text{sigma2} (\text{checkSum} f \text{ fromIntegral } r1) (\text{fromIntegral } r2) \text{ seed } (r1.r2)
\]

\[
\text{where} \quad \text{sigma2} s f \text{ seed } (r1,r2)
\]

\[
| \text{ otherwise } = \text{error} \text{ ”sum of probabilities is not 1” }
\]

\[
\text{checkSum} :: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}
\]

\[
| c = \text{max} = f c
\]

\[
| \text{ otherwise } = f(c) + (\text{checkSum} f (c+1) \text{ max})
\]

\[
\text{findk} :: \text{Float} \rightarrow [\text{Float}] \rightarrow \text{Int}
\]

\[
\text{findk } r \text{ fs } = \text{findk1} 0 \ r \text{ fs}
\]

\[
\text{findk1 } k r (f:fs) \mid r < f = k
\]

\[
\mid \text{ otherwise } = \text{findk1} (k+1) \ r \text{ fs}
\]

\[
\text{findk1 } k [] = k
\]

\[
\text{mkdlist} :: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Float} \rightarrow [\text{Float}]
\]

\[
\text{mkdlist } f d = \text{scanl } (\text{sumf } f) 0.0 [1..d]
\]

\[
\text{sumf} :: (\text{Float} \rightarrow \text{Float}) \rightarrow \text{Float} \rightarrow \text{Float}
\]

42
\[ \text{sumf } g \times y = x + (g \ y) \]

For illustration consider the following example.

{-
    pdist :: Float \to Float
    pdist \ d = 1/(2**d)
    
    pdistsum \ 1 = \ pdist \ 1
    \ pdistsum \ d = (\ pdist \ d) + (\ pdistsum \ (d-1))
    
    pdistnorm :: Float \to Float \to Float
    \ pdistnorm \ dmax \ d = 1/((\ pdistsum \ dmax) * (2**d))
-
}
Chapter 12

The module FiringRules

12.1 Overview

```haskell
module FiringRules(
    FiringToken (Wild, Value)
)
```

where

This module contains the common `datatype` used in `DiscreteEvent` and `Dataflow` module.

The data type `FiringToken` defines the data type for tokens. The constructor `Wild` constructs a token wildcard, the constructor `Value a` constructs a token with value `a`.

```haskell
data FiringToken a = Wild
                      | Value a deriving (Eq, Show)
```

A sequence (pattern) matches a signal, if the sequence is a prefix of the signal. Table 12.1 illustrates the use of tokens in firing rules:

<table>
<thead>
<tr>
<th>Token</th>
<th>ForSyDe Expression</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>NullS</td>
<td>matches always</td>
</tr>
<tr>
<td>[*]</td>
<td>[Wild]</td>
<td>matches signal with at least one token</td>
</tr>
<tr>
<td>[v]</td>
<td>[Value v]</td>
<td>matches signal with <code>v</code> as its first value</td>
</tr>
<tr>
<td>[<em>,</em>]</td>
<td>[Wild, Wild]</td>
<td>matches signals with at least two tokens</td>
</tr>
</tbody>
</table>

Table 12.1: Matching of Tokens
Chapter 13

The module Vector

13.1 Overview

The module Vector defines the data type Vector and the corresponding functions. It is a development of the module Vector defined by Reekie in [3]. Though the vector is modeled as a list, it should be viewed as an array, i.e. a vector has a fixed size. Unfortunately, it is not possible to have the size of the vector as a parameter of the vector data type, due to restrictions in Haskell's type system. Still most operations are defined for vectors with the same size.

```haskell
module Vector (
  Vector (..), vector, fromVector, unitV, nullV, lengthV,
  atV, replaceV, headV, tailV, lastV, initV, takeV, dropV,
  selectV, groupV, (<<:), (<:<), mapV, foldlV, foldrV, scanlV,
  scanrV, meshlV, meshrV, zipWithV, filterV, zipV, unzipV,
  concatV, reverseV, shiftlV, shiftrV, rotlV, rotlV,
  generateV, iterateV, copyV
) where

infixr  5 :>
infixl  5 <:
infixr  5 <>
```

13.2 The Data Type Vector

The data type Vector is modeled similar to a list. It has two data type constructors. NullV constructs the empty vector, while (:::) a vector by adding an value to an existing vector. Using the inheritance mechanism of Haskell we have declared Vector as an instance of the classes Read and Show.

