```haskell
{- $Id: AFRP.hs,v 1.14 2002/04/30 23:29:46 henrik Exp $
******************************************************************************
*                                 A F R P                                    *
*                                                                            *
*       Module:         AFRP                                                 *
*       Purpose:        The AFRP core.                                       *
*       Authors:        Antony Courtney and Henrik Nilsson                   *
*                                                                            *
******************************************************************************
-}

module AFRP (
-- Re-exported modules
    module Arrow,
    module AFRPTuple,   -- Utility functions and numeric instances for tuples.

-- Main types
    Time,       -- [s] Both for time w.r.t. some reference and intervals.
    SF,         -- Signal Function.
    Event,      -- Events; conceptually similar to Maybe (but abstract).

-- Main instances
    -- SF is an instance of Arrow and ArrowLoop. Method instances:
    -- arr      :: (a -> b) -> SF a b
    -- (>>>)    :: SF a b -> SF b c -> SF a c
    -- first    :: SF a b -> SF (a,c) (b,c)
    -- second   :: SF a b -> SF (c,a) (c,b)
    -- (***)    :: SF a b -> SF a' b' -> SF (a,a') (b,b')
    -- (&&&)    :: SF a b -> SF a b' -> SF a (b,b')
    -- returnA  :: SF a a
    -- loop     :: SF (a,c) (b,c) -> SF a b

    -- Event is an instance of Functor, Eq, and Ord. Some method instances:
    -- fmap     :: (a -> b) -> Event a -> Event b
    -- (==)     :: Event a -> Event a -> Bool
    -- (<=)     :: Event a -> Event a -> Bool

-- Basic signal functions
    identity,           -- :: SF a a
    constant,           -- :: b -> SF a b
    localTime,          -- :: SF a Time
    time,               -- :: SF a Time

-- Initialization
    initially,          -- :: a -> SF a a

-- Basic event sources
    never,              -- :: SF a (Event b)
    now,                -- :: b -> SF a (Event b)
    snap,               -- :: SF a (Event a)
    after,              -- :: Time -> b -> SF a (Event b)
    repeatedly,         -- :: Time -> b -> SF a (Event b)
    edge,               -- :: SF Bool (Event ())
    edgeJust,           -- :: SF (Maybe a) (Event a)
    edgeBy,             -- :: (a -> a -> Maybe b) -> a -> SF a (Event b)

-- Stateful event suppression
    notYet,             -- :: SF (Event a) (Event a)

-- Collection-oriented combinators
    par,                -- :: Functor col => col (SF a b) -> SF a (col b)

-- Switchers
    switch,  dSwitch,   -- :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
    kSwitch, dkSwitch,  -- :: SF a b
```

```haskell
                        --    -> SF (a,b) (Event c)
                        --    -> (SF a b -> c -> SF a b)
                        --    -> SF a b
    rSwitch, drSwitch,  -- :: SF a b -> SF (a,Event (SF a b)) b
    pSwitch, dpSwitch,  -- :: Functor col =>
                        --        col (SF a b)
                        --        -> SF (a, col b) (Event c)
                        --        -> (col (SF a b) -> c -> SF a (col b))
                        --        -> SF a (col b)
    rpSwitch,drpSwitch, -- :: Functor col =>
                        --        col (SF a b)
                        --        -> SF (a, Event (col (SF a b)->col (SF a b)))
                        --              (col b)

-- Wave-form generation
    hold,               -- :: a -> SF (Event a) a
    dHold,              -- :: a -> SF (Event a) a
    trackAndHold,       -- :: a -> SF (Maybe a) a
    dTrackAndHold,      -- :: a -> SF (Maybe a) a

-- Accumulators
    accum,              -- :: a -> SF (Event (a -> a)) (Event a)
    accumBy,            -- :: (b -> a -> b) -> b -> SF (Event a) (Event b)
    accumFilter,        -- :: (c -> a -> (c, Maybe b)) -> c
                        --    -> SF (Event a) (Event b)
    count,              -- :: Integral b => SF (Event a) (Event b)

-- Delay
    delay,              -- :: a -> SF a a

-- Integral
    integral,           -- :: Fractional a => SF a a

-- Loops with guaranteed well-defined feedback
    loopDelay,          -- :: c -> SF (a,c) (b,c) -> SF a b
    loopIntegral,       -- :: Fractional c => SF (a,c) (b,c) -> SF a b

-- Pointwise functions on events
    event,              -- :: a -> (b -> a) -> Event b -> a
    fromEvent,          -- :: Event a -> a
    isEvent,            -- :: Event a -> Bool
    isNoEvent,          -- :: Event a -> Bool
    tag,                -- :: Event a -> b -> Event b
    lMerge,             -- :: Event a -> Event a -> Event a
    rMerge,             -- :: Event a -> Event a -> Event a
    merge,              -- :: Event a -> Event a -> Event a
    mergeBy,            -- :: (a -> a -> a) -> Event a -> Event a -> Event a
    mergeEvents,        -- :: [Event a] -> Event a
    joinE,              -- :: Event a -> Event b -> Event (a,b)
    filterE,            -- :: (a -> Bool) -> Event a -> Event a
    gateE,              -- :: Bool -> Event a -> Event a

-- Reactimation
    reactimate,         -- :: IO a
                        --    -> (Bool -> IO (DTime, Maybe a))
                        --    -> (Bool -> b -> IO Bool)
                        --    -> SF a b
                        --    -> IO ()

-- Embedding (tentative: will be revisited)
    DTime,              -- [s] Sampling interval, always > 0.
