

# The ForSyDe Standard Library\*

Ingo Sander

24th April 2003

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	The Module <code>ForSyDeStdLib</code> . . . . .	3
1.1.1	Overview . . . . .	3
<b>2</b>	<b>ForSyDe Core Language</b>	<b>4</b>
2.1	The Module <code>Signal</code> . . . . .	4
2.1.1	Overview . . . . .	4
2.1.2	The Data Type <code>Signal</code> . . . . .	4
2.1.3	Functions on the Data Type <code>Signal</code> . . . . .	4
2.2	The Module <code>Vector</code> . . . . .	8
2.2.1	Overview . . . . .	8
2.2.2	The Data Type <code>Vector</code> . . . . .	8
2.2.3	Functions on the Data Type <code>vector</code> . . . . .	8
2.2.4	Implementation . . . . .	11
2.3	The Module <code>AbsentExt</code> . . . . .	14
2.3.1	Overview . . . . .	14
2.3.2	Functions on the Data Type <code>AbsentExt</code> . . . . .	15
2.3.3	Implementation of Library Functions . . . . .	15
2.4	The Module <code>Combinators</code> . . . . .	16
2.4.1	Overview . . . . .	16
<b>3</b>	<b>Libraries of System Functions and Data Types</b>	<b>16</b>
3.1	The Module <code>Memory</code> . . . . .	16
3.1.1	Overview . . . . .	16
3.1.2	Data Structure . . . . .	17
3.1.3	Functions on the data type <code>Memory</code> . . . . .	17
3.1.4	Implementation of Functions . . . . .	17
3.2	The Module <code>Queue</code> . . . . .	18
3.2.1	Overview . . . . .	18
3.2.2	The data type <code>Queue</code> . . . . .	18

---

\*This document describes version 2.3, which only includes the modules that have been used for [5]

3.2.3	Functions on the data types <code>Queue</code> and <code>FiniteQueue</code> . . . .	18
3.2.4	Implementation . . . . .	19
3.3	The Module <code>DFT</code> . . . . .	19
<b>4</b>	<b>Computational Model Libraries</b>	<b>21</b>
4.1	The Module <code>SynchronousLib</code> . . . . .	21
4.1.1	Overview . . . . .	21
4.1.2	Process Constructors for Combinatorial Processes . . . . .	22
4.1.3	Process Constructors for Sequential Processes . . . . .	22
4.1.4	Processes . . . . .	24
4.1.5	Implementation of Library Functions . . . . .	24
4.2	The Module <code>DomainInterfaces</code> . . . . .	27
4.2.1	Overview . . . . .	27
4.2.2	Domain Interface Constructors . . . . .	28
4.2.3	Implementation . . . . .	28
<b>5</b>	<b>Application Libraries</b>	<b>30</b>
5.1	The Module <code>SynchronousProcessLib</code> . . . . .	30
5.1.1	Overview . . . . .	30
5.1.2	Processes . . . . .	30
5.1.3	Implementation of Processes . . . . .	31
5.2	The Module <code>FIR</code> . . . . .	32

## 1 Introduction

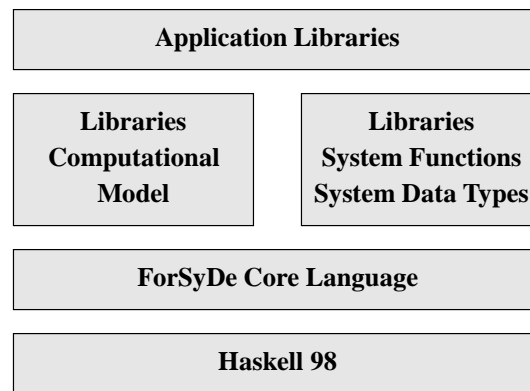


Figure 1: The ForSyDe Standard Library

The ForSyDe Standard Library consists of several layers as illustrated in Figure 1. The bottom layer is the Haskell 98 language [2]. The layer above Haskell 98 defines the *ForSyDe Core Language*. Here the fundamental data types, such as signal and vector, and the corresponding functions are defined. Computational models are defined in a

*Computational Model Library* and are located on top of the core language. Also on top of the core language there are the *Libraries of System Functions and Data Types*, which contains functions and data types that are typical for system applications and are independent of the computational model. The top layer of the ForSyDe Standard Library consists of *Application Libraries*. These libraries include components and functions that are modeled for specific computational models.

The ForSyDe Standard Library can be imported with

```
import ForSyDeStdLib
```

which includes all sub-modules of the library.

This appendix covers only the parts of the library that are used in this thesis. For preliminary versions of other computational models or the stochastic library see the ForSyDe web page [1].

The code is written in literate Haskell style, which makes it possible to include  $\LaTeX$  code for documentation. Only those parts of the literate program that are entirely enclosed between `\begin{code} . . . \end{code}` are treated as program text; all other lines are comments. This allows to include usual  $\LaTeX$  text, but also figures and equations as comments.

## 1.1 The Module ForSyDeStdLib

### 1.1.1 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 all other libraries.

```
module ForSyDeStdLib(  
    module DomainInterfaces,  
    module SynchronousProcessLib,  
    module SynchronousLib,  
    module Vector, module Signal, module Memory,  
    module AbsentExt, module Queue,  
    module Combinators, module DFT,  
    module FIR  
    ) where  
  
import DomainInterfaces  
import SynchronousProcessLib  
import SynchronousLib  
import Vector  
import Signal  
import Memory  
import AbsentExt  
import Queue  
import Combinators  
import FIR  
import DFT
```

## 2 ForSyDe Core Language

The ForSyDe core language includes the modules `Signal`, `Vector`, `AbsentExt` and `Combinators`.

### 2.1 The Module `Signal`

#### 2.1.1 Overview

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

```
module Signal( Signal (NullS, (:-)), (-:), (+++), (!-),
              signal, fromSignal,
              unitS, nullS, headS, tailS, atS, takeS, dropS,
              lengthS, infiniteS, copyS, selectS, writeS, readS
            )
where

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

#### 2.1.2 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.

```
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}`.

