SAVE Project Report, Dept. of Computer and Information
Science, Linköping University, Linköping, April 2001.

From Haskell to PRES+
Basic Translation Procedures

Luis Alejandro Cortés, Petru Eles, and Zebo Peng
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden

Abstract

We define in this report some basic procedures to translate Haskell descriptions (based on a library of Skeletons) into PRES+ models. In this way, a system initially described in Haskell, may be transformed into a representation that might be formally verified. Thus the representation of the system is verified using formal methods by model-checking the model against a set of required properties expressed by temporal logics. This work has been done in the frame of the SAVE project, which aims to study the specification and verification of heterogeneous electronic systems.

1. Introduction

The SAVE Project is a joint research work by ESDlab at the Royal Institute of Technology (KTH), Stockholm, and ESLAB at Linköping University (LiU), Linköping, with financial support from NUTEK and in cooperation with Saab Bofors Dynamics. The objective of the project is to devise improved solutions and methods for high level specification, verification, and refinement of electronic systems by use of formal methods [SAV99].

In the frame of SAVE, the design flow starts with a Haskell description of the system and, after some transformations at this level and initial system validation by means of simulation, the Haskell description is translated into the PRES+ formalism. This allows the representation of the system to be verified using formal methods by model-checking the model against a set of required properties expressed by temporal logics [Cor00]. The kind of verification performed at this stage refers to check safety (no dangerous states are ever reached), liveness (absence of deadlocks, so that the functionality may eventually be completed), and timing properties. This formal verification approach does NOT deal with the functional correctness of the system in terms of the expected output values. From the result of the model checking procedure there could be feedback to the Haskell description. Also the PRES+ model can be simulated by using the tool SimPRES in order to study (validate) the functionality of the system in terms of correct output values. In the next step of the design flow, architecture decisions are taken and these must be reflected in new details of the system, that is, the mapped PRES+ model. Also, at this point both formal verification and validation by simulation can be performed. In case that the system does not fulfill its required properties or does not have the expected behavior, it is possible to go back to previous phases of the design process. The SAVE design flow is illustrated in Figure 1.

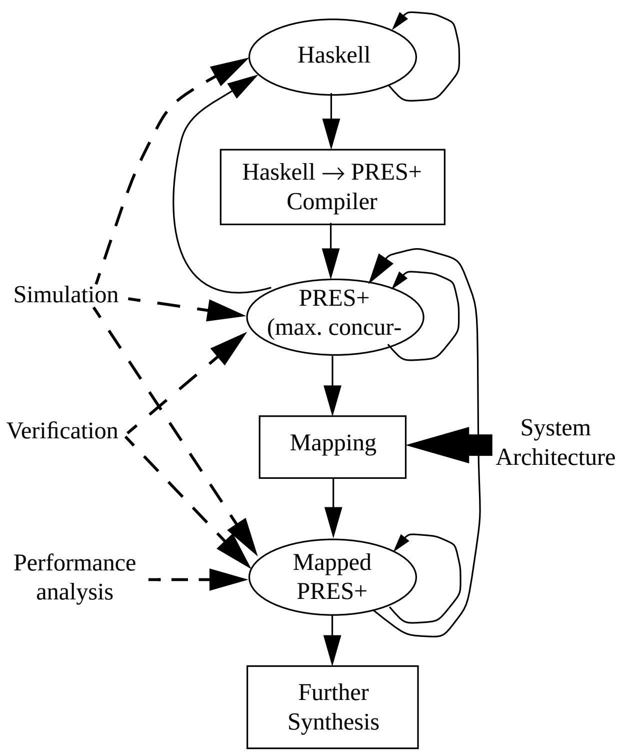


Figure 1. SAVE design flow

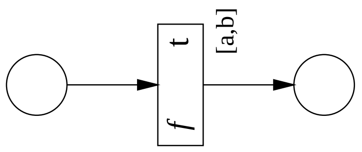
2. Haskell → PRES+ Translation

In this report, we concentrate on the Haskell → PRES+ translation by defining the basic translation procedures. In the SAVE design flow, the system is initially described in Haskell (a purely functional language) using *skeletons*. Skeletons are higher-order function which are used to model elementary processes. A skeleton takes elementary functions and signals as input parameters and produces signals as output. We define a set of basic PRES+ structures corresponding to Haskell skeletons as a first step in the translation procedure. The library of skeletons is appended at the end of this report.