    embed,              -- :: SF a b -> (a, [(DTime, Maybe a)]) -> [b]
    deltaEncode,        -- :: Eq a => DTime -> [a] -> (a, [(DTime, Maybe a)])
    deltaEncodeBy       -- :: (a -> a -> Bool) -> DTime -> [a]
                        --    -> (a, [(DTime, Maybe a)])
```

```
) where

import Monad (unless)

import Arrow
import AFRPDiagnostics
import AFRPTuple
import AFRPEvent


------------------------------------------------------------------------------
-- Basic type definitions with associated utilities
------------------------------------------------------------------------------

-- The time type is really a bit boguous, since, as time passes, the minimal
-- interval between two consecutive floating-point-represented time points
-- increases. A better approach is probably to pick a reasonable resolution
-- and represent time and time intervals by Integer (giving the number of
-- "ticks").

-- Time is used both for time intervals (duration), and time w.r.t. some
-- agreed reference point in time. Conceptually, Time = R, i.e. time can be 0
-- or even negative.
type Time = Double      -- [s]


-- DTime is the time type for lengths of sample intervals. Conceptually,
-- DTime = R+ = { x in R | x > 0 }. Don't assume Time and DTime have the
-- same representation.

type DTime = Double     -- [s]


-- Representation of signal function in initial state.
-- (Naming: "TF" stands for Transition Function.)

data SF a b = SF {sfTF :: a -> Transition a b}


-- Representation of signal function in running state.
-- It would have been nice to have a constructor SFId representing (arr id):
--
--     SFId {sfTF' :: DTime -> a -> Transition a b}
--
-- But it seems as if we need dependent types as soon as we try to exploit
-- that constructor (note that the type above is too general!), and a
-- work-around based on keeping around an extra function as a "proof" that we
-- can do the required coersions, yields codde which is no more efficient
-- than using SFArr in the first place.
-- (Naming: "TIVar" stands for "time-input-variable".)

data SF' a b
    = SFConst {sfTF' :: DTime -> a -> Transition a b, sfCVal :: b}
    | SFArr   {sfTF' :: DTime -> a -> Transition a b, sfAFun :: a -> b}
    | SFTIVar {sfTF' :: DTime -> a -> Transition a b}


-- A transition is a pair of the next state (in the form of a signal
-- function) and the output at the present time step.

type Transition a b = (SF' a b, b)


-- "Smart" constructors. The corresponding "raw" constructors should not
-- be used directly for construction.

sfConst :: b -> SF' a b
sfConst b = sf
    where
        sf = SFConst {sfTF' = \_ _ -> (sf, b), sfCVal = b}


-- !!! We could/should use NeverEvent here!
sfNever :: SF' a (Event b)
sfNever = sfConst NoEvent


sfId :: SF' a a
sfId = sf
    where
        sf = SFArr {sfTF' = \_ a -> (sf, a), sfAFun = id}


sfArr :: (a -> b) -> SF' a b
sfArr f = sf
    where
        sf = SFArr {sfTF' = \_ a -> (sf, f a), sfAFun = f}


-- Freezes a "running" signal function, i.e., turns it into a continuation in
-- the form of a plain signal function.
freeze :: SF' a b -> DTime -> SF a b
freeze sf dt = SF {sfTF = (sfTF' sf) dt}


freezeCol :: Functor col => col (SF' a b) -> DTime -> col (SF a b)
freezeCol sfs dt = fmap (flip freeze dt) sfs


------------------------------------------------------------------------------
-- Arrow instance and implementation
------------------------------------------------------------------------------

instance Arrow SF where
    arr    = arrPrim
    (>>>)  = compPrim
    first  = firstPrim
    second = secondPrim


-- Lifting.
arrPrim :: (a -> b) -> SF a b
arrPrim f = SF {sfTF = \a -> (sfArr f, f a)}


-- Composition.