#### 2.1.3 Functions on the Data Type `signal`

The module defines the following functions on the data type `Signal`:

```
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
selectS    :: Int -> Int -> Signal a -> Signal a
lengthS    :: Num a => Signal b -> a
```

```

infiniteS      :: (a -> a) -> a -> Signal a
writeS         :: Show a => Signal a -> [Char]
readS          :: Read a => [Char] -> Signal a
(-:)          :: Signal a -> a -> Signal a
(+++)         :: Signal a -> 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` 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. The function `selectS` takes three parameters, an offset, a stepsize and a signal and returns some elements of the signal such as in the following example:

```

Signal> selectS 2 3 (signal[1,2,3,4,5,6,7,8,9,10])
{3,6,9} :: Signal Integer

```

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 operator `(-:)` adds an element to a signal at the tail. Finally the operator `(+++)` concatenates two signals into one signal. The function `lengthS` returns the length of a *finite* stream. The function `infiniteS` creates an infinite signal. The first argument `f` is a function that is applied on the current value. The second argument `x` gives the first value of the signal.

```

Signal> takeS 5 (infiniteS (*3) 1)
{1,3,9,27,81} :: Signal Integer

```

The function `copyS` creates a signal with  $n$  values `x`. The function `writeS` transforms a signal into a string of the following format:

```

Signal> writeS (signal[1,2,3,4,5])
"1\n2\n3\n4\n5\n" :: [Char]

```

The function `readS` transforms such a formatted string into a signal.

```

Signal> readS "1\n2\n3\n4\n5\n" :: Signal Int
{1,2,3,4,5} :: Signal Int

```

The combinator `fanS` takes two processes `p1` and `p2` 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)

```

```

instance (Show a) => Show (Signal a) where
  showsPrec p NullS = showParen (p > 9) (
    showString "{}")
  showsPrec p xs    = showParen (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) => ReadS (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]

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

(!-) xs 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

selectS offset step xs = select1S step (dropS offset xs)
where
    select1S step NullS = NullS
    select1S step (x:-xs) = x :- select1S step (dropS (step-1) xs)

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

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

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

infiniteS f x         = x :- infiniteS f (f x)

copyS 0 x              = NullS
copyS n x              = x :- copyS (n-1) x

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

writeS NullS          = []
writeS (x:-xs) = show x ++ "\n" ++ writeS xs

readS xs              = readS' (words xs)
where
    readS' []          = NullS
    readS' ("\n":xs) = readS' xs
    readS' (x:xs)     = read x :- readS' xs

```

## 2.2 The Module Vector

### 2.2.1 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 [4]. 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.

```
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, rotrV, rotlV,
    generateV, iterateV, copyV, serialV, parallelV )
  where

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

### 2.2.2 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:>NullV` is shown as `<1, 2, 3>`.

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

### 2.2.3 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.

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

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.

```
unitV    :: a -> Vector a
nullV    :: Vector a -> Bool
```

```
lengthV :: Num a => Vector b -> a
replaceV :: Vector a -> Int -> a -> Vector a
atV      :: Num a => Vector b -> a -> b
```

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.

```
headV :: Vector a -> a
tailV :: Vector a -> Vector a
lastV :: Vector a -> a
initV :: Vector a -> Vector a
takeV :: (Num a, Ord a) => a -> Vector b -> Vector b
dropV :: (Num a, Ord a) => a -> Vector b -> Vector b
```

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.

```
selectV :: (Num a, Ord a) => a -> a -> a -> Vector b -> Vector b
```

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

```
groupV :: (Num a, Ord a) => a -> Vector b -> Vector (Vector b)
```

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` concatenates a vector of vectors into a single vector.

```
(<+>) :: Vector a -> Vector a -> Vector a
(<: ) :: Vector a -> a -> Vector a
```

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

```
mapV :: (a -> b) -> Vector a -> Vector b
```

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

```
zipWithV :: (a -> b -> c) -> Vector a -> Vector b -> Vector c
```

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

```
foldlV :: (a -> b -> a) -> a -> Vector b -> a
foldrV :: (b -> a -> a) -> a -> Vector b -> a
```

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.

```
filterV :: (a -> Bool) -> Vector a -> Vector a
```

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

```
zipV   :: Vector a -> Vector b -> Vector (a, b)
unzipV :: Vector (a, b) -> (Vector a, Vector b)
```

The function `shiftLV` shifts a value from the left into a vector. The function `shiftrV` shifts a value from the right into a vector. The functions `rotLV`, `rotrV` rotates a vector to the left or to the right. Note that these functions do not change the size of a vector.

```
shiftLV :: Vector a -> a -> Vector a
shiftrV :: Vector a -> a -> Vector a
rotrV    :: Vector a -> Vector a
rotLV    :: Vector a -> Vector a
```

The function `concatV` transforms a vector of vectors to a single vector. The function `reverseV` reverses the order of elements in a vector.

```
concatV  :: Vector (Vector a) -> Vector a
reverseV :: Vector a -> Vector a
```

The function `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 `generateV` behaves in the same way, but starts with the application of the supplied function to the supplied value. The function `copyV` generates a vector with a given number of copies of the same element.

```
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
```

```
iterateV :: Num a => a -> (b -> b) -> b -> Vector b
generateV :: Num a => a -> (b -> b) -> b -> Vector b
copyV     :: Num a => a -> b -> Vector b
```

The functions `serialV` and `parallelV` can be used to construct serial and parallel networks of processes.

```
serialV      :: Vector (a -> a) -> a -> a
parallelV    :: Vector (a -> b) -> Vector a -> Vector b
```

The functions `scanLV` and `scanrV` "scan" a function through a vector. The functions take an initial element apply a function recursively first on the element and then on the result of the function application.

```
scanLV :: (a -> b -> a) -> a -> Vector b -> Vector a
scanrV :: (b -> a -> a) -> a -> Vector b -> Vector a
```

Reekie also proposed the `meshLV` and `meshrV` iterators. They are like a combination of `mapV` and `scanLV` or `scanrV`. 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:

```
f x y = (x+y, x+y)

s1 = vector [1,2,3,4,5]
```

Here `mesh1V` can be used to calculate the running sum.