We start with the skeleton `mapS` which applies a function *f* to an input signal. This skeleton may be easily mapped into a PRES+ structure consisting of a single transition with one input place (corresponding to the input signal) and one output place. The function *f* is captured as transition function of the transition in the PRES+ structure as shown in Figure 2. Additionally, the lower and upper bounds for the execution time of *f* may be expressed as minimum and maximum transition delays `a` and `b`.

```
mapS :: (a -> b) -> Signal a -> Signal b
mapS f NullS   = NullS
mapS f (x:-xs) = f x :- (mapS f xs)
```

(a)

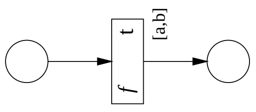
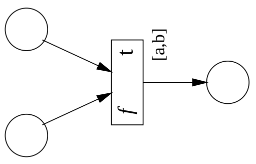
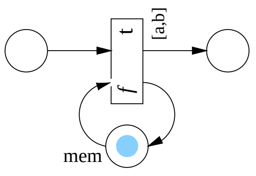


(b)

Figure 2. a) Skeleton `mapS`; b) Corresponding PRES+ structure

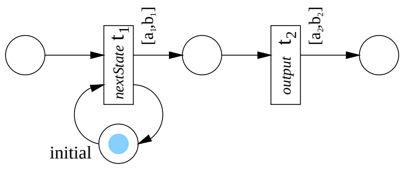
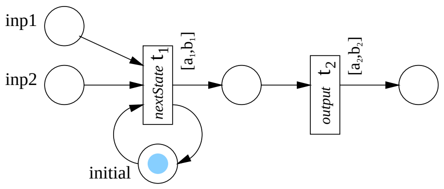
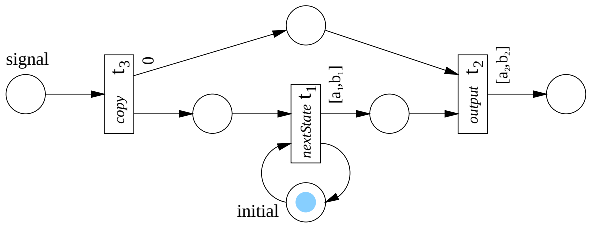
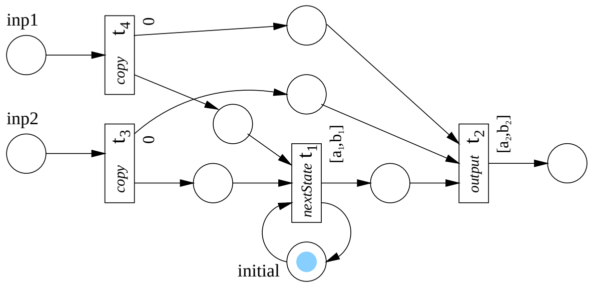
The skeleton `zipWithS`, which applies a two-argument function *f* to two input signals, is mapped into a PRES+ structure consisting of a single transition with two input places and one output place. *f* is captured as transition function. Table 1 shows the PRES+ structures corresponding to elemental atomic skeletons. Others, like `zipWith3S`, `zipWith4S`, `scanl2S`, and `scanl3S`, can be easily derived from the ones presented in Table 1.

Table 1: Translation of Atomic Skeletons

Skeleton	PRES+ Structure
`mapS`	
`zipWithS`	
`scanlS`	

Skeletons that are composed from atomic ones are illustrated in Table 2, as well as their corresponding PRES+ structure. Note that for `mealyS` and `mealy2S` we introduce transitions with transition function *copy* (given by $copy(x) = x$) and transition delay `0`. These are used to duplicate a token representing a signal that will be used by two different transitions.

