An imperative implementation of functional reactive animation

Conal Elliott

`http://www.research.microsoft.com/~conal`

Microsoft Research

Draft -- comments welcome

Updated December 4, 1998

1 Introduction

I have implemented many different versions of Fran, as partially documented in [1]. All of these versions, as well as those of the TBAG [4][7] and MediaFlow [5] predecessors, and DirectAnimation [6], are based essentially on a simple denotational semantics, with various optimizations, and all of them have had performance problems.

These notes explore an approach to the implementation of Fran that is fundamentally different from any of the previous versions. Most importantly, while the previous implementations were demand-driven and mostly functional, in this implementation events are data-driven, while behaviors use demand-driven and cached sampling, with data-driven cache invalidation. The details are inspired by Pidgets++ [8].

Besides being faster than the previous Fran implementations, the other main goal of this implementation is to be well-suited for integration with imperative software components (especially non-Haskell ones).

2 Listeners, events and repeaters

A *listener* is a callback function that gets invoked on every occurrence of an event to which it is attached.

```
type Occ a = (Time, a)  -- Event occurrence

type Listener a = Occ a -> IO ()
```

To install a listener, call `addListener`, which returns an `IO` action for later removal of the listener.

```
type Remover = IO ()   -- Remove added listener, etc.

addListener :: Event a -> Listener a -> IO Remover
```

A possible representation is a mutable set of listeners.

```
newtype Event a = Event  (LSet a)      -- Possible rep (not final)
type    LSet  a = MutSet (Listener a)
```

Note that this representation contains nothing of the semantics of an event. It only allows “talking” to listeners. But what makes an event occur? The answer comes from a mechanism we call a *repeater*, which is just a linked pair of listener and event. The event repeats whatever the listener is told.

```
newRepeater :: c -> IO (Listener a, Event a)
```

Creation of a Fran event based on some external real-world event is done by invoking `newRepeater` to get a pair `(l,e)`. Then whenever the external event occurs, imperative code tells the listener `l`, which causes the Fran event `e` to occur. Fran initialization invokes `newRepeater` for each external event of interest (such as `franTick` or `lbp u` for some user `u`). Thus, just as listeners allow Fran events to cause external effects, repeaters allow external processes to influence Fran events and behaviors.

The Fran implementation uses `newRepeater` for internally-generated events, in addition to external ones. As a typical example, here is an implementation of the event-handling combinator:

```
(==>)   :: Event a -> (a -> b) -> Event b
e ==> f  = unsafePerformIO $ do
  (l',e') <- newRepeater
  addListener e (\ (t,a) -> l' (t, f a))
  return e'
```

To see how this kind of definition works, consider what happens when the event `e` occurs with some time and value `(t,a)`. Because of the `addListener` call, `l`' gets invoked with `(t, f a)`, which becomes an occurrence of the constructed event `e`', thanks to the machinery behind `newRepeater`.

The implementations of `addListener` and `newRepeater` are quite simple.

```
newRepeater = do
  lset <- newMutSet
  return ( \occ -> foreachInMutSet lset ($ occ) , lset )

addListener = addToMutSet
```

Here we have assumed primitives for mutable sets:

```
newMutSet       :: IO (MutSet a)
addToMutSet     :: MutSet a -> a -> IO Remover
foreachInMutSet :: MutSet a -> (a -> IO ()) -> IO ()
```

The trivial, never-occurring event simply throws away the listener half, so that there is no way to make event occur.

```
neverE :: Event a
neverE = unsafePerformIO $ map snd newRepeater
```

As another example, here is an implementation of the event merge operator. The trick is to forward event occurrences to the listener half of a new repeater.

```
(.|.) :: Event a -> Event a -> Event a
e .|. e' = unsafePerformIO $ do
  (l'', e'') <- newRepeater
  addListener e  l''
  addListener e' l''
  return e''
```

Finally, another interesting case is the event filtering primitive. Some occurrences of the argument event become non-occurrences of the resulting event.

```
isJustE :: Event (Maybe a) -> Event a
isJustE e = unsafePerformIO $ do
  (l', e') <- newRepeater
  let l (te, Nothing)  = return ()
      l (te, Just a)   = l' (te,a)
  addListener e l
  return e'
```

Evaluation of event expressions thus results in data-driven occurrence-propagation networks.