```
Vector> mesh1V f 0 s1
(15,<1,3,6,10,15>)
```

```
mesh1V  :: (a -> b -> (a, c)) -> a -> Vector b -> (a, Vector c)
meshrV  :: (a -> b -> (c, b)) -> b -> Vector a -> (Vector c, b)
```

## 2.2.4 Implementation

```
instance (Show a) => Show (Vector a) where
  showsPrec p NullV = showParen (p > 9) (
    showString "<>")
  showsPrec p xs    = showParen (p > 9) (
    showChar '<' . showVector1 xs)
  where
    showVector1 NullV
      = showChar '>'
    showVector1 (x:>NullV)
      = 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:>NullV), rest] | ("<", r2) <- lex s,
                                   (x, r3)  <- reads r2,
                                   (">", rest) <- lex r3]
  ++
  [(NullV, 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:>NullV), r1] | (x, r2) <- reads s,
                                   (">", r1) <- lex r2]
  ++
  [(x:>xs), r3] | (x, r4) <- reads s,
                 ("", r5) <- lex r4,
                 (xs, r3) <- readsValues r5]
```

```

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

fromVector NullV = []
fromVector (x:>xs) = x : fromVector xs

unitV x = x :> NullV

nullV NullV = True
nullV _     = False

lengthV NullV = 0
lengthV (_:>xs) = 1 + lengthV xs

replaceV vs n x
  | n <= lengthV vs && n >= 0 = takeV n vs <+> unitV x
                               <+> dropV (n+1) vs
  | otherwise                 = vs

NullV `atV` _ = error "atV: _Vector_has_not_enough_elements"
(x:>_) `atV` 0 = x
(_:>xs) `atV` n = xs `atV` (n-1)

headV NullV = error "headV: _Vector_is_empty"
headV (v:>_) = v

tailV NullV = error "tailV: _Vector_is_empty"
tailV (_:>vs) = vs

lastV NullV = error "lastV: _Vector_is_empty"
lastV (v:>NullV) = v
lastV (_:>vs) = lastV vs

initV NullV = error "initV: _Vector_is_empty"
initV (_:>NullV) = NullV
initV (v:>vs) = v :> initV vs

takeV 0 _ = NullV
takeV _ NullV = NullV
takeV n (v:>vs) | n <= 0 = NullV
                | otherwise = v :> takeV (n-1) vs

dropV 0 vs = vs
dropV _ NullV = NullV
dropV n (v:>vs) | n <= 0 = v :> vs
                | otherwise = dropV (n-1) vs

selectV f s n vs | n <= 0
                  = NullV
                  | (f+s*n-1) > lengthV vs

```

```

        = error "selectV: _Vector_has_not_enough_elements"
    | otherwise
      = atV vs f :> selectV (f+s) s (n-1) vs

groupV n v
  | lengthV v < n = NullV
  | otherwise     = selectV 0 1 n v
                  :> groupV n (selectV n 1 (lengthV v-n) v)

NullV <+> ys = ys
(x:>xs) <+> ys = x :> (xs <+> ys)

xs <: x = xs <+> unitV x

mapV _ NullV = NullV
mapV f (x:>xs) = f x :> mapV f xs

zipWithV f (x:>xs) (y:>ys) = f x y :> (zipWithV f xs ys)
zipWithV _ _ _ = NullV

foldlV _ a NullV = a
foldlV f a (x:>xs) = foldlV f (f a x) xs

foldrV _ a NullV = a
foldrV f a (x:>xs) = f x (foldrV f a xs)

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

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

shifflV vs v = v :> initV vs

shiftrV vs v = tailV vs <: v

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

rotlV NullV = NullV
rotlV vs = lastV vs :> initV vs

concatV = foldrV (<+>) 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

serialV fs      x = serialV' (reverseV fs ) x
  where
    serialV' NullV  x = x
    serialV' (f:>fs) x = serialV fs (f x)

parallelV NullV  NullV = NullV
parallelV _      NullV
  = error "parallelV: Vectors have not the same size!"
parallelV NullV  _
  = error "parallelV: Vectors have not the same size!"
parallelV (f:>fs) (x:>xs) = f x := parallelV fs xs

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

```

## 2.3 The Module `AbsentExt`

### 2.3.1 Overview

The module `AbsentExt` is used to extend existing data types with the value "absent" ( $\perp$ ).

```

module AbsentExt(
    AbstExt (Abst, Prst), fromAbstExt, abstExt, psi,
    isAbsent, isPresent, abstExtFunc)
where

```

The data type `AbstExt` has two constructors. The constructor `Abst` is used to model the absence of a value, while the constructor `Prst` is used to model present values.

```

data AbstExt a          = Abst
                        | Prst a deriving (Eq)

```

The data type `AbstExt` is defined as an instance of `Show` and `Read`. `'_'` represents the value `Abst` while a present value is represented with its value, e.g. `Prst 1` is represented as `'1'`.

### 2.3.2 Functions on the Data Type `AbsentExt`

The module defines the following functions:

```

fromAbstExt      :: a -> AbstExt a -> a
isPresent        :: AbstExt a -> Bool
isAbsent         :: AbstExt a -> Bool
abstExtFunc      :: (a -> b) -> AbstExt a -> AbstExt b
psi              :: (a -> b) -> AbstExt a -> AbstExt b

```

The function `abstExt` converts a value into an extended value. The function `fromAbstExt` converts a value from a extended value.

The functions `isPresent` and `isAbsent` check for the presence or absence of a value.

The function `abstExtFunc` extends a function in order to process absent extended values. If the input is  $\perp$ , the output will also be  $\perp$ . The function `psi` is identical to `abstExtFunc` and should be used in future.

### 2.3.3 Implementation of Library Functions

```

instance Show a => Show (AbstExt a) where
    showsPrec _ x      = showsAbstExt x

showsAbstExt Abst      = (++) "_"
showsAbstExt (Prst x)  = (++) (show x)

instance Read a => Read (AbstExt a) where
    readsPrec _ x      = readsAbstExt x

readsAbstExt           :: (Read a) => Reads (AbstExt a)
readsAbstExt s         = [(Abst, r1) | ("_", r1) <- lex s]
                        ++ [(Prst x, r2) | (x, r2) <- reads s]

abstExt v              = Prst v

```

```

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

isPresent Abst             = False
isPresent (Prst _)        = True

isAbsent                    = not . isPresent

abstExtFunc f              = f'
  where f' Abst             = Abst
        f' (Prst x)        = Prst (f x)

psi                         = abstExtFunc

```

## 2.4 The Module Combinators

### 2.4.1 Overview

The module contains operators for function (sequential) composition and parallel composition.