Table 2: Translation of Skeletons

Skeleton	PRES+ Structure
`mooreS`	
`moore2S`	 inp1 inp2 nextState t1 [a1,b1] output t2 [a2,b2] initial
`mealyS`	 signal copy t3 0 nextState t1 [a1,b1] output t2 [a2,b2] initial
`mealy2S`	 inp1 copy t4 0 inp2 copy t3 0 nextState t1 [a1,b1] output t2 [a2,b2] initial

3. Example

We illustrate the basic Haskell → PRES+ translation procedures by means of a simple example, the subsystem *Audio Filter* of the *Equalizer* presented in [San01]. The audio filter has been described in Haskell as shown by the following code.

```
audioFilter bass treble audioIn = audioOut
   where audioOut = zipWith3S add3 bassPath middlePath treblePath
```

```
      bassPath = product (mapS exp bass) (lp audioIn)
      middlePath = scale 1.0 (bp audioIn)
      treblePath = product (mapS exp treble) (hp audioIn)
      scale k xs = mapS (* k) xs
      product xs ys = zipWithS (*) xs ys
      add3 a b c = a + b + c

lp = fir (vector [0.07749571497126, 0.09623416925141, 0.11114054570064,
   0.12073409187670, 0.12404450149000, 0.12073409187670, 0.11114054570064,
   0.09623416925141, 0.07749571497126])

bp = fir (vector [0.15287650949706, -0.00000000000000, -0.22846192040287,
  -0.00000000000000, 0.25797410154898, -0.00000000000000, -0.22846192040287,
  -0.00000000000000, 0.15287650949706])

hp = fir(vector [0.07749571497126, -0.09623416925141, 0.11114054570064,
  -0.12073409187670, 0.12404450149000, -0.12073409187670, 0.11114054570064,
  -0.09623416925141, 0.07749571497126])
```

By observing the atomic skeletons `zipWith3S`, `mapS`, and `zipWithS` in the description above, it is not difficult to obtain the PRES+ representation of the subsystem *Audio Filter* as shown in Figure 3. To implement filter activities a parametric function `fir` was used in the code above. Such a function can be mapped into a transition whose transition function is given by the Haskell code of `fir`. Another possibility is to make use of the hierarchy in PRES+ and set up super-transitions corresponding to the three instances of `fir` used in the description. We opted for the latter in order to illustrate how the concept of hierarchy is handled in PRES+ [Cor01] (though the first alternative yields also a correct model).

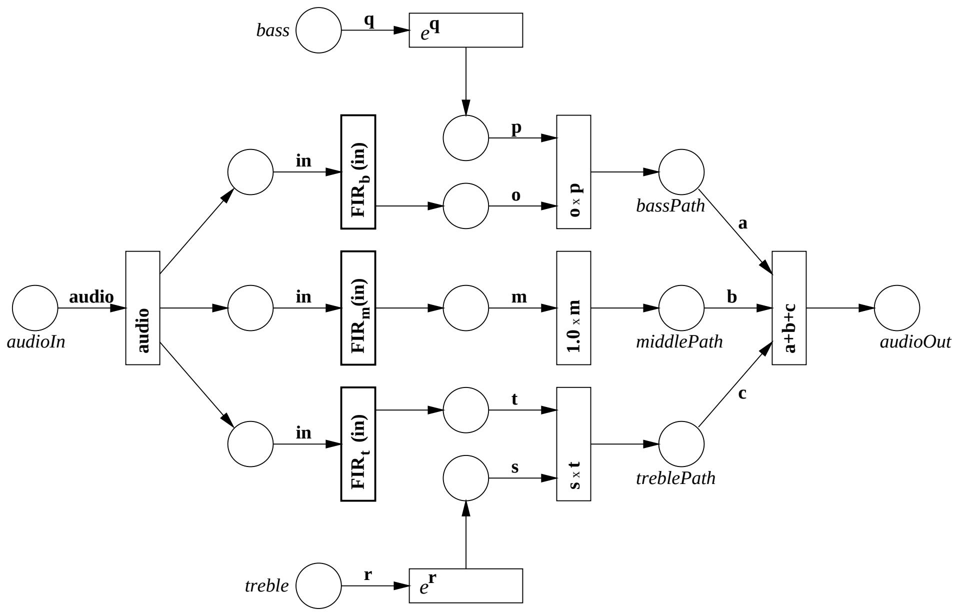


Figure 3. PRES+ model of the *Audio Filter*

In Figure 3, thick-line boxes represent super-transitions, each one of which is “refined” by a subnet that implements a FIR-filter. Recall that the behavior of FIR-filter is described by the equation

$$y_n = \sum_{i=0}^{k} x_{n-i} h_i$$

where x_n and y_n are the nth input and output signals respectively, and h_i, $0 \le i \le k$, are the coefficients of a k-order filter. Such a filter is modeled in PRES+ as illustrated in Figure 4. Note that the super-transitions *FIR*$_b$*(in), FIR*$_m$*(in),* and *FIR*$_t$*(in)* in Figure 3 are different since in each of them the filter coefficients vary, though the structure is the same. We have not given explicitly transition/super-transition delays, nonetheless lower and upper time limits of activities can be expressed in PRES+ in an easy manner.

