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 -> 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)
    edge8By,  
      -- :: (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
    rSwitch, drSwitch,  
      -- :: SF a b -> SF (a, Event (SF a b)) b
    pSwitch, dpSwitch,  
      -- :: Functor col =>
      --   col (SF a b)
      -- -> SF a (col b)
    -- -> SF (a, col b) (Event c)
    -- -> (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

    -- :: event b -> SF (Event a) (Event a)
    accumB,  
      -- :: (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,  
      -- :: [a] Sampling interval, always > 0.
    embed,  
      -- :: SF a b -> (a, [(DTIME, Maybe a)]) -> [b]
    deltaEncode,  
      -- :: Eq a => DTIME -> [a] -> (a, [(DTIME, Maybe a)])
    deltaEncodeBy,  
      -- :: (a -> b) Bool) -> DTIME -> [a]
    -- -> (a, [(DTIME, Maybe a)])
import Monad (unless)
import Arrow
import AFRPDiagnostics
import AFRPTuple
import AFRPEvent

-- Basic type definitions with associated utilities

-- The time type is really a bit bogus, 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 :: 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 coercions, yields code which is no more efficient
-- than using SFArr in the first place.
-- (Naming: "TVar" stands for "time-input-variable").

data SF a b
    | = SFCConst {sfTF :: DTime -> a -> Transition a b, sfCVal :: b}
    | arr = ( x in R | time -> a -> Transition a b, sfCVal :: b)
    | SFIVar {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.

-- "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 NeverEvent

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

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

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 exploit 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
-- Widening.
-- The definition exploits the following identities:
-- \first{\Const{b}} \equiv \Arr{\Const{b}}\ (\_, c) \to (b, c)
-- \first{(\Arr f)} \equiv \Arr{(\Const{a}, c) \to (\Const{a}, c)}
-- (It would have been nice to exploit \first{\Const{b}} \equiv \Identity, but it would
-- seem that we need dependent types for that.)

\begin{align*}
\firstPrm \defeq & \SF{a} \to \SF{(a, c), (b, c)} \\
\firstPrm{(\SF{\TF = tf10})} \defeq & \SF{\TF = tf0} \\
\text{where} & \\
{tf0 \ (a0, c0)} \defeq & (\fAux sf1, (b0, c0)) \\
\text{where} & \\
(sf1, b0) \defeq & tf10 \ a0 \\
\fAux{(\SF{\CVal = b})} \defeq & \Arr{(\_, c) \to (b, c)} \\
\fAux{(\SF{\Fun = f})} \defeq & \Arr{(\Const{a}, c) \to (\Const{a}, c)} \\
\fAux sf1 \defeq & SFTIVar{\TF = tf0} \\
\text{where} & \\
tf dt \ (a, c) \defeq (\fAux sf1', (b, c)) \\
\text{where} & \\
(sf1', b) \defeq (\TF, sf1') dt a
\end{align*}

\begin{align*}
\secondPrm \defeq & \SF{a} \to \SF{(c, a), (c, b)} \\
\secondPrm{(\SF{\TF = tf10})} \defeq & \SF{\TF = tf0} \\
\text{where} & \\
tf0 \ (c0, a0) \defeq (\spAux sf1, (c0, b0)) \\
\text{where} & \\
(sf1, b0) \defeq tf10 \ a0 \\
\spAux{(\SF{\CVal = b})} \defeq \Arr{(c, \_) \to (c, b)} \\
\spAux{(\SF{\Fun = f})} \defeq \Arr{(\Const{c}, \_a) \to (\Const{c}, \_)} \\
\spAux sf1 \defeq SFTIVar{\TF = tf0} \\
\text{where} & \\
tf dt \ (c, a) \defeq (\spAux sf1', (c, b)) \\
\text{where} & \\
(sf1', b) \defeq (\TF, sf1') dt a
\end{align*}

instance ArrowLoop where

\begin{align*}
\loopPrm \defeq & \SF{(a, c), (b, c)} \to \SF{a \ b} \\
\loopPrm{(\SF{\TF = tf10})} \defeq \SF{\TF = tf0} \\
\text{where} & \\
tf0 \ a0 \defeq (\loopAux sf1, b0) \\
\text{where} & \\
(sf1, (b0, c0)) \defeq tf10 \ a0, c0
\end{align*}
-- 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 \( \text{sfTF} = tf0 \)
  where
    tf0 _ = if \( t_{ev} \leq 0 \) then
      (sfNever, Event x)
    else
      (afterAux 0.0, NoEvent)

afterAux t = SFTIVar \( \text{sfTF}' = tf \)
  where
    tf dt _ = let \( t' = t + dt \)
    in
      if \( t' \geq t_{ev} \) 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 \( x \) = SF \( \text{sfTF} = tf0 \)
  where
    tf0 _ = (repAux p_ev 0.0, NoEvent)
    repAux t_ev t = SFTIVar \( \text{sfTF}' = tf \)
      where
        tf dt _ = let \( t' = t + dt \)
        in
          if \( t' \geq t_{ev} \) then
            (repAux (nextEventTime \( t_{ev} \) t', Event x) \( t' \), Event x)
          else
            (repAux t_ev t', NoEvent)

nextEventTime \( t_{ev} \) =
  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 occurrence 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.
deedgeBy :: (a -> a -> Maybe b) -> a -> SF a (Event b)
dedgeBy isEdge a_init = SF \( \text{sfTF} = tf0 \)
  where
    tf0 a0 = (ebAux a0, maybeToEvent (isEdge a_init a0))