-- The definition exploits the following identities:
--     sf         >>> constant c = constant c
--     constant c >>> arr f      = constant (f c)
--     arr f      >>> arr g      = arr (g . f)
-- (It would have been nice to explit e.g. identity >>> sf = sf, but it would
-- seem that we need dependent types for that.)
compPrim :: SF a b -> SF b c -> SF a c
compPrim (SF {sfTF = tf10}) (SF {sfTF = tf20}) = SF {sfTF = tf0}
    where
        tf0 a0 = (cpAux sf1 sf2, c0)
            where
                (sf1, b0) = tf10 a0
                (sf2, c0) = tf20 b0
```

```
        cpAux sf1             sf2@(SFConst{}) = sfConst (sfCVal sf2)
        cpAux sf1@(SFConst{}) sf2@(SFArr {})  = sfConst (sfAFun sf2
                                                           (sfCVal sf1))
        cpAux sf1@(SFArr {})  sf2@(SFArr {})  = sfArr (sfAFun sf2 . sfAFun sf1)
        cpAux sf1             sf2             = SFTIVar {sfTF' = tf}
            where
                tf dt a = (cpAux sf1' sf2', c)
                    where
                        (sf1', b) = (sfTF' sf1) dt a
                        (sf2', c) = (sfTF' sf2) dt b


-- Widening.
-- The definition exploits the following identities:
--     first (constant b) = arr (\(_, c) -> (b, c))
--     (first (arr f))    = arr (\(a, c) -> (f a, c))
-- (It would have been nice to explit first identity = identity, but it would
-- seem that we need dependent types for that.)
firstPrim :: SF a b -> SF (a,c) (b,c)
firstPrim (SF {sfTF = tf10}) = SF {sfTF = tf0}
    where
        tf0 (a0, c0) = (fpAux sf1, (b0, c0))
            where
                (sf1, b0) = tf10 a0

        fpAux (SFConst {sfCVal = b}) = sfArr (\(_, c) -> (b, c))
        fpAux (SFArr   {sfAFun = f}) = sfArr (\(a, c) -> (f a, c))
        fpAux sf1                    = SFTIVar {sfTF' = tf}
            where
                tf dt (a, c) = (fpAux sf1', (b, c))
                    where
                        (sf1', b) = (sfTF' sf1) dt a


secondPrim :: SF a b -> SF (c,a) (c,b)
secondPrim (SF {sfTF = tf10}) = SF {sfTF = tf0}
    where
        tf0 (c0, a0) = (spAux sf1, (c0, b0))
            where
                (sf1, b0) = tf10 a0

        spAux (SFConst {sfCVal = b}) = sfArr (\(c, _) -> (c, b))
        spAux (SFArr   {sfAFun = f}) = sfArr (\(c, a) -> (c, f a))
        spAux sf1                    = SFTIVar {sfTF' = tf}
            where
                tf dt (c, a) = (spAux sf1', (c, b))
                    where
                        (sf1', b) = (sfTF' sf1) dt a



------------------------------------------------------------------------------
-- ArrowLoop instance and implementation
------------------------------------------------------------------------------

instance ArrowLoop SF where
    loop = loopPrim


loopPrim :: SF (a,c) (b,c) -> SF a b
loopPrim (SF {sfTF = tf10}) = SF {sfTF = tf0}
    where
        tf0 a0 = (loopAux sf1, b0)
            where
                (sf1, (b0, c0)) = tf10 (a0, c0)

        loopAux (SFConst {sfCVal = (b, _)}) = sfConst b
        loopAux sf1                         = SFTIVar {sfTF' = tf}
            where
                tf dt a = (loopAux sf1', b)
                    where
                        (sf1', (b, c)) = (sfTF' sf1) dt (a, c)


------------------------------------------------------------------------------
-- Basic signal functions
------------------------------------------------------------------------------

-- Identity: identity = arr id
identity :: SF a a
identity = SF {sfTF = \a -> (sfId, a)}


-- Identity: constant b = arr (const b)
constant :: b -> SF a b
constant b = SF {sfTF = \_ -> (sfConst b, b)}


-- Outputs the time passed since the signal function instance was started.