This means that the vector 1::2::3::4:: NullV is shown as [1,2,3,4].

```haskell
data Vector a =
  NullV
  | a :> (Vector a) deriving (Eq)
```

13.3 Functions on the Data Type vector

The function vector converts a list into a vector, while the function fromVector converts a vector into a list.

```haskell
vector       :: [a] -> Vector a
fromVector   :: Vector a -> [a]
```

45
The function `unitV` creates a vector with one element. The function `nullV` returns `True` if a vector is empty. The function `lengthV` returns the number of elements in a value. The function `atV` returns the n-th element in a vector, starting from zero. The function `replaceV` replaces an element in a vector:

\[
\begin{align*}
\text{unitV} & : a \rightarrow \text{Vector } a \\
\text{nullV} & : \text{Vector } a \rightarrow \text{Bool} \\
\text{lengthV} & : \text{Num } a \Rightarrow \text{Vector } b \rightarrow a \\
\text{replaceV} & : \text{Vector } a \rightarrow \text{Int} \rightarrow a \rightarrow \text{Vector } a \\
\text{atV} & : \text{Num } a \Rightarrow \text{Vector } b \rightarrow a \rightarrow b
\end{align*}
\]

The functions `headV` and `lastV` return the first element or the the last element of a vector. The functions `tailV` returns all, but the first element of a vector, while `initV` returns all but the last elements of a vector. The function `takeV` returns the first n elements of a vector while the function `dropV` drops the first n elements of a vector.

\[
\begin{align*}
\text{headV} & : \text{Vector } a \rightarrow a \\
\text{tailV} & : \text{Vector } a \rightarrow \text{Vector } a \\
\text{lastV} & : \text{Vector } a \rightarrow a \\
\text{initV} & : \text{Vector } a \rightarrow \text{Vector } a \\
\text{takeV} & : (\text{Num } a, \text{ Ord } a) \Rightarrow a \rightarrow a \rightarrow a \rightarrow \text{Vector } b \rightarrow \text{Vector } b \\
\text{dropV} & : (\text{Num } a, \text{ Ord } a) \Rightarrow a \rightarrow \text{Vector } b \rightarrow \text{Vector } b
\end{align*}
\]

The function `selectV` selects elements in the vector. The first argument gives the initial element, starting from zero, the second argument gives the stepsize between elements and the last argument gives the number of elements.

\[
\begin{align*}
\text{selectV} & : (\text{Num } a, \text{ Ord } a) \Rightarrow a \rightarrow a \rightarrow a \rightarrow \text{Vector } b \rightarrow \text{Vector } b
\end{align*}
\]

The function `groupV` groups a vector into a vector of vectors of size n.

\[
\begin{align*}
\text{groupV} & : (\text{Num } a, \text{ Ord } a) \Rightarrow a \rightarrow \text{Vector } b \rightarrow \text{Vector } (\text{Vector } b)
\end{align*}
\]

The data constructor `(:)` adds an element add the front of the vector, while the operator `(_:_)` adds an element at the end. The operator `(_+_)` concatenates two vectors. The function `concatV` concats a vector of vectors into a single vector.

\[
\begin{align*}
(\langle\rangle) & : \text{Vector } a \rightarrow \text{Vector } a \rightarrow \text{Vector } a \\
(\langle\_\rangle) & : \text{Vector } a \rightarrow a \rightarrow \text{Vector } a
\end{align*}
\]

The higher-order function `mapV` applies a function on all elements of a vector.

\[
\begin{align*}
\text{mapV} & : (a \rightarrow b) \rightarrow \text{Vector } a \rightarrow \text{Vector } b
\end{align*}
\]

The higher-order function `zipWithV` applies a function pairwise on to vectors.

\[
\begin{align*}
\text{zipWithV} & : (a \rightarrow b \rightarrow c) \rightarrow \text{Vector } a \rightarrow \text{Vector } b \rightarrow \text{Vector } c
\end{align*}
\]