```
module Combinators where
```

```

funComb1                    = (.)

funComb2 p1 p2             = p
  where p s1 s2             = p1 (p2 s1 s2)

funComb3 p1 p2             = p
  where p s1 s2 s3         = p1 (p2 s1 s2 s3)

funComb4 p1 p2             = p
  where p s1 s2 s3 s4     = p1 (p2 s1 s2 s3 s4)

parComb p1 p2              = p
  where p s1 s2            = (p1 s1, p2 s2)

```

## 3 Libraries of System Functions and Data Types

### 3.1 The Module Memory

#### 3.1.1 Overview

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

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

```
import Vector
import AbsentExt
```

### 3.1.2 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..

```
type Adr                = Int
type MemSize           = Int

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

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

### 3.1.3 Functions on the data type Memory

The module defines the following access functions for the memory:

```
newMem                :: MemSize -> Memory a
memState              :: Memory a -> Access a -> Memory a
memOutput            :: Memory a -> Access a -> AbstExt 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.

### 3.1.4 Implementation of Functions

```
newMem size          = Mem size (copyV size Abst)

writeMem             :: Memory a -> (Int, a) -> Memory a
writeMem (Mem size vs) (i, x)
  | i < size && i >= 0 = Mem size (replaceV vs i (abstExt x))
  | otherwise         = Mem size vs

readMem             :: Memory a -> Int -> (AbstExt a)
readMem (Mem size vs) i
  | i < size && i >= 0 = vs `atV` i
  | otherwise         = Abst

memState mem (Read _) = mem
memState mem (Write i x) = writeMem mem (i, x)
```

infinite	finite	description
pushQ	pushFQ	pushes one element on the queue
pushListQ	pushListFQ	pushes a list of elements on the queue
popQ	popFQ	pops one element from the queue
queue	finiteQueue	transforms a list into a queue

Table 1: Functions on the data types `Queue` and `FiniteQueue`

```
memOutput mem (Read i)      = readMem mem i
memOutput _   (Write _ _)  = Abst
```

## 3.2 The Module `Queue`

### 3.2.1 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`.

### 3.2.2 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)
```

### 3.2.3 Functions on the data types `Queue` and `FiniteQueue`

Table 3.2.3 shows the functions on the data types `Queue` and `FiniteQueue`.

```
pushQ           :: Queue a -> a -> Queue a
pushListQ      :: Queue a -> [a] -> Queue a
popQ           :: Queue a -> (Queue a, AbsentExt a)
queue          :: [a] -> Queue a
pushFQ         :: FiniteQueue a -> a -> FiniteQueue a
pushListFQ     :: FiniteQueue a -> [a] -> FiniteQueue a
popFQ          :: FiniteQueue a
               -> (FiniteQueue a, AbsentExt a)
finiteQueue    :: Int -> [a] -> FiniteQueue a
```

### 3.2.4 Implementation

```

pushQ (Q q) x           = Q (q ++ [x])

pushListQ (Q q) xs      = Q (q ++ xs)

popQ (Q [])             = (Q [], Abst)
popQ (Q (x:xs))        = (Q xs, Prst x)

queue xs                = Q xs

pushFQ (FQ n q) x      = if length q < n then
                        (FQ n (q ++ [x]))
                        else
                        (FQ n q)

pushListFQ (FQ n q) xs = FQ n (take n (q ++ xs))

popFQ (FQ n [])        = (FQ n [], Abst)
popFQ (FQ n (q:qs))   = (FQ n qs, Prst q)

finiteQueue n xs       = FQ n (take n xs)

```

### 3.3 The Module DFT

The module includes the standard Discrete Fourier Transform (DFT) function, which is formulated as

$$X(K) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \quad 0 \leq k \leq N-1 \quad (1)$$

where

$$W_N = e^{-j2\pi/N} \quad (2)$$

and a fast Fourier transform (FFT) algorithm, for computing the DFT, when the size  $N$  is a power of 2.

**module** DFT(dft, fft) **where**

```

import Signal
import Vector
import Complex

```

Here follows the ForSyDe implementation of the DFT:

```

dft bigN x | bigN == (lengthV x) = mapV (bigX_k bigN x) (nVector x)
          | otherwise           = error "DFT: _vector_has_not_the_right_size!"
where
  nVector x          = iterateV (lengthV x) (+1) 0
  bigX_k bigN x k   = sumV (zipWithV (*) x (bigW k bigN))

```

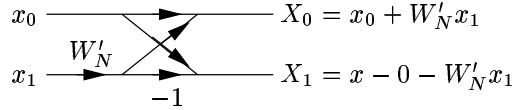


Figure 2: Basic butterfly computation in the decimation-in-time algorithm

```

bigW k bigN    = mapV (** k) (mapV cis (fullcircle bigN))
sumV           = foldlV (+) (0:+ 0)

fullcircle :: Integer -> Vector Double
fullcircle n = fullcircle1 0 (fromInteger n) n
  where
    fullcircle1 l m n
      | l == m    = NullV
      | otherwise = -2*pi*1/(fromInteger n)
                    :> fullcircle1 (l+1) m n

```

Here follows the Radix 2-FFT algorithm (decimation in time) implementation [3]. It is a divide and conquer algorithm, which reuses the calculation of the basic butterfly (Figure 2):

$$\begin{aligned}
 X(k) &= \sum_{m=0}^{(N/2)-1} f_{\text{even}}(m)W_{N/2}^{km} + W_N^k \sum_{m=0}^{(N/2)-1} f_{\text{odd}}(m)W_{N/2}^{km} \\
 &= F_{\text{even}}(k) + W_N^k F_{\text{odd}}(k) \quad k = 0, 1, \dots, N-1
 \end{aligned} \tag{3}$$

where

$$\begin{aligned}
 f_{\text{even}}(n) &= x(2n) \\
 f_{\text{odd}}(n) &= x(2n+1)
 \end{aligned}$$