localTime :: SF a Time
localTime = constant 1.0 >>> integral


-- Alternative name for localTime.
time :: SF a Time
time = localTime


------------------------------------------------------------------------------
-- Initialization
------------------------------------------------------------------------------

-- The identity function, except at (local) time 0 where the initial value is
-- output instead.
initially :: a -> SF a a
initially a_init = SF {sfTF = \_ -> (sfId, a_init)}


------------------------------------------------------------------------------
-- Basic event sources
------------------------------------------------------------------------------

-- Event source which never occurs.
-- !!! We could/should use NeverEvent here.
never :: SF a (Event b)
never = SF {sfTF = \_ -> (sfNever, NoEvent)}


-- Event source with a single occurrence at time 0. The value of the event
-- is given by t he function argument.
now :: b -> SF a (Event b)
now b0 = SF {sfTF = \_ -> (sfNever, Event b0)}


-- Event source with a single occurrence at time 0. The value of the event
-- is obtained by sampling the input at that time.
snap :: SF a (Event a)
snap = SF {sfTF = \a0 -> (sfNever, Event a0)}
```

```haskell
-- Event source with a single occurrence at or as soon after (local) time t_ev
-- as possible.
after :: Time -> b -> SF a (Event b)
after t_ev x = SF {sfTF = tf0}
    where
        tf0 _ = if t_ev <= 0 then
                    (sfNever, Event x)
                else
                    (afterAux 0.0, NoEvent)

        afterAux t = SFTIVar {sfTF' = tf}
            where
                tf dt _ = let t' = t + dt
                          in
                              if t' >= t_ev then
                                  (sfNever, Event x)
                              else
                                  (afterAux t', NoEvent)


-- Event source with repeated occurrences with interval dt
-- Note: If the interval is too short w.r.t. the sampling intervals,
-- the result will be that events occur at every sample. However, no more
-- than one event results from any sampling interval, thus avoiding an
-- "event backlog" should sampling become more frequent at some later
-- point in time.
repeatedly :: Time -> b -> SF a (Event b)
repeatedly p_ev x | p_ev > 0 = SF {sfTF = tf0}
                  | otherwise = afrpErr "repeatedly" "Non-positive period."
    where
        tf0 _ = (repAux p_ev 0.0, NoEvent)

        repAux t_ev t = SFTIVar {sfTF' = tf}
            where
                tf dt _ = let t' = t + dt
                          in
                              if t' >= t_ev then
                                  (repAux (nextEventTime t' t_ev) t', Event x)
                              else
                                  (repAux t_ev t', NoEvent)
                nextEventTime t t_ev =
                    fromIntegral (truncate (t / p_ev) + 1) * p_ev


-- A rising edge detector. Useful for things like detecting key presses.
-- Note that we initialize the loop with state set to True so that there
-- will not be an occurence at t0 in the logical time frame in which
-- this is started.
edge :: SF Bool (Event ())
edge = edgeBy isEdge True
    where
        isEdge False False = Nothing
        isEdge False True  = Just ()
        isEdge True  True  = Nothing
        isEdge True  False = Nothing


-- Edge detector parameterized on the edge detection function and initial
-- state, i.e., the previous input sample. The first argument to the
-- edge detection function is the previous sample, the second the current one.
edgeBy :: (a -> a -> Maybe b) -> a -> SF a (Event b)
edgeBy isEdge a_init = SF {sfTF = tf0}
    where
        tf0 a0 = (ebAux a0, maybeToEvent (isEdge a_init a0))

        ebAux a_prev = SFTIVar {sfTF' = tf}
            where
                tf dt a = (ebAux a, maybeToEvent (isEdge a_prev a))


-- Detects an edge where a maybe signal is changing from nothing to something.
edgeJust :: SF (Maybe a) (Event a)
edgeJust = edgeBy isJustEdge (Just undefined)
    where
        isJustEdge Nothing  Nothing     = Nothing
        isJustEdge Nothing  ma@(Just _) = ma
        isJustEdge (Just _) (Just _)    = Nothing
        isJustEdge (Just _) Nothing     = Nothing


------------------------------------------------------------------------------
-- Stateful event suppression
------------------------------------------------------------------------------

-- Suppression of initial (at local time 0) event.