The higher-order functions `foldPV` and `foldRV` fold a function from the right or from the left over a vector using an initial value.

\[
\begin{align*}
\text{foldPV} & : (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow \text{Vector } b \rightarrow a \\
\text{foldRV} & : (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow \text{Vector } b \rightarrow a
\end{align*}
\]

The higher-function `filterV` takes a predicate function and a vector and creates a new vector with the elements for which the predicate is true.

\[
\begin{align*}
\text{filterV} & : (a \rightarrow \text{Bool}) \rightarrow \text{Vector } a \rightarrow \text{Vector } a
\end{align*}
\]

The function `zipV` zips two vectors into a vector of tuples. The function `unzipV` unzips a vector of tuples into two vectors.

\[
\begin{align*}
\text{zipV} & : \text{Vector } a \rightarrow \text{Vector } b \rightarrow \text{Vector } (a, b) \\
\text{unzipV} & : \text{Vector } (a, b) \rightarrow (\text{Vector } a, \text{Vector } b)
\end{align*}
\]
The function \texttt{shiftV} shifts a value from the left into a vector. The function \texttt{shiftV}\texttt{rV} shifts a value from the right into a vector. The functions \texttt{rotV}, \texttt{rotV}\texttt{rV} rotates a vector to the left or to the right. Note that these functions do not change the size of a vector:

\begin{verbatim}
shiftV :: Vector a \rightarrow a \rightarrow Vector a
shiftV\texttt{rV} :: Vector a \rightarrow a \rightarrow Vector a
rotV :: Vector a \rightarrow Vector a
rotV\texttt{rV} :: Vector a \rightarrow Vector a
\end{verbatim}

The following functions are still undocumented: The function \texttt{concatV} transforms a vector of vectors to a single vector. The function \texttt{reverseV} reverses the order of elements in a vector:

\begin{verbatim}
concatV :: Vector (Vector a) \rightarrow Vector a
reverseV :: Vector a \rightarrow Vector a
\end{verbatim}

The function \texttt{iterateV} generates a vector with a given number of elements starting from an initial element using a supplied function for the generation of elements. The function \texttt{generateV} behaves in the same way, but starts with the application of the supplied function to the supplied value. The function \texttt{copyV} generates a vector with a given number of copies of the same element.

\begin{verbatim}
Vector> iterateV 5 (+1) 1
<1,2,3,4,5> :: Vector Integer
Vector> generateV 5 (+1) 1
<2,3,4,5,6> :: Vector Integer
Vector> copyV 7 5
<5,5,5,5,5,5,5> :: Vector Integer
\end{verbatim}

\texttt{iterateV} :: \texttt{Num a} \Rightarrow a \rightarrow (b \rightarrow b) \rightarrow b \rightarrow Vector b
\texttt{generateV} :: \texttt{Num a} \Rightarrow a \rightarrow (b \rightarrow b) \rightarrow b \rightarrow Vector b
\texttt{copyV} :: \texttt{Num a} \Rightarrow a \rightarrow b \rightarrow Vector b

The functions \texttt{scanlV} and \texttt{scanV} "scan" a function through a vector. The functions take an initial element apply a functions recursively first on the element and then on the result of the function application.

\begin{verbatim}
scanlV :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow Vector b \rightarrow Vector a
scanlV :: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow Vector b \rightarrow Vector a
\end{verbatim}

The following code is still not documented!

Reekie also proposed the \texttt{meshIV} and \texttt{meshV} iterators. They are like a combination of \texttt{mapV} and \texttt{scanlV} or \texttt{scanV}. The argument function supplies a pair of values: the first is input into the next application of this function, and the second is the output value. As an example consider the expression:

\begin{verbatim}
f x y = (x+y, x+y)
\end{verbatim}

\begin{verbatim}
s1 = vector [1,2,3,4,5]
\end{verbatim}

Here \texttt{meshIV} can be used to calculate the running sum.