The calculation of an eight-point FFT is illustrated in Figure 3.

```

fft bigN xv | bigN == (lengthV xv) = mapV (bigX xv) (kVector bigN)
            | otherwise = error "FFT: _vector_has_not_the_right_size!"

kVector bigN = iterateV bigN (+1) 0

bigX (x0:>x1:>NullV) k | even k = x0 + x1 * bigW 2 0
                    | odd k  = x0 - x1 * bigW 2 0
bigX xv k = bigF_even k + bigF_odd k * bigW bigN (fromInteger k)
  where bigF_even k = bigX (evens xv) k
        bigF_odd k = bigX (odds xv) k
        bigN = lengthV xv

bigW bigN k = cis (-2 * pi * (fromInteger k) / (fromInteger bigN))

evens NullV      = NullV
evens (v1:>NullV) = v1 :> NullV
evens (v1:>_:>v) = v1 :> evens v

```

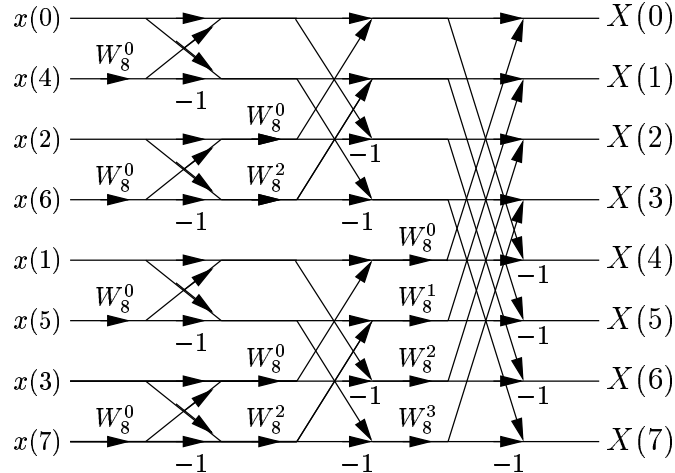


Figure 3: Eight-point decimation-in-time FFT algorithm

```
odds NullV      = NullV
odds (_:>NullV) = NullV
odds (_:>v2:>v) = v2 :> odds v
```

## 4 Computational Model Libraries

### 4.1 The Module SynchronousLib

#### 4.1.1 Overview

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

```
module SynchronousLib(
    module Vector, module Signal, module AbsentExt,
    mapSY, zipWithSY, zipWith3SY,
    zipWith4SY, zipWithxSY, scanlSY,
    scanl2SY, scanl3SY, scanldSY, scanld2SY,
    scanld3SY, delaySY, delaynSY, whenSY,
    fillSY, holdSY, zipSY, zip3SY, zip4SY, unzipSY,
    unzip3SY, unzip4SY, zipxSY, unzipxSY, mapxSY,
    mooreSY, moore2SY, moore3SY, mealySY, mealy2SY,
    mealy3SY, fstSY, sndSY, sourceSY
) where
```

```
import Signal
import Vector
import AbsentExt
```

#### 4.1.2 Process Constructors 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 process constructors 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
zipWithxSY :: (Vector a -> b) -> Vector (Signal a) -> Signal b
mapxSY     :: (a -> b) -> Vector (Signal a) -> Vector (Signal b)
```

The process constructor `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 process constructor `zipWithxSY` works as `zipWithSY`, but works with a vector of signals as input.

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

#### 4.1.3 Process Constructors for Sequential Processes

Sequential processes have a local state. Process Constructors 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 process constructors for sequential processes:

```
delaySY    :: a -> Signal a -> Signal a
delaynSY   :: a -> Int -> Signal a -> Signal a
scanlSY    :: (a -> b -> a) -> a -> Signal b -> Signal a
scanl2SY   :: (a -> b -> c -> a) -> a -> Signal b -> Signal c
           -> Signal a
scanl3SY   :: (a -> b -> c -> d -> a) -> a -> Signal b
           -> Signal c -> Signal d -> Signal a
scanldSY   :: (a -> b -> a) -> a -> Signal b -> Signal a
scanld2SY  :: (a -> b -> c -> a) -> a -> Signal b -> Signal c
```

```

scanld3SY      :: (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
sourceSY      :: (a -> a) -> a -> Signal a
filterSY      :: (a -> Bool) -> Signal a -> Signal (AbstExt a)

```

The process constructor `delaySY` delays the signal one event cycle by introducing an initial value at the beginning of the output signal. The process constructor `delaynSY` delays the signal  $n$  events by introducing  $n$  identical default values.

We define two different basic process constructors to construct sequential processes, `scanlSY` and `scanldSY`. Both process constructors take a function `ns` and a state `m` as arguments. Both process constructors use the function `ns` 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 `scanldSY` has the current state value as output. The following example exemplifies this:

```

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

```

Process Constructors like `scanl2SY`, `scanl2DelaySY` are used in the same way for several input signals.

The process constructors `mooreSY` and `mealySY` are used to model state machines. These process constructors are based on the process constructor `scanldSY` as is naturally for state machines in hardware, that the output operates on the current state and not on the next state.

These process constructors take a function `ns` 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` process constructor 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` process constructor operates on both the current state and the input.

The process `sourceSY` takes a function  $f$  and an initial state  $s_0$  and generates an infinite signal starting with the initial state  $s_0$  as first output followed by the recursive application of  $f$  on the current state which also serve as output values. The process that has the infinite signal of natural numbers as output is constructed by

```
SynchronousLib> takesS 5 (sourceSY (+1) 0)
{0,1,2,3,4} :: Signal Integer
```

The process constructor `filterSY` takes a predicate `p` and produces a process, that discards all values that do not fulfill the predicate `p`. In this case the output is  $\perp$ .

#### 4.1.4 Processes

The module also contains the following synchronous processes:

```
whenSY      :: Signal (AbstExt a) -> Signal (AbstExt b)
              -> Signal (AbstExt a)
fillSY      :: a -> Signal (AbstExt a) -> Signal a
holdSY      :: a -> Signal (AbstExt a) -> Signal a
zipSY       :: Signal a -> Signal b -> Signal (a,b)
zip3SY      :: Signal a -> Signal b -> Signal c -> Signal (a,b,c)
zip4SY      :: Signal a -> Signal b -> Signal c -> Signal d
              -> Signal (a,b,c,d)
unzipSY     :: Signal (a,b) -> (Signal a,Signal b)
unzip3SY    :: Signal (a, b, c) -> (Signal a, Signal b, Signal c)
unzip4SY    :: Signal (a,b,c,d)
              -> (Signal a,Signal b,Signal c,Signal d)
zipxSY      :: Vector (Signal a) -> Signal (Vector a)
unzipxSY    :: Signal (Vector a) -> Vector (Signal a)
fstSY       :: Signal (a,b) -> Signal a
sndSY       :: Signal (a,b) -> Signal b
```

The process constructor `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  $\perp$  when the event has an absent value.

The process constructor `fillSY` creates a process that 'fills' a signal with timed values by replacing absent values with a given value.

The process constructor `holdSY` creates a process that 'fills' a signal with values by replacing absent values by the preceding present value. Only in cases, where no preceding value exists, the absent value is replaced by the supplied 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`, `zip4SY`, `unzip4SY` and `unzip4SY` perform the corresponding function for three and four signals.

The process `zipxSY` 'zip' a signal of vectors into a vector of signals. The process `unzipxSY` 'unzips' a vector of signals into a signal of vectors.

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

#### 4.1.5 Implementation of Library Functions

```
mapSY _ NullS = NullS
mapSY f (x:-xs) = f x :- (mapSY f xs)
```

```

zipWithSY _ NullS _ = NullS
zipWithSY _ _ NullS = NullS
zipWithSY f (x:-xs) (y:-ys) = f x y :- (zipWithSY f xs ys)

zipWith3SY _ NullS _ _ = NullS
zipWith3SY _ _ NullS _ = NullS
zipWith3SY _ _ _ NullS = NullS
zipWith3SY f (x:-xs) (y:-ys) (z:-zs) = f x y z
                                         :- (zipWith3SY f xs ys zs)

zipWith4SY _ NullS _ _ _ = NullS
zipWith4SY _ _ NullS _ _ = NullS
zipWith4SY _ _ _ NullS _ = NullS
zipWith4SY _ _ _ _ NullS = NullS
zipWith4SY f (w:-ws) (x:-xs) (y:-ys) (z:-zs)
              = f w x y z
              :- (zipWith4SY f ws xs ys zs)

zipWithxSY f = mapSY f . zipxSY

mapxSY f = mapV (mapSY f)

scanlSY _ _ NullS = NullS
scanlSY f mem (x:-xs) = f mem x :- (scanlSY f newmem xs)
                        where newmem = f mem x

scanl2SY _ _ NullS _ = NullS
scanl2SY _ _ _ NullS = NullS
scanl2SY f mem (x:-xs) (y:-ys) = f mem x y
                                :- (scanl2SY f newmem xs ys)
                                where newmem = f mem x y

scanl3SY _ _ NullS _ _ = NullS
scanl3SY _ _ _ NullS _ = NullS
scanl3SY _ _ _ _ NullS = NullS
scanl3SY f mem (x:-xs) (y:-ys) (z:-zs)
      = f mem x y z :- (scanl3SY f newmem xs ys zs)
      where newmem = f mem x y z

scanldSY _ _ NullS = NullS
scanldSY f mem (x:-xs) = mem :- (scanldSY f newmem xs)
                        where newmem = f mem x

scanld2SY _ _ NullS _ = NullS
scanld2SY _ _ _ NullS = NullS
scanld2SY f mem (x:-xs) (y:-ys) = mem :- (scanld2SY f newmem xs ys)
                        where newmem = f mem x y

scanld3SY _ _ NullS _ _ = NullS

```

```

scanld3SY _ _ _ NullS _ _ = NullS
scanld3SY _ _ _ _ NullS = NullS
scanld3SY f mem (x:-xs) (y:-ys) (z:-zs)
  = mem :- (scanld3SY 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

mooreSY nextState output initial
  = mapSY output . (scanldSY nextState initial)

moore2SY nextState output initial inp1 inp2 =
  mapSY output (scanld2SY nextState initial inp1 inp2)

moore3SY nextState output initial inp1 inp2 inp3 =
  mapSY output (scanld3SY nextState initial inp1 inp2 inp3)

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

mealy2SY nextState output initial inp1 inp2 =
  zipWith3SY output (scanld2SY nextState initial inp1 inp2)
  inp1 inp2

mealy3SY nextState output initial inp1 inp2 inp3 =
  zipWith4SY output (scanld3SY nextState initial inp1 inp2 inp3)
  inp1 inp2 inp3

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

sourceSY f s0 = o
  where
  o = delaySY s0 s
  s = mapSY f o

whenSY NullS _ = NullS
whenSY _ NullS = NullS
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) = x

```

```

holdSY a xs = scanlSY hold a xs
              where hold a Abst      = a
                  hold _ (Prst x) = x

zip3SY (x:-xs) (y:-ys) = (x, y) :- zipSY xs ys
zip3SY _ _ _          = NullS

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

zip4SY (w:-ws) (x:-xs) (y:-ys) (z:-zs) = (w, x, y, z)
                                           :- zip4SY ws xs ys zs
zip4SY _ _ _ _ _                    = NullS

unzipSY NullS = (NullS, NullS)
unzipSY ((x, y):-xys) = (x:-xs, y:-ys) where (xs, ys) = unzipSY xys

unzip3SY NullS = (NullS, NullS, NullS)
unzip3SY ((x, y, z):-xyzs) = (x:-xs, y:-ys, z:-zs) where
                               (xs, ys, zs) = unzip3SY xyzs

unzip4SY NullS = (NullS, NullS, NullS, NullS)
unzip4SY ((w,x,y,z):-wxyzs) = (w:-ws, x:-xs, y:-ys, z:-zs) where
                               (ws, xs, ys, zs) = unzip4SY wxyzs

zipxSY NullV = NullS
zipxSY (NullS :-> xss) = zipxSY xss
zipxSY ((x:-xs) :-> xss) = (x :-> (mapV headS xss))
                          :- (zipxSY (xs :-> (mapV tailS xss)))

unzipxSY NullS = NullV
unzipxSY (NullV :- vss) = unzipxSY vss
unzipxSY ((v:>vs) :- vss) = (v :- (mapSY headV vss))
                          :- (unzipxSY (vs :- (mapSY tailV vss)))

fstSY = mapSY fst

sndSY = mapSY snd

```

## 4.2 The Module `DomainInterfaces`

### 4.2.1 Overview

The module `DomainInterfaces` defines domain interface constructors for the multi-rate computational model.