notYet :: SF (Event a) (Event a)
notYet = initially NoEvent


------------------------------------------------------------------------------
-- Collection-oriented combinators
------------------------------------------------------------------------------

-- Spatial parallel composition of a signal function collection.
par :: Functor col => col (SF a b) -> SF a (col b)
par sfs0 = SF {sfTF = tf0}
    where
        tf0 a0 =
            let sfbs0 = fmap (\sf0 -> (sfTF sf0) a0) sfs0
                sfs   = fmap fst sfbs0
                bs0   = fmap snd sfbs0
            in
                (parAux sfs, bs0)


-- Internal definition. Also used in parallel swithers.
parAux :: Functor col => col (SF' a b) -> SF' a (col b)
parAux sfs = SFTIVar {sfTF' = tf}
    where
        tf dt a =
            let sfbs' = fmap (\sf -> (sfTF' sf) dt a) sfs
                sfs'  = fmap fst sfbs'
                bs    = fmap snd sfbs'
            in
                (parAux sfs', bs)


------------------------------------------------------------------------------
-- Switchers
------------------------------------------------------------------------------

-- !!! We could detect a NeverEvent in the various switches.
-- !!! In which case we must ensure that SFConst ... NeverEvent ...
-- !!! is covered as well!

-- Basic switch.
switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
switch (SF {sfTF = tf10}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
```

```
            case tf10 a0 of
                (sf1, (b0, NoEvent))  -> (switchAux sf1, b0)
                (_,    (_,  Event c0)) -> sfTF (k c0) a0

        switchAux (SFConst {sfCVal = (b, NoEvent)}) = sfConst b
        switchAux sf1                               = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    case (sfTF' sf1) dt a of
                        (sf1', (b, NoEvent)) -> (switchAux sf1', b)
                        (_,     (_, Event c)) -> sfTF (k c) a


-- Switch with delayed observation.
dSwitch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
dSwitch (SF {sfTF = tf10}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
            let (sf1, (b0, ec0)) = tf10 a0
            in (case ec0 of
                    NoEvent  -> dSwitchAux sf1
                    Event c0 -> fst (sfTF (k c0) a0),
                b0)

        dSwitchAux (SFConst {sfCVal = (b, NoEvent)}) = sfConst b
        dSwitchAux sf1                               = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    let (sf1', (b, ec)) = (sfTF' sf1) dt a
                    in (case ec of
                            NoEvent -> dSwitchAux sf1'
                            Event c -> fst (sfTF (k c) a),

                        b)


-- "Call-with-current-continuation" switch.
kSwitch :: SF a b -> SF (a,b) (Event c) -> (SF a b -> c -> SF a b) -> SF a b
kSwitch sf10@(SF {sfTF = tf10}) (SF {sfTF = tfe0}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
            let (sf1, b0) = tf10 a0
            in
                case tfe0 (a0, b0) of
                    (sfe, NoEvent)  -> (kSwitchAux sf1 sfe, b0)
                    (_,   Event c0) -> sfTF (k sf10 c0) a0

        kSwitchAux sf1 (SFConst {sfCVal = NoEvent}) = sf1
        kSwitchAux sf1 sfe                          = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    let (sf1', b) = (sfTF' sf1) dt a
                    in
                        case (sfTF' sfe) dt (a, b) of
                            (sfe', NoEvent) -> (kSwitchAux sf1' sfe', b)
                            (_,    Event c) -> sfTF (k (freeze sf1 dt) c) a


-- kSwitch with delayed observation.
dkSwitch :: SF a b -> SF (a,b) (Event c) -> (SF a b -> c -> SF a b) -> SF a b
dkSwitch sf10@(SF {sfTF = tf10}) (SF {sfTF = tfe0}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
            let (sf1, b0) = tf10 a0
            in (case tfe0 (a0, b0) of
                    (sfe, NoEvent)  -> kSwitchAux sf1 sfe
                    (_,   Event c0) -> fst (sfTF (k sf10 c0) a0),
                b0)

        kSwitchAux sf1 (SFConst {sfCVal = NoEvent}) = sf1
        kSwitchAux sf1 sfe                          = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    let (sf1', b) = (sfTF' sf1) dt a
                    in (case (sfTF' sfe) dt (a, b) of
                            (sfe', NoEvent) -> kSwitchAux sf1' sfe'
                            (_, Event c) -> fst (sfTF (k (freeze sf1 dt) c) a),
                        b)


-- Recurring switch.