\begin{verbatim}
Vector> meshIV f 0 s1
(15,<1,3,6,10,15>)
\end{verbatim}

\begin{verbatim}
meshIV :: (a \rightarrow b \rightarrow (a, c)) \rightarrow a \rightarrow Vector b \rightarrow (a, Vector c)
meshIVV :: (a \rightarrow b \rightarrow (c, b)) \rightarrow b \rightarrow Vector a \rightarrow (Vector c, b)
\end{verbatim}
13.4 Implementation

13.4.1 The Data Type vector

instance (Show a) ⇒ Show (Vector a) where
showsPrec p NullIV = showParen (p > 9) (showString "<")
showsPrec p xs = showParen (p > 9) (showChar '≤' . showVector1 xs)

where
  showVector1 NullIV
    = showChar '>'
  showVector1 (x:NullIV)
    = shows x . showChar '>'
  showVector1 (x:xs)
    = shows x . showChar '≤' . showVector1 xs

instance Read a ⇒ Read (Vector a) where
readsPrec _ s = readsVector s
readsVector :: (Read a) ⇒ ReadS (Vector a)
readsVector s = 
        [((x:NullIV), rest) | ("≤", r2) ← lex s,
            (x, r3) ← reads r2,
            (">", rest) ← lex r3]
        ++
        [(NullIV, r4) | ("≤", r5) ← lex s,
            (">", r4) ← lex r5]
        ++
        [((x:xs), r6) | ("≤", r7) ← lex s,
            (x, r8) ← reads r7,
            (".", r9) ← lex r8,
            (xs, r6) ← readsValues r9]
readsValues :: (Read a) ⇒ ReadS (Vector a)
readsValues s = 
        [((x:NullIV), r1) | (x, r2) ← reads s,
            (">", r1) ← lex r2]
        ++
        [((x:xs), r3) | (x, r4) ← reads s,
            (".", r5) ← lex r4,
            (xs, r3) ← readsValues r5]

13.4.2 Functions on the Data Type Vector

vector [] = NullIV
vector (x:xs) = x :> (vector xs)

fromVector NullIV = []
fromVector (x:xs) = x : fromVector xs

unitV x = x :> NullIV

nullIV NullIV = True
nullIV _ = False

lengthV NullIV = 0
lengthV (x:xs) = 1 + lengthV xs

replaceV vs n x
\[ n \leq \text{lengthV} \land n > 0 = \text{takeV} n \land vs \leftrightarrow \text{unitV} x \leftrightarrow \text{dropV} \ n+1 \land vs \\
\text{otherwise} \quad = \ vs
\]

\text{NullV} \quad \text{atV} = \text{error} "\text{Vector\&_{\_}has\_not\_enough\_elements}"

\( x:_m \) \quad \text{atV} 0 = x

\( x:_m \_m \) \quad \text{atV} n = x \cdot \text{atV} (n-1)

\text{headV} \quad \text{NullV} = \text{error} "\text{head\_\_Vector\_is\_empty}\"

\text{headV} (v:_m) = v

\text{tailV} \quad \text{NullV} = \text{error} "\text{tail\_\_Vector\_is\_empty}\"

\text{tailV} (v:_m) = vs

\text{lastV} \quad \text{NullV} = \text{error} "\text{last\_\_Vector\_is\_empty}\"

\text{lastV} (v:_m) = v

\text{lastV} (v:_m) = \text{lastV} vs

\text{initV} \quad \text{NullV} = \text{error} "\text{init\_\_Vector\_is\_empty}\"

\text{initV} (v:_m) = \text{NullV}

\text{initV} (v:_m) = v : \text{initV} vs

\text{takeV} 0 = \text{NullV}

\text{takeV} _m = \text{NullV}

\text{takeV} n (v:_m) = \begin{cases} 
\text{NullV} & n \leq 0 \\
\text{otherwise} = v : \text{takeV} (n-1) vs 
\end{cases}

\text{dropV} 0 = \text{NullV}

\text{dropV} _m = \text{ NullV}

\text{dropV} n (v:_m) = \begin{cases} 
\text{NullV} & n \leq 0 \\
\text{otherwise} = v : vs 
\end{cases}

\text{selectV} f s n vs = \begin{cases} 
\text{NullV} & n \leq 0 \\
\text{otherwise} = \text{selectV} \ f s \ n \ vs 
\end{cases}