3 Another event representation

The representation above introduces propagation network plumbing even to handle trivial and merged events. One problem with it is that event merging creates artificial indirection between causes and effects. This indirection costs in space and time. Instead, we can represent events not as a single weak listener set, but rather as a *collection* of them. Then `neverE` would be the empty collection and "`.|.`" combines collections. Representing these collections as sets would be very simple, but then requires a somewhat expensive append operation. Instead, we use an algebraic data type.[1]

```
data Event a = NeverE
             | EitherE (Event a) (Event a)
             | LSetE (LSet a)

neverE = NeverE
```

[1] I'm not sure that the small speed-up (if any) warrants the small additional complexity, e.g., in having to write recursive functions over events instead of using maps, etc.

```
    (.|.)  = EitherE
```

A better implementation of "`.|.`", exploits the fact that `neverE` is a left- and right-identity.

```
    NeverE .|. e = e
    e      .|. NeverE = e
    e1     .|. e2 = EitherE e1 e2
```

Now `addListener` iterates over the collection.[2]

```
    addListener NeverE = const (return (return ()))

    addListener (e ‘EitherE‘ e’) = \ l -> do
      rem  <- add  l
      rem’ <- add’ l
      return (do {rem ; rem’})
     where
       add  = addListener e
       add’ = addListener e’

    addListener (LSetE lset) = lsetAdd lset
```

Note that there is no listener set associated with `neverE`. It turns out that many listeners get added to `neverE` in our implementation, so it is beneficial that using collections eliminates actual accumulation of these useless listeners. *[Preview of the handling of dead events, which includes neverE as a special case.]*

4 Reactive behaviors

Fran supports reactive behaviors using the following primitive;

```
    switcher :: Behavior a -> Event (Behavior a) -> Behavior a
```

For now, assume that behavior are represented as samplers:

```
    type Behavior a = Time -> IO a
```

The behavior "`b0 ‘switcher‘ e`" starts out acting like `b0`, switching to a new behavior whenever `e` occurs. We can implement `switcher` easily and efficiently by using a piece of state to hold the "current behavior".[3] An installed listener stores each new behavior in this state.

```
    b0 ‘switcher‘ e = unsafePerformIO $ do
      bRef <- newIORef b0
      addListener e (\(_,b) -> writeIORef bRef b)
      return (\ t -> do { currB <- readIORef bRef ; currB t })
```

5 Pruning propagation networks – weak pointers and sustainers

In any propagation network, such as described above for Fran events and reactive behaviors, the network should be pruned as clients terminate, to reclaim space and processing. In a functional or other safe language, client objects are not deleted explicitly. Moreover, references from suppliers keep the clients alive. If not corrected, this situation will cause not only space and effort to be consumed in increasing amounts.

As an example, consider the behavior "`b0 ‘switcher‘ e`", in which `b0` and each new behavior generated by e is itself reactive. Whenever a reactive behavior is switched away from, assuming that there are no more references to it, the ref it contains is no longer useful, nor is the work it takes to update it, which may come from many cascaded event handlers, filters and merging. However, the event `e` contains a pointer to the ref, so it will not be reclaimed, and the set of primitive events driving the whole network is being kept alive by its repeaters, so all the work in the network leading to obsolete events behaviors will still be executed.

We can solve both of these problems with a single technique, namely the use of *weak sets* (as in the Scheme dialect "T" as well as some Smalltalk and Java implementations) for the references that services keep to their clients. A weak set is essentially a collection of second-class, or "weak", pointers. When a client loses all first-class references, any remaining second-

[2] The style of definition supports efficient partial application.