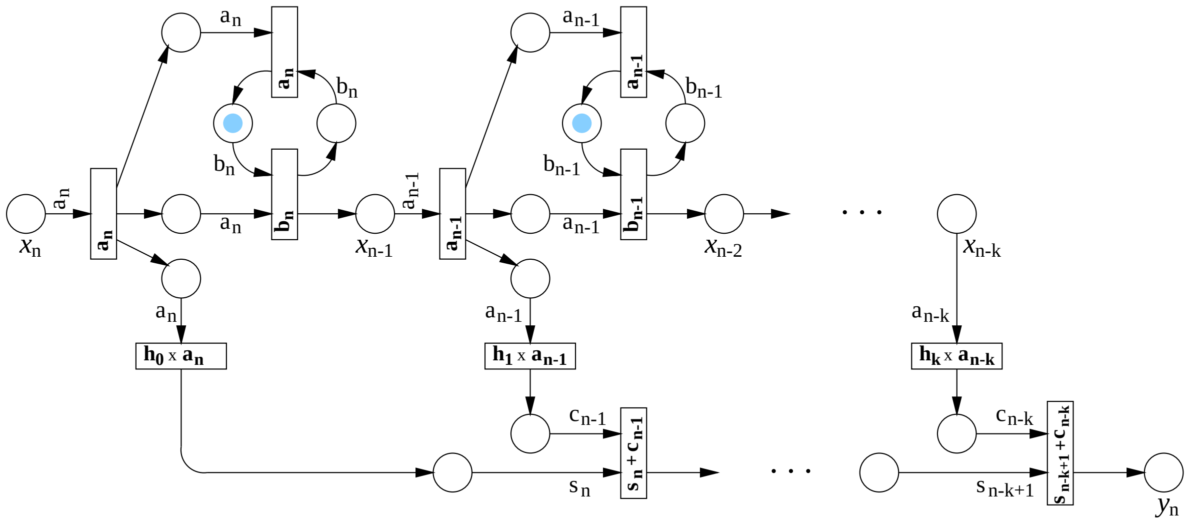


Figure 4. FIR-filter

References

[Cor00] L. A. Cortés, P. Eles, and Z. Peng, “Verification of Heterogeneous Electronic Systems using Model Checking,” SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, July 2000.

[Cor01] L. A. Cortés, P. Eles, and Z. Peng, “Hierarchies for the Modeling and Verification of Embedded Systems,” SAVE Project Report, Dept. of Computer and Information Science, Linköping University, Linköping, February 2001.

[SAV99] “SAVE: Specification and Verification of Heterogeneous Electronic Systems,” Project Plan, March 1999.

[San01] I. Sander, “System Model of an Equalizer,” March 2001.

[Wu00] W. Wu and A. Jantsch, “A System Design Methodology Based on a Formal Computational Model,” SAVE Project Report, Dept. of Electronics, Royal Institute of Technology, Stockholm, January 2000.

Appendix A. Library of Skeletons (by Ingo Sander)