\text{groupV} n v = \begin{cases} 
\text{lengthV} v < n = \text{NullV} \\
\text{otherwise} = \text{selectV} 0 1 n v : \text{groupV} n \ (\text{selectV} n 1 \ (\text{lengthV} v-n) v)
\end{cases}

\text{NullV} = \text{ys} = y s

\( x:_m \_m \) \quad \text{ys} = x : \text{(ys)}

\text{xs} = x : \text{xs} \leftrightarrow \text{unitV} x

\text{mapV} = \text{NullV}

\text{mapV} f (x:_m \_m) = f x : \text{mapV} f x

\text{zipWithV} f (x:_m \_m) (y:_m \_m) = f x y : \text{(zipWithV} f x y \_m)

\text{foldL} a \text{NullV} = a

\text{foldL} f a (x:_m \_m) = \text{foldL} f (f a x) x

\text{foldL} a \text{NullV} = a

\text{foldL} f a (x:_m \_m) = f x (\text{foldL} f a x)

\text{foldL} a \text{NullV} = a

\text{foldL} f a (x:_m \_m) = f x (\text{foldL} f a x)
filterV NullV = NullV
filterV p (v::vs) = if (p v) then
  v :: filterV p vs
else
  filterV p vs

zipV (x::xs) (y::ys) = (x, y) :: zipV xs ys
zipV NullV _ _ = NullV
unzipV NullV = (NullV, NullV)
unzipV ((x, y) :: xys) = (x::xs, y::ys) where (xs, ys) = unzipV xys

shiftlV vs v = v :: initV vs
shiftlV vs v = tailV vs :: v

rotrV NullV = NullV
rotrV vs = tailV vs :: headV vs

rotrlV NullV = NullV
rotrlV vs = lastV vs :: initV vs

concatV = foldlV (::) NullV

reverseV NullV = NullV
reverseV (v::vs) = reverseV vs :: v

generateV 0 _ _ = NullV
generateV n f a = x :: generateV (n-1) f x
  where x = f a

iterateV 0 _ _ = NullV
iterateV n f a = a :: iterateV (n-1) f (f a)

copyV k x = iterateV k id x

scanlV _ _ NullV = NullV
scanlV f a (x::xs) = q :: scanlV f q xs
  where q = f a x

scanrV _ _ NullV = NullV
scanrV f a (x::NullV) = f x a :: NullV
scanrV f a (x::xs) = f x y :: ys
  where ys@(y::_) = scanrV f a xs

meshlV a NullV = (a, NullV)
meshlV f a (x::xs) = (a', y::ys)
  where (a', y) = f a x
  (a'', ys) = meshlV f a' xs

meshrV a NullV = (NullV, a)
meshrV f a (x::xs) = (y::ys, a'')
  where (y, a'') = f x a'
  (ys, a') = meshrV f a xs
Chapter 14

The module Memory

14.1 Overview

This module contains the data structure and access functions for the memory model.

```haskell
module Memory
module AbsentExt, Memory (..), Access (..),
MemSize, Adr, newMem, memState, memOutput
where

import Vector
import AbsentExt
```

14.2 Data Structure

The data type Memory is modeled as a vector. The data type Access defines two access patterns, Read adr and Write adr val, where adr can be of any type.

```haskell
type Adr = Int

type MemSize = Int

data Memory a = Mem Adr (Vector (AbsentExt a))
  deriving (Eq, Show)

data Access a = Read Adr
  | Write Adr a
  deriving (Eq, Show)
```

14.3 Functions on the data type Memory

The module defines the following access functions for the memory:

```haskell
newMem :: MemSize → Memory a
memState :: Memory a → Access a → Memory a
memOutput :: Memory a → Access a → AbsentExt a
```