[3] *[Discuss all of this* `unsafePerformIO` *stuff.]*

class pointers are cleared, and the client is reclaimed. Because many nodes are both clients and services, reclamation often propagates. *[Cite [1] and describe connection.]*

Weak sets have the following interface:

```
newWeakSet       :: IO (WeakSet a)
addToWeakSet     :: WeakSet a -> a -> IO Remover
foreachInWeakSet :: WeakSet a -> (a -> IO ()) -> IO ()
```

Unfortunately, while the use of weak sets solves one problem, it creates an opposite one. Since propagation networks are held together by weak pointers, they tend to disintegrate like cotton candy in a rain shower. Our solution to this new problem is to add artificial pointers. An artificial pointer is a first class pointer that serves no purpose other than keeping some object from getting reclaimed.

Weak pointers and artificial pointers are dual mechanisms forming the basis of single programming technique. When considering this technique, first ask two questions: (a) what are the service-client relationships between objects, and (b) is the service implementation to be pulled from the client or pushed from the service, i.e., demand- or data-driven? If the implementation is demand-driven, then the technique is not needed, because the client will have a pointer to the service, in order to pull results, and that pointer will sustain the service. If, on the other hand, the implementation is data-driven, give the service a weak pointer to the client (one per client), and give the client an artificial pointer to the service. The weak service-to-client pointer enables communication, while the artificial client-to-service pointer sustains the service for the lifetime of the client.

The idea of artificial pointers is embodied in the `Sustainer` abstract type.

```
data Sustainer
trivialSustainer :: Sustainer
sustain :: a -> Sustainer
```

The current implementation represents a sustainer a suspended computation that depends on the value being sustained. We don't need any constructors, since the only value is an error. The point is never to evaluate, so that the sustained values stay sustained.[4]

```
trivialSustainer = error "sustainer evaluated"

sustain a = if a ‘unsafePtrEq‘ a then
               trivialSustainer
            else trivialSustainer
```

[Ohh… make Sustainer wrap an IO (), and have sustain make a ref for a and return “readIORef ref >> return ()”.]

Now, how do we use weak sets and sustainers in Fran? As suggested above, we must first define the service-client relationships. Each listener set is served by a listener that drives it. For instance, consider the event `e′` = `e==>f`. This event is represented by a single listener set `lset`, together with a listener `l` that applies `f` to the occurrence values for `e`, and tells the members of `lset`. Since `l` listens to `e`, it is *weakly* contained in all of the listener sets in `e`. Since `l` serves `lset`, however, it must have a sustainer in the representation of `lset`.[5] Also, `e` serves `l`, so `l` must sustain `e`.[6] Thus, the following representations:

```
data Listener a = Listener (Occ a -> IO ()) Sustainer

data LSet a = LSet (WeakSet (Listener a)) Sustainer
```

We change the crucial `newRepeater` primitive to take a value (usually an event) to be sustained by the listener it constructs. The constructed weak listener set sustains the listener. Because of the circularity of weak and artificial pointers, `fixIO` shows up regularly.

---

[4] I implemented `sustain` using `unsafePtrEq`, because it is the only implemented Haskell primitive I know of that is totally polymorphic but really looks at its value. Naturally, I'd prefer using real Haskell. I think existential types would support a straightforward representation as `data Sustainer = Sustainer a`. Maybe ultimately `sustainers` have to be primitive, because any implementation I can think of could be defeated by a sufficiently smart optimizer. For instance, the implementation here could be defeated by either noticing that the `then` and `else` branches are identical (which is easily fixed) or that the predicate is always true. Also the existential representation could probably exploit a parametricity argument to justify removing the would-be sustained value from the representation. *[Comments?]*

[5] Earlier, I had put the l sustainer in the representation of `e`, but that choice is undesirable because the listener sets in an event often outlive the event that contains them. Because `lset` is accessible from `e′`, it is implicitly “sustained” (via strong pointer) by `e′`, and thus `l` is transitively sustained by `e′`. Also, with the representation of events as *collections* of listener sets, it is now the case that an event may be served by several listeners (one per `LSetE`).