rSwitch :: SF a b -> SF (a, Event (SF a b)) b
rSwitch sf = switch (first sf) rSwitch'
    where
        rSwitch' sf = switch (sf *** notYet) rSwitch'


-- Recurring switch with delayed observation.
drSwitch :: SF a b -> SF (a, Event (SF a b)) b
drSwitch sf = dSwitch (first sf) drSwitch'
    where
        drSwitch' sf = dSwitch (sf *** notYet) drSwitch'


-- Parallel switch (dynamic collection of signal functions spatially composed
-- in parallel).
pSwitch :: Functor col =>
    col (SF a b) -> SF (a,col b) (Event c) -> (col (SF a b)->c-> SF a (col b))
    -> SF a (col b)
pSwitch sfs0 (SF {sfTF = tfe0}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
            let sfbs0 = fmap (\sf0 -> (sfTF sf0) a0) sfs0
                sfs   = fmap fst sfbs0
                bs0   = fmap snd sfbs0
            in
                case tfe0 (a0, bs0) of
                    (sfe, NoEvent)  -> (pSwitchAux sfs sfe, bs0)
                    (_,   Event c0) -> sfTF (k sfs0 c0) a0

        pSwitchAux sfs (SFConst {sfCVal = NoEvent}) = parAux sfs
        pSwitchAux sfs sfe = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    let sfbs' = fmap (\sf -> (sfTF' sf) dt a) sfs
                        sfs'  = fmap fst sfbs'
                        bs    = fmap snd sfbs'
                    in
                        case (sfTF' sfe) dt (a, bs) of
                            (sfe', NoEvent) -> (pSwitchAux sfs' sfe', bs)
                            (_,    Event c) -> sfTF (k (freezeCol sfs dt) c) a


-- Parallel switch with delayed observation.
dpSwitch :: Functor col =>
    col (SF a b) -> SF (a,col b) (Event c) -> (col (SF a b)->c->SF a (col b))
    -> SF a (col b)
dpSwitch  sfs0 (SF {sfTF = tfe0}) k = SF {sfTF = tf0}
    where
        tf0 a0 =
```

```
            let sfbs0 = fmap (\sf0 -> (sfTF sf0) a0) sfs0
                bs0   = fmap snd sfbs0
            in
                (case tfe0 (a0, bs0) of
                     (sfe, NoEvent)  -> dpSwitchAux (fmap fst sfbs0) sfe
                     (_,   Event c0) -> fst (sfTF (k sfs0 c0) a0),
                 bs0)

        dpSwitchAux sfs (SFConst {sfCVal = NoEvent}) = parAux sfs
        dpSwitchAux sfs sfe = SFTIVar {sfTF' = tf}
            where
                tf dt a =
                    let sfbs' = fmap (\sf -> (sfTF' sf) dt a) sfs
                        bs    = fmap snd sfbs'
                    in
                        (case (sfTF' sfe) dt (a, bs) of
                             (sfe', NoEvent) -> dpSwitchAux (fmap fst sfbs')
                                                            sfe'
                             (_,    Event c) -> fst (sfTF (k (freezeCol sfs dt)
                                                             c)
                                                          a),
                         bs)


-- Recurring parallel switch.
-- (Unclear if possible to derive from pSwitch, but the definition would in
-- any case be complicated.)
rpSwitch :: Functor col =>
    col (SF a b) -> SF (a, Event (col (SF a b) -> col (SF a b))) (col b)
rpSwitch sfs0 = SF {sfTF = tf0}
    where
        tf0 (a0, ef0) =
            let sfs0' = case ef0 of
                            NoEvent  -> sfs0
                            Event f0 -> f0 sfs0
                sfbs0 = fmap (\sf0 -> sfTF sf0 a0) sfs0'
                sfs   = fmap fst sfbs0
                bs0   = fmap snd sfbs0
            in
                (rpSwitchAux sfs, bs0)

        rpSwitchAux sfs = SFTIVar {sfTF' = tf}
            where
                tf dt (a, ef) =
                    let sfbs = case ef of
                                   NoEvent -> fmap (\sf -> sfTF' sf dt a) sfs
                                   Event f -> fmap (\sf -> sfTF sf a)
                                                   (f (freezeCol sfs dt))
                        sfs' = fmap fst sfbs
                        bs   = fmap snd sfbs
                    in
                        (rpSwitchAux sfs', bs)


-- Recurring parallel switch with delayed observation (aka. the DARPA switch!)