```

module DomainInterfaces(downDI, upDI, par2serxDI, ser2parxDI,
                        par2ser2DI, par2ser3DI, par2ser4DI,
                        ser2par2DI, ser2par3DI, ser2par4DI) where

```

```

import Signal
import Vector
import SynchronousLib

```

## 4.2.2 Domain Interface Constructors

The domain interface constructors `downDI` and `upDI` take a parameter  $k$  and down- and up-sample an input signal.

The domain interface constructors `par2ser2DI`, `par2ser3DI` and `par2ser4DI` implement the domain interface constructor  $p2sDI(m)$  for  $m = 2, 3, 4$ . The domain interface constructors `par2serxDI` implements the domain interface constructor  $p2sDI(m)$  for a variable  $m$ .

The domain interface constructors `ser2par2DI`, `ser2par3DI` and `ser2par4DI` implement the domain interface constructor  $s2pDI(m)$  for  $m = 2, 3, 4$ . The domain interface constructors `ser2parxDI` implements the domain interface constructor  $s2pDI(m)$  for a variable  $m$ .

```

downDI    :: Num a => a -> Signal b -> Signal b
upDI      :: Num a => a -> Signal b -> Signal (AbstExt b)
par2serxDI :: Vector (Signal a) -> Signal a
ser2parxDI :: (Num a, Ord a) => a -> Signal (AbstExt b)
          -> Vector (Signal (AbstExt b))
par2ser2DI :: Signal a -> Signal a -> Signal a
par2ser3DI :: Signal a -> Signal a -> Signal a -> Signal a
par2ser4DI :: Signal a -> Signal a -> Signal a -> Signal a
          -> Signal a
ser2par2DI :: Signal a -> Signal (AbstExt a, AbstExt a)
ser2par3DI :: Signal a -> Signal (AbstExt a, AbstExt a, AbstExt a)
ser2par4DI :: Signal a
          -> Signal (AbstExt a, AbstExt a, AbstExt a, AbstExt a)

```

## 4.2.3 Implementation

```

downDI n xs    = down1 n 1 xs
  where down1 n m NullS = NullS
        down1 1 1 (x:-xs) = x :- down1 1 1 xs
        down1 n 1 (x:-xs) = x :- down1 n 2 xs
        down1 n m (x:-xs) = if m == n then
                              down1 n 1 xs
                            else
                              down1 n (m+1) xs

upDI n NullS    = NullS
upDI n (x:-xs) = (Prst x) :- ((copyS (n-1) Abst) ++ upDI n xs)

par2ser2DI xs ys = par2ser2DI' (zipSY xs ys)
  where par2ser2DI' NullS = NullS
        par2ser2DI' ((x,y):-xys) = x:-y:-par2ser2DI' xys

```

```

par2ser3DI xs ys zs = par2ser3DI' (zip3SY xs ys zs)
  where par2ser3DI' NullS = NullS
        par2ser3DI' ((x,y,z):-xyzs) = x:- y :-z :- par2ser3DI' xyzs

par2ser4DI ws xs ys zs = par2ser4DI' (zip4SY ws xs ys zs)
  where par2ser4DI' NullS = NullS
        par2ser4DI' ((w,x,y,z):-wxyzs)
          = w:-x:-y:-z:- par2ser4DI' wxyzs

ser2par2DI = group2SY . delaynSY Abst 2 . mapSY abstExt
ser2par3DI = group3SY . delaynSY Abst 3 . mapSY abstExt
ser2par4DI = group4SY . delaynSY Abst 4 . mapSY abstExt

par2serxDI = par2serxDI' . zipxSY
  where par2serxDI' NullS = NullS
        par2serxDI' (xv:-xs) = (signal . fromVector) xv
                               +-+ par2serxDI' xs

ser2parxDI n = unzipxSY . delaySY (copyV n Abst)
              . filterAbstDI . group n

group2SY NullS = NullS
group2SY (x:-NullS) = NullS
group2SY (x:-y:-xys) = (x, y) :- group2SY xys

group3SY NullS = NullS
group3SY (x:-NullS) = NullS
group3SY (x:-y:-NullS) = NullS
group3SY (x:-y:-z:-xyzs) = (x, y, x) :- group3SY xyzs

group4SY NullS = NullS
group4SY (w:-NullS) = NullS
group4SY (w:-x:-NullS) = NullS
group4SY (w:-x:-y:-NullS) = NullS
group4SY (w:-x:-y:-z:-wxyzs) = (w, x, y, z) :- group4SY wxyzs

filterAbstDI :: Signal (AbstExt a) -> Signal a
filterAbstDI NullS = NullS
filterAbstDI (Abst:-xs) = filterAbstDI xs
filterAbstDI ((Prst x):-xs) = x :- filterAbstDI xs

group n xs = mapSY (output n) (scan1SY (addElement 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

```

## 5 Application Libraries

### 5.1 The Module `SynchronousProcessLib`

#### 5.1.1 Overview

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

```

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

import SynchronousLib
import Signal
import AbsentExt
import Queue
import Memory

```

#### 5.1.2 Processes

The library defines the following processes:

```

fifoDelaySY      :: Signal [a] -> Signal (AbstExt a)
finiteFifoDelaySY :: Int -> Signal [a] -> Signal (AbstExt a)
memorySY         :: Int -> Signal (Access a) -> Signal (AbstExt a)
mergeSY          :: Signal (AbstExt a) -> Signal (AbstExt a)
                 -> Signal (AbstExt 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 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 process `counter` implements a counter, that counts from `min` to `max`. The process `counterS` has no input and its output is an infinite signal.