[6] Minimally, `l` needs to sustain not all of `e`, but just the listeners that drive the listener sets in `e`. Sustaining all of `e` is just a short-cut. (Note that a listener implicitly sustains the listener set it drives, so we are not keeping listener sets alive unnecessarily.)

```
newRepeater :: c -> IO (Listener a, Event a)
newRepeater c = do
  fixIO ( \ ~(l’,_) -> do
    lset@(LSet wset _) <- newLSet l’
    return ( Listener (\ occ -> foreachInWeakSet wset (flip tell occ))
                      (sustain c)
           , LSetE lset ) )
```

A very common pattern is to create a listener that sustains the given event, and adds the listener to the event. The listen function encapsulates this pattern:

```
listen :: Event a -> (Occ a -> IO ()) -> IO (Listener a, Remover)
listen e iof = do
  remover <- addListener e l
  return (l, remover)
 where
   l = Listener iof (sustain e)
```

For convenience and efficiency, we define most of the event combinators by means of a more specialized function that takes the driving event `e` and a function that both decides for each occurrence of `e`, whether the new event occurs, and with what value, possibly examining and/or changing state.[7]

```
mkFilteredE :: Event a -> (Occ a -> IO (Maybe b)) -> IO (Remover, Event b)
mkFilteredE e f = do
  debugMsgLn "mkFilteredE"
  (remove,lset) <- fixIO ( \ ~(_, LSet wset _) -> do
    -- Yield a listener that applies each listener in the weak set.
    -- Apply f to occ first, because it can often do some work without
    -- knowing which listener gets the result.
    (l, remove) <- listen e $ \occ@(t,_) ->
      f occ >>= maybe (return ())
                      (\ b -> foreachInWeakSet wset (flip tell (t,b)))

    lset   <- newLSet l                 -- no listeners at first
    return (remove,lset) )

  return (remove, LSetE lset)
```

Many of the event combinators have very simple implementations. For instance:[8]

```
(==>)  :: Event a -> (a -> b) -> Event b
e ==> f = unsafePerformIO $ map snd $
  mkFilteredE e (return . Just . f . snd)

isJustE :: Event (Maybe a) -> Event a
isJustE e = unsafePerformIO $ map snd $
  mkFilteredE e (return . snd)

-- Create and snapshot a behavior at an event occurrence
snapshot :: Event a -> Behavior b -> Event (a,b)
e ‘snapshot‘ f = unsafePerformIO $ map snd $
  mkFilteredE e $ \(te,x) -> do
    (y,_) <- b ‘at‘ te
    return (Just (x, y))

-- Just the first occurrence of a given event
onceE :: Event a -> Event a
onceE e = unsafePerformIO $ map snd $
  fixIO $ \ ~(remove,_) ->
    mkFilteredE e $ \ (_,a) -> do
      remove
      return (Just a)
```

[7] It may be worthwhile to allow `f` to generate more than one occurrence. If so, replace `Maybe b` with `[b]`.

[8] The behavior sampling function `at` used in `snapshot`, is described in Section 7.

6 Reactive events

An event may itself change with time, in response to other events. It's a somewhat obscure feature, but in case you're interested, here's the primitive.

```
switcherE :: Event a -> Event (Event a) -> Event a
```