drpSwitch :: Functor col =>
    col (SF a b) -> SF (a, Event (col (SF a b) -> col (SF a b))) (col b)
drpSwitch sfs0 = SF {sfTF = tf0}
    where
        tf0 (a0, ef0) =
            let sfbs0 = fmap (\sf0 -> sfTF sf0 a0) sfs0
                bs0   = fmap snd sfbs0
            in
                (case ef0 of
                     NoEvent  -> drpSwitchAux (fmap fst sfbs0)
                     Event f0 -> drpSwitchAux (fmap fst
                                                    (fmap (\sf0 -> sfTF sf0 a0)
                                                          (f0 sfs0))),
                 bs0)

        drpSwitchAux sfs = SFTIVar {sfTF' = tf}
            where
                tf dt (a, ef) =
                    let sfbs = fmap (\sf -> sfTF' sf dt a) sfs
                        bs   = fmap snd sfbs
                    in
                        (case ef of
                             NoEvent -> drpSwitchAux (fmap fst sfbs)
                             Event f ->
                                 let sfs' = f (freezeCol sfs dt)
                                 in
                                     drpSwitchAux (fmap fst
                                                        (fmap (\sf->sfTF sf a)
                                                              sfs')),
                         bs)


------------------------------------------------------------------------------
-- Wave-form generation
------------------------------------------------------------------------------

-- Zero-order hold.
hold :: a -> SF (Event a) a
hold a0 = switch (arr dup >>> first (constant a0)) hold'
    where
        hold' a = switch (constant a &&& notYet) hold'


-- Delayed zero-order hold.
-- !!! Actually, since we're switching into a stateless signal function,
-- !!! this is just (hold a0 >>> delay a0).
-- !!! So why do we have dHold???
dHold a0 = dSwitch (arr dup >>> first (constant a0)) dHold'
    where
        dHold' a = dSwitch (constant a &&& notYet) dHold'


-- Tracks input signal when available, holds last value when disappears.
trackAndHold :: a -> SF (Maybe a) a
trackAndHold a = arr (maybe NoEvent Event) >>> hold a


-- Similar to trackAndHold, but using a delayed hold:
-- !!! Actually, this is just (trackAndHold a0 >>> delay a0).
dTrackAndHold :: a -> SF (Maybe a) a
dTrackAndHold a = arr (maybe NoEvent Event) >>> dHold a


------------------------------------------------------------------------------
-- Accumulators
------------------------------------------------------------------------------

accum :: a -> SF (Event (a -> a)) (Event a)
accum = accumBy (flip ($))


-- Identity: accumBy f = accumFilter (\b a -> let b' = f b a in (b',Just b'))
accumBy :: (b -> a -> b) -> b -> SF (Event a) (Event b)
accumBy f b_init = SF {sfTF = tf0}
    where
```

```haskell
        tf0 NoEvent    = (abAux b_init, NoEvent)
        tf0 (Event a0) = let b' = f b_init a0
                         in (abAux b', Event b')

        abAux b = SFTIVar {sfTF' = tf}
            where
                tf _ NoEvent   = (abAux b, NoEvent)
                tf _ (Event a) = let b' = f b a
                                 in (abAux b', Event b')


accumFilter :: (c -> a -> (c, Maybe b)) -> c -> SF (Event a) (Event b)
accumFilter f c_init = SF {sfTF = tf0}
    where
        tf0 NoEvent    = (afAux c_init, NoEvent)
        tf0 (Event a0) = case f c_init a0 of
                             (c', Nothing) -> (afAux c', NoEvent)
                             (c', Just b0) -> (afAux c', Event b0)

        afAux c = SFTIVar {sfTF' = tf}
            where
                tf _ NoEvent   = (afAux c, NoEvent)
                tf _ (Event a) = case f c a of
                                     (c', Nothing) -> (afAux c', NoEvent)
                                     (c', Just b)  -> (afAux c', Event b)


count :: Integral b => SF (Event a) (Event b)
count = accumBy (\n _ -> n + 1) 0


------------------------------------------------------------------------------
-- Delay
------------------------------------------------------------------------------

-- Consider changing name to pre?
delay :: a -> SF a a
delay a_init = SF {sfTF = tf0}
    where
        tf0 a0 = (delayAux a0, a_init)

        delayAux a_prev = SFTIVar {sfTF' = tf}
            where
                tf dt a = (delayAux a, a_prev)


------------------------------------------------------------------------------
-- Integral
------------------------------------------------------------------------------

-- Integration using the rectangle rule.
