Functional Implementations of Continuous Modeled Animation (Expanded Version)

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

July 6, 1998

Technical Report
MSR-TR-98-25

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Functional Implementations of Continuous Modeled Animation (Expanded Version)

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

July 6, 1998

Abstract

Animation is a temporally *continuous* phenomenon, but is typically programmed in terms of a discrete sequence of changes. The use of discreteness serves to accommodate the machine that is *presenting* an animation, rather than the person *modeling* an animation with the help of a computer. Using a continuous model of time for animation allows for natural specification, avoiding some artificial details, but is difficult to implement with generality, robustness and efficiency. This paper presents and motivates *continuous modeled animation*, and sketches out a naive functional implementation for it. An examination of some of the practical problems with this implementation leads to several alternate representations, all of which have difficulties in themselves, some quite subtle. We hope that the insights and techniques discussed in this paper lead to still better representations, so that animation may be specified in natural terms without significant loss of performance.[1]

1 Introduction

A functional approach to animation offers the possibility to make animations much easier and more natural to express, by removing the need to direct the details of *presentation* and allowing the programmer instead to say what an animation is [1, 11]. Following the evolution of 3D graphics, we have termed this approach "modeling", as opposed to "presentation" [8].

Given that we want to model animations, what notion of time should we adopt? The first fundamental choice is discrete vs continuous, that is do we

[1]A shorter version of this report appears in the proceedings of PLILP/ALP '98, and is © Springer-Verlag.

think of time as moving forward in a (discrete) sequence of "clock ticks", or a (continuous) flow?

A discrete model of time fits more easily into popular *computational* concepts, because modern computers operate in temporally discrete steps. More specifically, animations are presented as a sequence of frames, delivered to the user at a finite rate.

On the other hand, a continuous model of time seems a more natural fit with our human perception of time. Moreover, we have a rich heritage of mathematical, scientific, and engineering tools for understanding and describing the basic animation concepts of motion, growth, etc, and these tools are based on the continuous notion of time. These tools include polynomials, trigonometry, and calculus, together with their applications to physical motion, governed by Newton's law of motion and its applications to gravitational attraction, springs, etc.

To clarify the benefits of continuous animation, note that an analogous discrete-vs-continuous choice applies to the concept of an *image*. We may view an image either discretely, as a finite array of colors, or continuously, as an assignment of colors to continuous 2D space. Again, the discrete model accommodates computers well, while the continuous model fits better with our human perception and hence our intuitions. One concrete benefit of continuous images is that they are infinitely stretchable, or in other words, resolution-independent. This flexibility is one of the reasons for the popularity of vector graphics image representations in software for graphic design. Another reason is that representations are often much more compact than their discrete counterparts. Both of these advantages stem from the fact that vector graphics representations store concise declarative specifications, such as piecewise-cubic curves, that serve to outline parts of an image. These descriptions are then rendered in real time to produce discrete bitmaps for display or printing. For text in particular, this idea is called "scalable fonts", and was popularized by Postscript [17].

Just as continuous image models naturally give rise to spatial resolution-independence and hence scalability, the continuous time model for animation yields *temporal* resolution-independence and scalability. For instance, one can stretch or squeeze a given animation to fit a desired duration.

Fran (*Functional reactive animation*) is a theory of animation, and implementation as a Haskell library, that is based on a continuous time model. As such, it offers the possibility of allowing people to express animations in ways that match their intuitive understandings and that leverage the wealth of mathematical tools at their disposal.

The cost of offering a continuous time model is a more challenging implementation problem, since continuous animation descriptions must be translated into discrete lower-level software and hardware structures.[2] In fact, we have

[2]The translation from continuous to discrete inevitably introduces error. Usually these errors are negligible, but sometimes, as in some systems of differential equations or condition-

implemented Fran many times, and have not yet achieved a satisfactory combination of generality, robustness and efficiency. Several subtle difficulties became apparent to us only through implementation and experimentation.

The purpose of this paper is to present the implementation techniques we have explored, together with some insight into practical difficulties, and some ideas for other possible techniques, in the hope of facilitating work leading to still better representations.

Section 4 presents a few more implementation alternatives that we have investigated, but have implemented only partially or not at all.

Previous papers on Fran have presented its vocabulary and semantics, its role as an "embedded language", and examples of its use [11, 9, 8, 23]. This paper gives only minimal treatment of these issues.