The function newMem creates a new memory, where the number of entries is given by a parameter. The function memState gives the new state of the memory, after an access to a memory. A Read operation leaves the memory unchanged. The function memOutput gives the output of the memory after an access to the memory. A Write operation gives an absent value as output.
14.4 Implementation of Functions

\[ \text{newMem size} = \text{Mem size (copyV size Abst)} \]

\[ \text{writeMem} :: \text{Memory a } \rightarrow (\text{Int}, a) \rightarrow \text{Memory a} \]
\[ \text{writeMem (Mem size vs) (i, x)} \]
\[ | i < \text{size} \&\& i \geq 0 = \text{Mem size (replaceV vs i (abstExt x))} \]
\[ | \text{otherwise} = \text{Mem size vs} \]

\[ \text{readMem} :: \text{Memory a } \rightarrow \text{Int } \rightarrow (\text{AbstExt a}) \]
\[ \text{readMem (Mem size vs) i} \]
\[ | i < \text{size} \&\& i \geq 0 = \text{vs 'atV' i} \]
\[ | \text{otherwise} = \text{Abst} \]

\[ \text{memState mem (Read _)} = \text{mem} \]
\[ \text{memState mem (Write i x)} = \text{writeMem mem (i, x)} \]

\[ \text{memOutput mem (Read i)} = \text{readMem mem i} \]
\[ \text{memOutput _ (Write _ _)} = \text{Abst} \]
Chapter 15

The module Queue

15.1 Overview

The module Queue provides two data types, that can be used to model queue structures, such as FIFOs. There is a data type for an queue of infinite size Queue and one for finite size FiniteQueue.

15.2 The data type Queue

A queue is modeled as a list. The data type FiniteQueue has an additional parameter, that determines the size of the queue.

```
module Queue where

import AbsentExt

data Queue a = Q [a] deriving (Eq, Show)
data FiniteQueue a = FQ Int [a] deriving (Eq, Show)
```

15.3 Functions on the data types Queue and FiniteQueue

Table 15.3 shows the functions on the data types Queue and FiniteQueue.

<table>
<thead>
<tr>
<th>infinite</th>
<th>finite</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushQ</td>
<td>pushQ</td>
<td>pushes one element on the queue</td>
</tr>
<tr>
<td>pushListQ</td>
<td>pushListQ</td>
<td>pushes a list of elements on the queue</td>
</tr>
<tr>
<td>popQ</td>
<td>popQ</td>
<td>pops one element from the queue</td>
</tr>
<tr>
<td>queue</td>
<td>finiteQueue</td>
<td>transforms a list into a queue</td>
</tr>
</tbody>
</table>

Table 15.1: Functions on the data types Queue and FiniteQueue
15.4 Implementation

\[
\begin{align*}
\text{pushQ} \ (Q \ q) \ x &= Q \ (q \mapsto [x]) \\
\text{pushListQ} \ (Q \ q) \ xs &= Q \ (q \mapsto xs) \\
\text{popQ} \ (Q \ []) &= ([], \ \text{Abst}) \\
\text{popQ} \ (Q \ (x:xs)) &= (Q \ xs, \ \text{Prest} \ x) \\
\text{queue} \ xs &= Q \ xs \\
\text{pushFQ} \ (FQ \ n \ q) \ x &= \begin{cases} 
Q \ n \ (q \mapsto [x]) & \text{if } \text{length} \ q < n \\
Q \ n \ (q \mapsto [x]) & \text{else}
\end{cases} \\
\text{pushListFQ} \ (FQ \ n \ q) \ xs &= Q \ n \ (\text{take} \ n \ (q \mapsto xs)) \\
\text{popFQ} \ (FQ \ n \ []) &= (FQ \ n \ [], \ \text{Abst}) \\
\text{popFQ} \ (FQ \ n \ (q:qs)) &= (FQ \ n \ qs, \ \text{Prest} \ q) \\
\text{finiteQueue} \ n \ xs &= Q \ n \ (\text{take} \ n \ xs)
\end{align*}
\]
Chapter 16

Possible Extensions to the ForSyDe Standard Library

16.1 Overview

This module includes proposed extensions to the ForSyDe standard library. The Extensions are structured according to their corresponding library. The idea is, that this module serves as a collection of candidates of functions that can be included into the ForSyDe standard library in later releases.

This means, that this library is \textit{not} stable!

\begin{verbatim}
module ForSyDeStdLibExtensions where

import ForSyDeStdLib
\end{verbatim}
Bibliography