In the implementation, whenever the event-valued event occurs, we drop interest in the old event, and start listening to the new one. A call to `newRepeater` gives the necessary indirection so that `switcherE` can return a single event driven by various events over time.

```
e ‘switcherE‘ ee = unsafePerformIO $ map snd $ do
  eRef <- newIORef e
  fixIO $ \ ~(eel,e’) -> do
    -- The listener sustains the current event and the ee listener
    (l’, e’)   <- newRepeater (eRef,eel)
    -- e’ mimics e initially
    removerRef <- addListener e l’ >>= newIORef

    (eel,_) <- listen ee $ \(_,newEv) -> do
       -- Uninstall old listener and install new one.
       readIORef removerRef >>= id
       addListener newEv l’ >>= writeIORef removerRef
       -- Sustain new event instead of old.
       writeIORef eRef newEv

    return (eel, e’)
```

7 Event arrays

The `arrayE` primitive enables efficient "event routing", by turning a single pair-valued event into an array of events. See Appendix B of [3] for a sample use.

```
arrayE :: Ix a => (a,a) -> Event (a,b) -> Array a (Event b)
```

The implementation works by turning an occurrence `(a,b)` of the given event into occurrence `b` of the event indexed by `a` in the resulting array. As usual, `newRepeater` provides the needed connections.

```
arrayE bounds abEv = unsafePerformIO $ do
  -- Make listeners and events.  First lists and then arrays.  Each event
  -- sustains the corresponding listener, and each listener sustains abEv.
  (ls, bEvs) <- map unzip (accumulate (replicate size (newRepeater abEv)))
  let evArr = array bounds (zip as bEvs)
      lArr  = array bounds (zip as ls)
  -- Turn occurrence (a,b) of abEv into occurrence b of evArr!a
  listen abEv $ \(te,(a,b)) -> tell (lArr ! a) (te,b)
  return evArr
 where
   as   = range     bounds
   size = rangeSize bounds
```

8 Efficient behaviors

The implementation of behaviors is built on top of the simpler notion of a "sampler", which is a sampling function that yields a value and a flag saying whether the sample value is at least temporarily constant.

```
type Sampler a = Time -> IO (a, Bool)
```

A behavior is something that can be sampled, and has an "invalidation event" that occurs whenever a constant segment ends.

```
data Behavior a = Behavior (Sampler a) (Event ()) Sustainer

at (Behavior f _) = f
invalidateEv (Behavior _ e) = e
```

Constant behaviors always return the same value, and affirm constancy:

```
constantB :: a -> Behavior a
constantB a = Behavior (const (return (a, True))) neverE
```

The behavior "`time`" has value *t* at time *t*, and has no constant segments:

```
time :: Behavior Time
time = Behavior (\ t -> return (t,False)) neverE
```

The third and final non-reactive behavior primitive, "**`$*`**" combines a function-valued and an argument-valued behavior, like the classic S combinator. Its implementation combines not only the sampled values, but also the constancy flags. An application is constant if the function and argument parts are. (A sufficient, but not necessary, condition.)

```
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
fb $* xb = Behavior sampler invalidate
 where
   sampler t  = do (f,fConst) <- fb ‘at‘ t
                   (x,xConst) <- xb ‘at‘ t
                   return (f x, fConst && xConst)
   invalidate = invalidateEv fb .|. invalidateEv xb
```

Fran then uses "**`$*`**" to define n-ary lifting functionals:

```
lift0                         = constantB
lift1 f b1                    = lift0 f $* b1
lift2 f b1 b2                 = lift1 f b1 $* b2
lift3 f b1 b2 b3              = lift2 f b1 b2 $* b3
...
```

These functionals are then used to define many lifted functions. Overloading makes the lifting transparent most of the time. Some functions, especially non-strict ones, merit special treatment. Here is a lifted conditional:[9]

```
condB :: Behavior Bool -> Behavior a -> Behavior a -> Behavior a
condB c b b’ = Behavior sampler invalidate
 where
  sampler t = do (cVal,cConst) <- c ‘at‘ t
                 (vVal,vConst) <- (if cVal then b else b’) ‘at‘ t
                 return (vVal, cConst && vConst)

  invalidate = invalidateEv c
             .|. invalidateEv b  ‘whenE‘ c
             .|. invalidateEv b’ ‘whenE‘ (notB c)
```

8.1 Caching

A problem with our behavior implementation above is that it redundantly samples behaviors that are used more than once. As in [2], we use lazy memoization to avoid the redundant computation. Because we assume time-monotonic sampling and event processing as a global invariant,[10] it suffices to cache a single sampling.

```
type BCache a = IORef (Time, (a,Bool))

newBCache :: IO (BCache a)
newBCache = newIORef (minTime, (error "Accessed undefined cache value", False))
```

Rather than building caching into all behaviors, we introduce it only where merited, as in "**`$*`**" and "`condB`".