2 A User Perspective on Fran

Fran is a declarative animation library (or "embedded language") that embodies the continuous time model of animation, and is implemented in Haskell [13].

2.1 Behaviors

For any type `ty`, the Fran type `Behavior ty` represents `ty`-valued animations, i.e., functions from continuous time to `ty`. As an example, consider the following definition of a real-valued animation, which has value 1 at time 0 and grows quadratically:

```
type RealB = Behavior Double   -- helpful synonym

growQuad :: RealB
growQuad = 1 + time ** 2
```

There is some notational magic going on here. We are using behavior-specific overloadings of familiar numerical operators for addition and exponentiation, and even numeric literals, using Haskell's type class mechanism. The types used in this example are as follows:

```
1, 2      :: RealB
(+), (**) :: RealB -> RealB -> RealB
time      :: RealB
```

Literals like 1 and 2 are implicitly converted into behaviors through application of the polymorphic `constantB` function:

```
constantB :: a -> Behavior a
```

based events, errors can become significant. These problems are inherent in applying computers to simulate continuous phenomena, regardless of the programming paradigm.

For any value x, the behavior constantB x has value x at every time.

The behavior versions of functions like + and **, as well as sin, log, etc., are all defined in terms of the non-behavior versions, by using lifting operations.

```
lift1 :: (a -> b) ->
         Behavior a -> Behavior b
lift2 :: (a -> b -> c) ->
         Behavior a -> Behavior b -> Behavior c
lift3 :: (a -> b -> c -> d) ->
         Behavior a -> Behavior b -> Behavior c -> Behavior d
-- etc
```

Lifting is hidden inside of type class instances, with declarations like the following:

```
instance  Num a => Num (Behavior a)  where
  (+)        =  lift2 (+)
  (*)        =  lift2 (*)
  negate     =  lift1 negate
  abs        =  lift1 abs
  fromInt n  =  constantB (fromInt n)
```

The first line says that for any "number type" a, the type of a-valued behaviors is also a number type. The second line says that the behavior version of + is the binary-lifted version of the unlifted + operation. (The definitions appear self-referential, but are not. Overload resolution distinguishes the two different versions of "+".) The function fromInt is used to resolve literal integer overloading. The reason for this definition, instead of the more obvious one involving lift1, is that fromInt must still work on numbers, not number-valued behaviors. This requirement is partly desirable, but is also partly due to the restricted type of fromInt. In such cases, Fran provides additional definitions with names formed by adding "B" or "*" to the unlifted version, such as the following.

```
fromIntB :: Num a => Behavior Int -> Behavior a
fromIntB = lift1 fromInt

(==*) :: Eq a => Behavior a -> Behavior a -> Behavior Bool
(==*) =  lift2 (==)
```

Because Behavior is a type constructor, we can apply it to any type. For instance, when applied to a type Point2 of 2D static points, the result is the type of motionsn in 2D space.

It is often quite natural to express behaviors in terms of time-varying velocities. For this reason, Fran supports integration over a variety of types, as long as they implement vector space operations.

```
integral :: VectorSpace v => Behavior v -> User -> Behavior v
```

Examples of vector space types are reals, 2D vectors, and 3D vectors. The "user" argument to `integral` has two roles. One is to choose the integral's start time, and the other is to assist choice of step-size in the numerical solution. If a user `u` has start time t_0, then `integral b u` is a behavior that at time t has approximately the value $\int_{t_0}^{t} b(t')dt'$.

For modularity, it is useful to construct a behavior and then, separately, transform it in time. Fran, therefore, has a time transformation operation. Since a time transform remaps time, it is specified as a time-valued behavior, which semantically is a function from time to time.

```
timeTransform :: Behavior a -> Behavior Time -> Behavior a
```

The lifting operators can all be expressed in terms of a single operator:

```
($*) :: Behavior (a -> b) -> Behavior a -> Behavior b
```

The name "`$*`" comes from the fact that it is the lifted version of function application, whose Haskell infix operator is "`$`".

The lifting operators are defined simply in terms of `constantB` and "`$*`", as follows.

```
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
-- etc
```

Note that the basic combinators "`$*`", `constantB`, `time`, and `timeTransform` correspond semantically to type-specialized versions of the classic S, K, I, and B combinators.