integral :: Fractional a => SF a a
integral = SF {sfTF = tf0}
    where
        igrl0  = 0.0

        tf0 a0 = (integralAux igrl0 a0, igrl0)

        integralAux igrl a_prev = SFTIVar {sfTF' = tf}
            where
                tf dt a = (integralAux igrl' a, igrl')
                    where
                       igrl' = igrl + a_prev * realToFrac dt
```

```haskell
------------------------------------------------------------------------------
-- Loops with guaranteed well-defined feedback
------------------------------------------------------------------------------

loopDelay :: c -> SF (a,c) (b,c) -> SF a b
loopDelay = undefined

loopIntegral :: Fractional c => SF (a,c) (b,c) -> SF a b
loopIntegral = undefined


------------------------------------------------------------------------------
-- Reactimation
------------------------------------------------------------------------------

-- Reactimation of a signal function.
-- init ....... IO action for initialization. Will only be invoked once,
--              at (logical) time 0, before first call to "sense".
--              Expected to return the value of input at time 0.
-- sense ...... IO action for sensing of system input.
--      arg. #1 ....... True: action may block, waiting for an OS event.
--                      False: action must not block.
--      res. #1 ....... Time interval since previous invocation of the sensing
--                      action (or, the first time round, the init action),
--                      returned. The interval must be _strictly_ greater
--                      than 0. Thus even a non-blocking invocation must
--                      ensure that time progresses.
--      res. #2 ....... Nothing: input is unchanged w.r.t. the previously
--                      returned input sample.
--                      Just i: the input is currently i.
--                      It is OK to always return "Just", even if input is
--                      unchanged.
-- actuate .... IO action for outputting the system output.
--      arg. #1 ....... True: output may have changed from previous output
--                      sample.
--                      False: output is definitely unchanged from previous
--                      output sample.
--                      It is OK to ignore argument #1 and assume that the
--                      the output has always changed.
--      arg. #2 ....... Current output sample.
--      result .......  Termination flag. Once True, reactimate will exit
--                      the reactimation loop and return to its caller.
-- sf ......... Signal function to reactimate.

reactimate :: IO a
              -> (Bool -> IO (DTime, Maybe a))
              -> (Bool -> b -> IO Bool)
              -> SF a b
              -> IO ()
reactimate init sense actuate (SF {sfTF = tf0}) =
    do
        a0 <- init
        let (sf, b0) = tf0 a0
        loop sf a0 b0
    where
        loop sf a b = do
            done <- actuate True b
            unless (a `seq` b `seq` done) $ do
                (dt, ma') <- sense False
                let a' = maybe a id ma'
                    (sf', b') = (sfTF' sf) dt a'
                loop sf' a' b'


------------------------------------------------------------------------------
```

```
-- Embedding
------------------------------------------------------------------------------

-- New embed interface. We will probably have to revisit this. To run an
-- embedded signal function while retaining full control (e.g. start and
-- stop at will), one would probably need a continuation based interface
-- (as well as a continuation based underlying implementation).
--
-- E.g. here are interesting alternative (or maybe complementary)
-- signatures:
--
--    sample :: SF a b -> SF (Event a) (Event b)
--    sample' :: SF a b -> SF (Event (DTime, a)) (Event b)

embed :: SF a b -> (a, [(DTime, Maybe a)]) -> [b]
embed sf0 (a0, dtas) = b0 : loop a0 sf dtas
    where
        (sf, b0) = (sfTF sf0) a0

        loop a_prev sf [] = []
        loop a_prev sf ((dt, ma) : dtas) =
            b : (a `seq` b `seq` (loop a sf' dtas))
            where
                a        = maybe a_prev id ma
                (sf', b) = (sfTF' sf) dt a


deltaEncode :: Eq a => DTime -> [a] -> (a, [(DTime, Maybe a)])
deltaEncode _  []        = afrpErr "deltaEncode" "Empty input list."
deltaEncode dt aas@(_:_) = deltaEncodeBy (==) dt aas


deltaEncodeBy :: (a -> a -> Bool) -> DTime -> [a] -> (a, [(DTime, Maybe a)])
deltaEncodeBy _  _  []        = afrpErr "deltaEncodeBy" "Empty input list."
deltaEncodeBy eq dt (a0 : as) = (a0, zip (repeat dt) (debAux a0 as))
    where
        debAux a_prev []                     = []
        debAux a_prev (a:as) | a `eq` a_prev = Nothing : debAux a as
                             | otherwise     = Just a  : debAux a as
```