The basic tool is `cachingSampler`, which takes a cache and sampler, as well as a value to be sustained, and yields a caching sampler.

```
cachingSampler :: BCache a -> Sampler a -> b -> Sampler a
cachingSampler cache sampler sustainee t = do
  (tLast, p@(_, isConst)) <- readIORef cache
  if isConst || tLast==t then
     return p
   else do
     p <- sampler t
     writeIORef cache (t,p)
     return p
```

[9] Really, `condB` is defined as an optimization of a class method. The definition given here is the behavior version of that method.

[10] *[State this restriction up front.]*

Note that `sustainee` argument is not used above. It is only there so that the constructed sampler will sustain it. The value is typically an event listener that writes to state being read by the sampler.

Then we make a caching behavior as follows:

```
cachingBehavior :: Sampler a -> Event () -> Behavior a
cachingBehavior sampler invalidate = unsafePerformIO $ do
    cache <- newBCache

    -- At each invalidation, check the cache.  If it contains a constant
    -- value, replace the sample time by the invalidation time, and mark
    -- as non-constant, since events don't affect a behavior until
    -- immediately *after* the occurrence.
    (l,_) <- listen invalidate $ \(te,_) -> do
       (tLast, (x, isConst)) <- readIORef cache
       when isConst (writeIORef cache (te, (x, False)))

    -- Since the listener serves the sampler (by writing to the ref that
    -- the sampler is reading), cause the sampler to sustain the
    -- listener.
    return $ Behavior (cachingSampler cache sampler l) invalidate
```

Now, in order to make "`$*`" and `condB` exploit caching, simply call `cachingBehavior` in place of the `Behavior` constructor.

8.2 Reactive behaviors

Reactive behaviors are based on the `switcherB` primitive, whose implementation is explained below.

```
switcherB :: Behavior a -> Event (Behavior a) -> Behavior a

b0 `switcherB` e = unsafePerformIO $ do
  samplerRef <- newIORef (at b0)

  (l,_) <- listen e $ \(_,b) ->
     addDelayedAction $ writeIORef samplerRef (at b)

  -- Sustains its first argument when partially applied.
  let sampler sustainee t = readIORef samplerRef >>= ($ t)

  return $ Behavior (sampler l)
                    (e -=> () .|.
                     (invalidateEv b0 `switcherE` e ==> invalidateEv))
```

Some explanation:

- The implementation keeps a ref holding the current sampler. Whenever the event occurs, the sampler is replaced.
- The event listener serves the sampler, in that the listener writes to state from which the sampler reads. Therefore, the sampler sustains the listener.
- The semantics of `switcherB` say that the behavior switches immediately *after* the event occurrence. There might, however, be samplings on the current time, still waiting to be done. For this reason, we must delay the sampler replacement, using `addDelayedAction`.
- The sampler is responsible for sustaining the event listener, because it reads from state that the listener writes to.
- The behavior constructed by `switcherB` may have constant segments, so we need to define an appropriate invalidation event. There are two possible causes for a constant segment to end. One is that `switcherB`'s event argument occurs, and the other is that a constant segment of the current behavior ends. The second case is handled by the event-level switcher function.[11]

[11] The `GBehavior` type class has a switcher method whose definition for behaviors is `switcherB` and for events is `switcherE`.