2.2 Events

Besides behaviors, the other principle notion in Fran is the *event*. For any type `ty`, the Fran type `Event ty` is the type of `ty`-valued events, each of which is semantically a time-sorted sequence of time/`ty` pairs, also refered to as "occurrences" of the event.

This paper is chiefly concerned with the implementation of continuous *behaviors*, rather than events (which are intrinsically discrete). We will informally describe event operators as they arise.

2.3 Reactive Behaviors

While one can define an infinite set of behaviors using just the behavior combinators given above, such behaviors are not very dynamic. Fran greatly enriches these behaviors by supporting reactivity, via the following primitive, which uses behavior-valued events.

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

Given behavior `b` and an event `e`, the behavior `` b `untilB` e `` acts like `b` until `e` occurs, and then switches to acting like the behavior that accompanies the event occurrence. Although `untilB` only needs the *first* occurrence of an event, the other occurrences are used by some of the event combinators. This meaning of events turned out to be more convenient than the one in [11], since it enables higher level combinators for constructing reactive behaviors, as illustrated in [8].

3 Implementing Continuous Behaviors

We now turn to the main thrust of our paper, which is an exploration of how to implement continuous behaviors.

Representation A: time-to-value functions. The semantics of behaviors suggest a very simple representation:

```
newtype Behavior a = Behavior (Time -> a)

-- Sample a behavior "at" a given time
at :: Behavior a -> Time -> a
Behavior f `at` t = f t
```

This definition introduces both a new type constructor and a value constructor, both called `Behavior`. (A simple `type` definition in Haskell would not have allowed overloading.) Using this representation, it is easy to define our simple behavior combinators:

```
constantB x = Behavior (const x)

fb $* xb = Behavior (\ t -> (fb `at` t) (xb `at` t))

time = Behavior (\ t -> t)

timeTransform b tt = Behavior (at b . at tt)
```

To sample a behavior constructed by `` b `untilB` e `` at a time t, first check whether the event `e` occurs before t.[3] If so, sample the new behavior, `b'`, that is part of the event occurrence, and if not, sample `b`.

[3] By choosing *before*, rather than before *or at*, the sampling time t, animations may be self- or mutually-reactive.

```
b `untilB` e = Behavior sample
 where
   sample t = case (e `occ` t) of
                Nothing  ->  b  `at` t
                Just b'  ->  b' `at` t
```

Here we presume a function `occ`, with the following signature, for checking for an event occurrence before a given time.

```
occ :: Event a -> Time -> Maybe a
```

The reason we cannot represent events directly via their semantics, i.e., as a list of occurrences (time/value pairs), is that `untilB` will often need to that an event has no occurrence before a time t, before it is possible to know the time of the first occurrence. An event is thus represented as a time-sorted list of "possible occurrences", each of which consists of a time and a `Maybe` value.

```
newtype Event a = Event [(Time, Maybe a)]
```

The `occ` function simply searches a possible occurrence list looking for a genuine occurrence before the given time.

A thorough description of Fran's implementation of events is outside the scope of this paper, but here is one case:

```
-- The event e .|. e' corresponds to the union of occurrences
-- of e and e', listing e occurrences before e' occurrences
-- when simultaneous.

(.|.) :: Event a -> Event a -> Event a
Event possOccs .|. Event possOccs' =
  Event (merge possOccs possOccs')
 where
   merge os@(p@(te, mb) : osRest) os'@(p'@(te', mb') : osRest')
     | te  <=  te' =  p  : merge osRest  os'
     | otherwise   =  p' : merge os      osRest'
   merge [] os' = os'
   merge os []  = os
```

3.1 The Problem of Non-Incremental Sampling

While Representation A given above is appealing in terms of simplicity, it has a serious performance problem. It allows nothing to be remembered from one sampling to another. In animation, however, behaviors are typically sampled at a sequence of display times separated by small differences. For instance, given an integral behavior, it is vital for efficiency that intermediate results are carried from one sampling to the next, so that only a small amount of extra work is required. Similarly, for a *reactive* behavior, i.e., one constructed with `untilB`, it is important to make incremental progress in event detection.