-- 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 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 \( \text{sfTF} = tf0 \)
  where
    tf0 a0 =
      let sfsbs0 = fmap (\( sf0 \) -> (sfTF sf0 a0) sfs0)
      in
        (parAux sfs sfsbs0)

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

-- !!! 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 \( \text{sfTF} = tf0 \)) k = SF \( \text{sfTF} = tf0 \)
  where
    tf0 a0 =
case tf10 a0 of
    (sf1, (b0, NoEvent)) -> (switchAux sf1, b0)
    (_, (c0, 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)
        (_, (c, 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 c0) a0)

kSwitchAux sf1 (SFConst (sfCVal = NoEvent)) = sf1
kSwitchAux sf1 = 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

-- 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 sf0@(SF (sfTF = tfe0)) k = SF (sfTF = tf0)
where
tf0 a0 =
    let sfs0 = fmap (\sfe -> (sfTF sf0) a0) sf0
        sfs = fmap fst sfs0
        bs0 = fmap snd sfs0
        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 sfs' = fmap (\sfe' -> (sfTF' sfe') dt a) sfs
        sfs' = fmap fst sfs'
        bs = fmap snd sfs'
        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 sf0@(SF (sfTF = tfe0)) k = SF (sfTF = tf0)
where
tf0 a0 =
let sfbs0 = fmap (\sf0 -> (sfTF sf0 a0) sfs0) bs0 = fmap snd sfbs0 in (case tf0 (a0, bs0) of
    (sfe, NoEvent) -> dpSwitchAux (fmap fst sfbs0) sfe
    (..., Event c0) -> fst (sfTF (k (freezeCol sfs dt) c) a0),
    bs0))

dpSwitchAux sfs = (SFCConst (sfcVal = NoEvent)) = parAux sfs
dpSwitchAux sfs = (SFIVar (sftf' = tf))
where
tf dt a =
    let sfbs' = fmap (\sf -> (sfTF sf' a) dt a) sfs
        bs = fmap snd sfbs'
    in (case (sftf' sf) 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)

rpSwitchAux sfs0 = SF (sftf = tf0)
where
tf0 (a0, ef0) =
    let sfbs0' = case ef0 of
            NoEvent -> sfs0
            Event f0 -> f0 sfs0
    sfsb0 = fmap (\sf0 -> sftf sf0 a0) sfs0'
    sfs = fmap fst sfbs0
    bs0 = fmap snd sfbs0
    in (rpSwitchAux sfs, bs0)

rpSwitchAux sfs = SFIVar (sftf' = tf)
where
tf dt (a, ef) =
    let sfbs = case ef of
            NoEvent -> fmap (\sf -> sfTF sf' df 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)
drpSwitchAux 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) bs0)

-- 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 0 notYet) hold'

-- Delayed zero-order hold. --- actually, since we're switching into a stateless signal function,
-- this is just (hold a0 >>> delay a0).

-- Accumulators

-- Identity: accumBy f = accumFilter (\b a -> let b' = f b a in (b', Just b'))

-- Similar to trackAndHold, but using a delayed hold;
afAux b' = SFTIVar {sFT' = tf}
  where
  tf _ NoEvent = (afAux b, NoEvent)
  tf _ (Event a) = let b'' = f b a
                   in (afAux 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 a) = case f c_init a 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 \(\_ \to n + 1\) 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

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.
-- \#1 \ldots. 1O 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.
-- \#2 \ldots. 1O action for sensing of system input.
-- \#3 \ldots. True: action may block, waiting for an OS event.
-- False: action must not block.
-- \#4 \ldots. 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.
-- \#5 \ldots. Nothing: input is unchanged w.r.t. the previously
-- returned input sample.
-- Just 1: the input is currently 1.
-- It is OK to always return "Just", even if input is
-- unchanged.
-- \#6 \ldots. 1O action for outputting the system output.
-- \#7 \ldots. True: output may have changed from previous output
-- sample.
-- False: output is definitely unchanged from previous
-- output sample.
-- \#8 \ldots. It is OK to ignore argument \#1 and assume that the
-- output has always changed.
-- \#9 \ldots. Current output sample.
-- result \ldots. Termination flag. Once True, reactimate will emit
-- the reactivation loop and return to its caller.
-- sf \ldots. Signal function to reactimate.

reactimate :: IO a
  => (Bool -> IO (DTTime, 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 (DT ime, a)) (Event b)

embed :: SF a b -> (a, [(DT ime, Maybe a)]) -> [b]
embed sf0 (a0, dtas) = b0 : loop a0 sf dtas
  where
    (sf, b0) = (sfT 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) = (sfT' sf) dt a

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

deltaEncodeBy :: (a -> a -> DT ime -> [a] -> (a, [(DT ime, Maybe a)]))
deltaEncodeBy _ _    = afrpErr "deltaEncodeBy" "Empty input list."
deltaEncodeBy eq dt (a0 : as) = (a0, zip (repeat dt) (deltaAux a0 as))
  where
deltaAux a_prev []   = []
deltaAux a_prev (a:as) |
  a 'eq' a_prev = Nothing : deltaAux a as
  otherwise     = Just a : deltaAux a as