9 IO-valued events and behaviors

It is often useful to set up a sequence of repeated actions to be executed over time and interleaved with other such sequences. Since events represent timed sequences of values in general, it is perhaps not too surprising that IO-valued events provide a powerful and convenient way to specify sequences of actions. The essential primitive is `doE`:

```
doE :: Event (IO ()) -> IO Remover
```

The remover returned allows for later termination. Here is a straightforward, but flawed implementation. (The use of `snd` as an argument to `listen` is to extract the `IO ()` out of the occurrence. The `map snd` is to extract the remover from the listener/remover pair returned by `listen`.)

```
doE ioe = map snd $ listen ioe snd
```

The flaw in this implementation is that there is nothing to sustain the `ioe` listener. It will evaporate at the next GC. We could return it from `doE`, but it would be inconvenient for the client to sustain it.[12] For this reason, `doE` really uses a global mutable set that sustains listeners.

It is also useful to perform some actions as often as possible, rather than at meaningfully determined times. In such cases, it is natural to express the actions as IO-valued behaviors, rather than events.

```
addIOB :: IOB () -> IO Remover
```

For technical reasons that will probably go away, we will need to use behaviors over functions from time to `IO ()`.

```
type TIOB a = Behavior (Time -> IO a)

addTIOB :: TIOB () -> IO Remover
```

The `IO` functions must be idempotent, even when given different time arguments. The time argument might be used to say when an update is to take place, but not what, as in the sprite-engine. This assumption is critically important, since it allows the implementation to be efficient in the common case that a behavior being output is temporarily or permanently constant. Idempotence seems a reasonable restriction, because the timing or number of executions cannot be specified, and so non-idempotent actions would have indeterminate results.

The implementation of `addTIOB` maintains a pool of work items, each of which contains a `TIOB`.

```
data WorkItem = WorkItem
  (TIOB ())                    -- to do
  (IORef Bool)                 -- active?
  (IORef (Maybe Remover))      -- listener remover unless removed
```

The pool is traversed as frequently as possible. For each item, the `TIOB` is sampled, and the resulting `Time -> IO ()` function is invoked. If the `TIOB` is currently constant, the item is then removed from the work pool, and otherwise it is kept. When a constant segment from a removed work item ends, an invalidation listener established by `addTIOB` reinserts the item in work pool if it is not already there (according to the boolean ref). If the new segment is constant also, then the new value will get executed just once, and removed from the work pool again.

```
addTIOB tiob = do
  activeRef  <- newIORef False
  removerRef <- newIORef Nothing
  let item    = WorkItem tiob activeRef removerRef
      restore = restoreWorkItem item
  -- Intially put in work pool.  Then on each
  -- invalidation, restore if inactive.
  restore
  remove <- doE $ invalidateEv tiob -=> do
                    isActive <- readIORef activeRef
                    unless isActive restore
  writeIORef removerRef (Just remove)
  return (killTIOB item)
```

[12] Well … I guess that's what I was thinking. Now I'm not so sure.

```
restoreWorkItem :: WorkItem -> IO ()
restoreWorkItem item@(WorkItem _ activeRef _) = do
  writeIORef activeRef True
  updateIORef theWorkPool (return . (item:))
```

The function `updateIORef` is just a useful utility for updating refs:

```
updateIORef :: IORef a -> (a -> IO a) -> IO ()
updateIORef ref iof = readIORef ref >>= iof >>= writeIORef ref
```

Killing a work item stops the behavior change listener if running, and marks the work item as deleted. It doesn't actually get removed from a work pool until the next time `doWork` is called.

```
killTIOB :: TIOBTag -> Remover
killTIOB (WorkItem _ _ removerRef) =
  readIORef removerRef >>=
    maybe (wError "killTIOB: already destroyed!")
          (\ remove -> do remove ; writeIORef removerRef Nothing)
```

The function `runTIOBsOnce` actually causes the work items to be processed, and eliminates deleted and constant items.

```
runTIOBsOnce :: Time -> IO ()
```

Fran's display functions call `runTIOBsOnce`. When Fran is used with HaskellScript, the following function should be used instead. The two arguments identify a "timer" by its object name and event name in the container window.[13]

```
runTIOBs :: String -> String -> IO (IEventSink Interface)
```

10 Event termination

[The ideas in this section have not been implemented.]