Consider the implementation of `untilB` above, and suppose we want to sample a reactive behavior with times $t_1, t_2, \ldots$. For every t_i, sampling must consider whether the event has occurred, and choose to sample the old behavior or the new one. Moreover, it is frequently the case that the new behavior it itself reactive, and so on, in an infinite chain. In such a case, the time it takes to compute each sample will increase without bound, which is clearly unacceptable. Moreover, consider what is involved in computing "`e ‘occ‘ t`" in `untilB`. There are two possibilities: either event occurrences are being rediscovered, or they are cached somehow. If they are rediscovered, then the cost of sampling increases even more so with each sampling. If, however, they are cached, then the cache itself becomes a large space leak. Together these problems cause what we call a "space-time leak".

Representation B: residual behaviors. A crucial observation is that typically, the sampling times t_1, t_2, ..., are monotonically increasing. If we assume that this typical case holds, then we can remove the space-time leak.[4] The idea is to have sampling yield not only a value, but also a "residual behavior".

```
newtype Behavior a = Behavior (Time -> (a, Behavior a))

at :: Behavior a -> Time -> (a, Behavior a)
Behavior f ‘at‘ t = f t
```

The combinator implementations are not quite as simple as before, but still reasonable. Constant behaviors always return the same pair:

```
constantB x = b
 where
   b = Behavior (const (x, b))
```

The `time` behavior is also simple.

```
time = Behavior (\ t -> (t, time))
```

The "`$*`" combinator samples its argument behaviors, applies one resulting value to the other, and applies "`$*`" to the residual behaviors.

```
fb $* xb = Behavior sample
 where sample t = (f x, fb’ $* xb’)
         where (f, fb’) = fb ‘at‘ t
               (x, xb’) = xb ‘at‘ t
```

Time transformation can be implemented much like "`$*`".

[4] Unfortunately, the cost of this assumption is a significant restriction in the time transforms that may be applied to reactive behaviors.

```
timeTransform b tt = Behavior sample
 where sample t = (x, timeTransform b' tt')
         where (t', tt') = tt `at` t
               (x , b' ) = b  `at` t'
```

Time transformation can violate our monotonicity assumption for time streams, if the time transform `tt` is not itself monotonic. It would be possible to check for monotonic sampling dynamically, at least for reactive behaviors, though Fran does not do so. Checking monotonicity "statically", i.e., when a `timeTransform` or `untilB` behavior is constructed does not seem to be feasible.

The `occ` function, used for checking event occurrences, is changed in a way much like behaviors, so that it now yields a residual event along with a possible occurrence.

```
occ :: Event a -> Time -> (Maybe a, Event a)
```

Next, consider reactivity. Sampling a reactive behavior with a time that is after the event's first occurrence, the newly constructed behavior (a) no longer checks for the event occurrence, thus eliminating the time leak, and (b) no longer holds onto the old behavior, thus eliminating the space leak.

```
b `untilB` e  = Behavior sample
 where
   sample t =
     case (e `occ` t) of
       -- No occurrence before t; keep looking
       (Nothing, eNext) -> let (x, bNext) = b `at` t in
                             (x, bNext `untilB` eNext)

       -- Found it; sample new behavior
       (Just bNew, _)  -> bNew `at` t
```

Representation C: stream functions. An alternative solution to the problem of non-incremental sampling is to map time streams to value streams. We will see in Section 3.2 that this representation has advantages over Representation B.

```
newtype Behavior a = Behavior ([Time] -> [a])

at :: Behavior a -> [Time] -> [a]
Behavior f `at` ts = f ts
```

Constant behaviors always return the same list containing an infinite repetition of a value:

```
constantB x = Behavior (const (repeat x))
```

The "`$*`" combinator samples the function and argument behaviors, and applies resulting functions to corresponding arguments, using the binary mapping functions `zipWith`.

```
fb $* xb =
  Behavior (\ ts -> zipWith ($) (fb `at` ts) (xb `at` ts))
```

Time is the identity as it was in Representation A, but of a different type:

```
time = Behavior (\ ts -> ts)
```

Reactivity is implemented by scanning through a list of possible event occurrences, while enumerating behavior samples. For convenience, assume a stream sampler function for events:

```
occs :: Event a -> [Time] -> [Maybe a]
```

The implementation of `untilB`:

```
b `untilB` e =
  Behavior (\ ts -> loop ts (b `at` ts) (e `occs` ts))
 where
   -- Non-occurrence.  Emit first b sample and continue looking
   loop (_:ts') (x:xs') (Nothing:mbOccs') =
      x : loop ts' xs' mbOccs'
   -- First event occurrence.  Discard the rest of the b values
   -- and possible event occurrences, and continue with the
   -- new behavior
   loop ts      _       (Just bNew : _  ) = bNew `at` ts
```

A weakness of Representations B and C is that they cause a great deal of construction with each sampling. For this reason, it is important to have a garbage collector that deals very efficiently with the rapid production of short-lived structures, as in generational garbage collection. For instance, the Glasgow Haskell Compiler [19] has such a collector.