### 5.1.3 Implementation of Processes

```

fifoDelaySY xs           = mooreSY fifoState fifoOutput (queue []) xs

fifoState                :: Queue a -> [a] -> Queue a
fifoState (Q []) xs     = (Q xs)
fifoState q xs          = fst (popQ (pushListQ q xs))

fifoOutput               :: Queue a -> AbstExt a
fifoOutput (Q [])       = Abst
fifoOutput (Q (x:xs))   = Prst x

finiteFifoDelaySY n xs
  = mooreSY fifoStateFQ fifoOutputFQ (finiteQueue n []) xs

fifoStateFQ :: FiniteQueue a -> [a] -> FiniteQueue a
fifoStateFQ (FQ n []) xs = (FQ n xs)
fifoStateFQ q xs         = fst (popFQ (pushListFQ q xs))

fifoOutputFQ :: FiniteQueue a -> AbstExt a
fifoOutputFQ (FQ n []) = Abst
fifoOutputFQ (FQ n (x:xs)) = Prst x

memorySY size xs        = mealySY ns o (newMem size) xs
  where
    ns mem (Read x)     = memState mem (Read x)
    ns mem (Write x v) = memState mem (Write x v)
    o mem (Read x)     = memOutput mem (Read x)
    o mem (Write x v) = memOutput mem (Write x v)

mergeSY xs ys           = moore2SY mergeState mergeOutput [] xs ys
  where
    mergeState []       Abst Abst = []
    mergeState []       Abst (Prst y) = [y]
    mergeState []       (Prst x) Abst = [x]
    mergeState []       (Prst x) (Prst y) = [x, y]
    mergeState (u:us)   Abst Abst = us
    mergeState (u:us)   Abst (Prst y) = us ++ [y]
    mergeState (u:us)   (Prst x) Abst = us ++ [x]
    mergeState (u:us)   (Prst x) (Prst y) = us ++ [x, y]

    mergeOutput []      = Abst
    mergeOutput (u:us) = Prst u

groupSY k = mooreSY f g s0
  where
    s0 = NullV
    f v x | lengthV v == 0 = unitV x
          | lengthV v == k = unitV x

```

```

| otherwise = v <: x
g v | lengthV v == 0 = Prst NullV
g v | lengthV v == k = Prst v
g v | otherwise = Abst

```

```
counterSY m n = sourceSY f m
```

```
where
```

```

f x | x >= n = m
    | otherwise = succ x

```

## 5.2 The Module FIR

A FIR-filter is described by the following equation, which is illustrated in Figure 4:

$$y_n = \sum_{m=0}^k x_{n-m} h_m \quad (4)$$

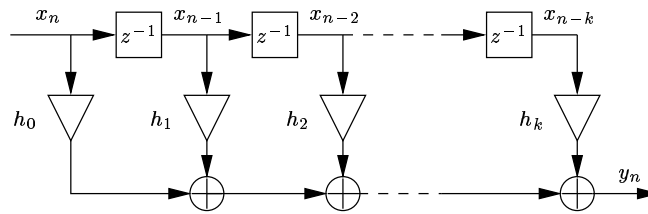


Figure 4: FIR-filter

The state of the FIR-Filter can be seen as a shift register with parallel output. The first element in the shift register at cycle  $n$  is  $x_n$  and the last element is  $x_{n-k}$ . In the next cycle a new value  $x_{n+1}$  is shifted into the register from the left, all other elements are shifted one place to the right, and the value  $x_{n-k}$  is discarded. We model the shift register with the process  $shiftreg_k$ . The process is based on the process constructor  $scanlSY$  which takes the shift function  $shiftV$  as first argument and an initial vector of size  $k + 1$  with zeroes as initial values.

The output of the shiftregister, a signal of vectors, is transformed with the process  $unzipxSY$  into a vector of signals. Then the process  $innerProd$  calculates the inner product of the coefficient vector  $h$  and the output of the shift register. The process is implemented by the process constructor  $zipWithSY$  that takes a parametrized function  $ipV(h)$  as arguments.

```
module FIR (fir) where
```

```
import SynchronousLib
```

```
fir h = innerProd h . sipo k 0.0
```

```
where k = lengthV h
```

```
sipo n s0 = unzipxSY . scanlSY shiftV initState
```

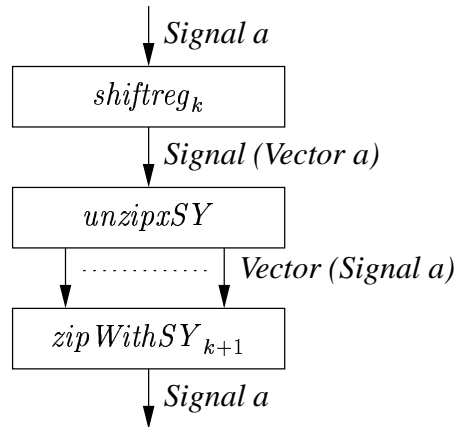


Figure 5: FIR-filter model

```

where initState = copyV n s0

innerProd h = zipWithxSY (ipV h)
  where ipV NullV NullV = 0
        ipV (h:>hv) (x:>xv) = h*x + ipV hv xv

```

All kinds of FIR-filters can now be modeled by means of *fir*. The only argument needed is the list of coefficients, which is given as a vector of any size. To illustrate this, an 8-th order band pass filter is modeled as follows.

```

bp = fir (vector [
  0.06318761339784, 0.08131651217682,
  0.09562326700432, 0.10478344432968,
  0.10793629404886, 0.10478344432968,
  0.09562326700432, 0.08131651217682,
  0.06318761339784 ])

```

## References

- [1] The ForSyDe webpage. <http://www.ele.kth.se/ForSyDe>.
- [2] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.
- [3] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing*. Prentice Hall, 3 edition, 1996.
- [4] H. J. Reekie. *Realtime Signal Processing*. PhD thesis, University of Technology at Sydney, Australia, 1995.

- [5] I. Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.