Under some circumstances, the implementation can know that an event will have no more occurrences.

- `onceE e'`, when `e'` either terminates or occurs;
- `e' ==> f`, `e' 'suchThat' p`, etc, for a terminated `e'`;
- `e' .|. e''`, where `e'` and `e''` have both terminated;
- `e0 'switcher' ee`, where `ee` has terminated and the current (last) event has terminated;
- `arrayE e'`, when `e'` terminates; or
- `predicate b`, when `b` has gone false forever.

How can a behavior be known to have gone constant forever?

- it's currently constant, and its invalidation event has terminated; or
- if it is a switcher whose event has terminated and whose current behavior has gone constant forever.

Could we use events for event termination? Represent an event as a collection of listener sets, a ref containing the number of live listener sets remaining, and a termination event.

```
termE (onceE e') == e' -=> () .|. termE e'

termE (e' ==> f) == termE e'

termE (isJustE e') == termE e'

termE (e' .|. e'') == (termE e' >> termE e') .|. (termE e'' >> termE e')
```

Make a listener know how many live listener sets are driving it. When the count drops to zero, kill the listener set that it's driving.

[13] To do: make this function create a timer by itself.

11 Miscellany

° `addDelayedAction`

° Should events be able to go dormant? Suppose an event has listeners, but they don't contribute to anything visible. This situation is typical shortly before a weakly sustained event is GC'd, but it may occur at other times as well. I think we should support dormancy, so that events don't eat up processing when they shouldn't. That way, GC really just manages memory, not work. It's cool to GC work, but the GC is only triggered by running low on space, not on time, so it artificially encourages small heap sizes, which I think is bad for GC performance.

12 Acknowledgements

Sigbjorn Finne and Simon Peyton Jones discussed many representation possibilities and issues with me. Meurig Sage has helped with various aspects of the implementation, found big and little bugs, and generally provided very helpful feedback along the way.

References

[1] Henry G. Baker and Carl E. Hewitt, The Incremental Garbage Collection of Processes, *ACM SIGPLAN Notices*, 12(8), pp. 55-59, ACM Press, August 1977.

[2] Conal Elliott, Functional Implementations of Continuous Modeled Animation, in *Proceedings of PLILP/ALP '98*. Expanded version at `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?-RecordID=164`.

[3] Conal Elliott, *A "Fifteen Puzzle" in Fran*. Microsoft Research tech report, October, 1998. `http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=187`.

[4] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi, TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. *Proceedings of SIGGRAPH '94*. `http://www.research.microsoft.-com/~conal/papers/siggraph94.ps`.

[5] Conal Elliott, Greg Schechter, and Salim Abi-Ezzi. *MediaFlow, a Framework for Distributed Integrated Media*. Sun Microsystems Laboratories, June 1995. Technical Report SMLI TR-95-40.

[6] Microsoft Corporation, *DirectAnimation Overview*, `http://microsoft.com/directx/pavilion/danim/-default.asp`.

[7] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D Graphics in C++ — with an Object-Oriented, Multiple Dispatching Implementation. *Proceedings of the 4th Eurographics Workshop on Object-Oriented Graphics*, 1994. `http://www.research.microsoft.com/~conal/papers/eoog94.ps`.

[8] Enno Scholz and Boris Bokowski, PIDGETS++ - A C++ Framework Unifying PostScript Pictures, GUI Objects, and Lazy One-Way Constraints, Conference on the Technology of Object-Oriented Languages and Systems (TOOLS USA 96). `http://www.inf.fu-berlin.de/~scholz/tools-published.ps.gz`