A.1. Atomic Skeletons

```
{-
   Module:      AtomicSkeletons

   Filename:    AtomicSkeleton.hs

   Description: This module provides the atomicSkeletons, that are defined
                inside the ForSyDe-methodology.
                mapS
                zipWithS
                scanlS
                delayS
                whenT

   Revisions:

   Date    Version Author       Changes

   19/6-00   0.1   Ingo Sander  created
   21/8-00   0.2   Ingo Sander  Redefinition of datatype Signal
   18/9-00   0.21  Ingo Sander  Introduction of zipWith3S, zipWith4S,
                                scanl2S, scanl3S
   28/9-00   0.22  Ingo Sander  uses new internal representation of Signal
   6/10-00   0.23  Ingo Sander  zipS, unzipS, groupS, concatS introduced
-}

module AtomicSkeletons(mapS, zipWithS, zipWith3S, zipWith4S, scanlS,
                 scanl2S, scanl3S, delayS, whenT, fillT, holdT,
                 zipS, unzipS, groupS, concatS) where

import DataTypes
import Vector

---------------
-- SKELETONS --
---------------

{-
   We use the following convention for Skeletons:
     - Skeletons, which can be used with all Signaltypes are denoted
       by "nameS"
     - Skeletons, which can only be used with timed signals are dentod
       by "nameT"
-}

{-
   The skeleton mapS applies a funktion f on the values of all events
   in a signal.
-}

mapS :: (a -> b) -> Signal a -> Signal b
mapS f NullS   = NullS
mapS f (x:-xs) = f x :- (mapS f xs)


{-
   The skeleton ’zipWithS’ applies a ’two-operand-function’ on elementwise
   on two input signals.
-}

zipWithS :: (a -> b -> c) -> Signal a -> Signal b -> Signal c
zipWithS f NullS   _       = NullS
zipWithS f _       NullS   = NullS
zipWithS f (x:-xs) (y:-ys) = f x y :- (zipWithS f xs ys)

{-
   The skeleton 'zipWith3S' applies a 'three-operand-function' on elementwise
   on three input signals.
-}

zipWith3S :: (a -> b -> c -> d) -> Signal a -> Signal b
          -> Signal c -> Signal d
zipWith3S f NullS   _       _       = NullS
zipWith3S f _       NullS   _       = NullS
zipWith3S f _       _       NullS   = NullS
zipWith3S f (x:-xs) (y:-ys) (z:-zs) = f x y z :- (zipWith3S f xs ys zs)

{-
   The skeleton 'zipWith4S' applies a 'four-operand-function' on elementwise
   on four input signals.
-}

zipWith4S :: (a -> b -> c -> d -> e) -> Signal a -> Signal b
          -> Signal c -> Signal d -> Signal e
zipWith4S f NullS   _       _       _      = NullS
zipWith4S f _       NullS   _       _      = NullS
zipWith4S f _       _       NullS   _      = NullS
zipWith4S f _       _       _       NullS  = NullS
zipWith4S f (w:-ws) (x:-xs) (y:-ys) (z:-zs)
       = f w x y z :- (zipWith4S f ws xs ys zs)

{-
   The skeleton scanlS applies a function f on the head event of the signal
   and a inner state mem. The result of this function call server as
   output event and as new state mem.
-}

scanlS :: (a -> b -> a) -> a -> Signal b -> Signal a
scanlS f mem NullS   = NullS
scanlS f mem (x:-xs) = f mem x :- (scanlS f newmem xs)
                       where newmem = f mem x

{-
   The skeleton scanlS applies a function f on the head events of
   two input signals and an inner state mem. The result of this function
   call serves as output event and as new state mem.
-}
scanl2S :: (a -> b -> c -> a) -> a -> Signal b -> Signal c -> Signal a
scanl2S f mem NullS   _       = NullS
scanl2S f mem _       NullS   = NullS
scanl2S f mem (x:-xs) (y:-ys) = f mem x y :- (scanl2S f newmem xs ys)
                                where newmem = f mem x y

{-
   The skeleton scanlS applies a function f on the head events of
   three input signals and an inner state mem. The result of this function
   call serves as output event and as new state mem.
-}
scanl3S :: (a -> b -> c -> d -> a) -> a -> Signal b
        -> Signal c -> Signal d -> Signal a
```

```
scanl3S f mem NullS   _       _       = NullS
scanl3S f mem _       NullS   _       = NullS
scanl3S f mem _       _       NullS   = NullS
scanl3S f mem (x:-xs) (y:-ys) (z:-zs)
        = f mem x y z :- (scanl3S f newmem xs ys zs)
          where newmem = f mem x y z

{-
   The skeleton ’delayS’ delays the output one event cycle by inserting an
   event e
-}

delayS :: a -> Signal a -> Signal a
delayS e es = e:-es


{-
   The skeleton ’whenT’ synchronizes a signal with another signal.
   The first skeleton gets the value Absent when the second signal
   has the value Absent. Otherwise it keeps its value.
-}

whenT :: TimedSignal a -> TimedSignal b -> TimedSignal a
whenT NullS   _            = NullS
whenT _       NullS        = NullS
whenT (x:-xs) (Absent:-ys) = Absent :- (whenT xs ys)
whenT (x:-xs) (y:-ys)      = x      :- (whenT xs ys)



fillT :: TimedValue a -> TimedSignal a -> TimedSignal a
fillT a xs = mapS (replaceAbsent a) xs
             where replaceAbsent a Absent = a
                   replaceAbsent a x      = x

holdT :: TimedValue a -> TimedSignal a -> TimedSignal a
holdT a xs = scanlS hold a xs
             where hold a Absent      = a
                   hold a (Present x) = Present x


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

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

groupS n NullS = NullS
groupS 0 _     = NullS
groupS n xs
       | nullS xs  = NullS
       | otherwise = v :- groupS n (dropS n xs)
                     where v = takeSV n xs

concatS NullS   = NullS
concatS (v:-vs) = appendVS v (concatS vs)

-- Help Functions

appendVS NullV   s = s
appendVS (x:>xs) s = x :- appendVS xs s

takeSV k = tk k NullV
```

```
           where tk 0 v s       = v
                 tk k v NullS   = NullV
                 tk k v (x:-xs) = tk (k-1) (v<:x) xs
```

A.2. Skeletons

```
{-
   Module:      Skeletons

   Filename:    Skeletons.hs

   Description: This module provides skeletons, that are composed from
                atomic skeletons.

   Revisions:

   Date    Version Author       Changes

   19/6-00   0.1   Ingo Sander  created
   21/8-00   0.2   Ingo Sander  Datatype Signal redefined
   18/9-00   0.21  Ingo Sander  moore2S, mealy2S introduced
                                Redifintion of mealyS-skeletons
   28/9-00   0.22  Ingo Sander  uses new internal representation of Signal
-}

module Skeletons(mooreS, moore2S, moore3S, mealyS, mealy2S, mealy3S,
              partitionT) where

import ForSyDeCore

{-
   The skeleton 'partitionT' partitions a signal into two signals depending
   on a predicate function p
-}

partitionT ::    (TimedValue a -> Bool) -> TimedSignal a
              -> (TimedSignal a, TimedSignal a)
partitionT p NullS   = (NullS, NullS)
partitionT p (x:-xs) = (xTrue (x:-xs), xFalse (x:-xs))
                    where xTrue NullS   = NullS
                          xTrue (x:-xs) = if p x then
                                x :- (xTrue xs)
                          else
                                Absent :- (xTrue xs)
                          xFalse NullS  = NullS
                          xFalse (x:-xs)= if p x then
                                Absent :- (xFalse xs)
                          else
                                x :- (xFalse xs)



------------------------
-- COMPOSED SKELETONS --
------------------------

mooreS :: (a -> b -> a) -> (a -> c) -> a -> Signal b -> Signal c
mooreS nextState output initial = mapS output . scanlS nextState initial

moore2S :: (a -> b -> c -> a) -> (a -> d) -> a -> Signal b
```

```
        -> Signal c -> Signal d
moore2S nextState output initial inp1 inp2 =
     mapS output (scanl2S nextState initial inp1 inp2)

moore3S :: (a -> b -> c -> d -> a) -> (a -> e) -> a -> Signal b
        -> Signal c -> Signal d -> Signal e
moore3S nextState output initial inp1 inp2 inp3 =
     mapS output (scanl3S nextState initial inp1 inp2 inp3)

mealyS :: (a -> b -> a) -> (a -> b -> c) -> a -> Signal b -> Signal c
mealyS nextState output initial signal =
       zipWithS output (scanlS nextState initial signal) signal

mealy2S :: (a -> b -> c -> a) -> (a -> b -> c -> d) -> a
        -> Signal b -> Signal c -> Signal d
mealy2S nextState output initial inp1 inp2 =
     zipWith3S output (scanl2S nextState initial inp1 inp2) inp1 inp2

mealy3S :: (a -> b -> c -> d -> a) -> (a -> b -> c -> d -> e) -> a
        -> Signal b -> Signal c -> Signal d -> Signal e
mealy3S nextState output initial inp1 inp2 inp3 =
     zipWith4S output (scanl3S nextState initial inp1 inp2 inp3)
            inp1 inp2 inp3
```