3.2 The Problem of Redundant Sampling

Another serious problem with all the representations preceding is that they lead to redundant sampling. As a simple example, consider the following behavior that linearly interpolates between two numerical behaviors `b1` and `b2`, according to the interpolation parameter `a`, so that the resulting behavior is equal to `b1` when `a` is zero and `b2` when `a` is one.

```
interp :: RealB -> RealB -> RealB -> RealB
interp b1 b2 a  =  (1 - a) * b1 + a * b2
```

The problem is that sampling the behavior generated by `interp` at some time t ends up sampling `a` at t twice, due to the repeated use of `a` in the body of `interp`. If `a` is a complex behavior the redundant sampling is costly. Worse yet, composition of functions like `interp` multiplies the redundancy, as in the following example.

```
doubleInterp :: RealB -> RealB -> RealB -> RealB -> RealB -> RealB
doubleInterp b1 b2 b3 b4 a = interp b1 b2 a'
  where a' = interp b3 b4 a
```

When the result of `doubleInterp` is sampled with a time t, the behavior `a'` will be sampled twice at t, which means `a` will be sampled four times, and the interpolation work for `a'` done twice. If we were to cube the result of `doubleInterp`, via the definition

```
cube :: Num a => a -> a
cube x = x * x * x
```

then sampling work would be multiplied by three, causing `a` to be evaluated twelve times for each sample time t.

One approach to solving the problem of redundant sampling is applying lazy memoization [16, 7], which may be supported with a function of the form type.

```
memo :: Eq a => (a -> b) -> (a -> b)
```

Semantically, `memo` is the identity. Operationally, the closure returned contains a mutable "memo table".

Any of the three representations discussed so far may be memoized. For instance, in Representation B one could simply memoize the constructed sampling functions. By way of example, here is an appropriately modified "`$*`". The only change is that the created `sample` function is memoized.

```
fb $* xb = Behavior (memo sample)
 where sample t = (f x, fb' $* xb')
         where (f, fb') = fb `at` t
               (x, xb') = xb `at` t
```

A drawback to this implementation is that the memoization overhead must be paid for each sample time of each component behavior, and so slows down sampling, rather than speeding it up, except in circumstances of extreme redundant sampling.

A more efficient alternative would to start with Representation C, so that memoization works at the level of *lists* of times rather than individual times. Rather than base memoization on the usual elementwise notion of list equality, which would be particularly problematic because our time lists are infinite, it suffices to use pointer equality on the list representations, as recommended in [16].

This is the representation used in the current version of Fran (1.11), except that memo tables are managed explicitly, rather than through a higher-order `memo` function. The reason for this exception is that memo tables need to be "aged", as will explained below.

Our algebra of behaviors is related to, and in some ways inspired by, Backus' language FP [3]. In FP, programs are always expressed at the function level, with application to "object-level" values kept implicit. This property leads to redundant applications of a function to the same argument, similar to the problem discussed in this section. The Illinois Functional Programming Interpreter [20] addressed this problem by using an "expression cache." For some recursive algorithms, expression caching reduced the asymptotic running time. Normally this caching had a more modest effect, speeding up some computations and slowing down others.

3.3 The Problem of Space-Leaking Memo Tables

A subtle but important consideration for any use of memoization is garbage collection. The memoized functions contain tables of domain and range values, and these tables typically grow without bound. When all references to a memoized function are lost, the contents of its memo table are reclaimed, as described in [7]. For example, consider the following behavior, which uses the `cube` and `interp` functions defined above to stretch a given picture. (The Fran type `ImageB` represents "image-valued behaviors", i.e., two-dimensional animations.)

```
anim1 :: ImageB
anim1 = stretch s1 pic
 where
   s1 = cube (interp 0.5 1.5 (cube time))
```

If `anim1` is displayed, it will be sampled with a time list that depends on the speed of the machine and the time-varying load from other processes. As long as there is no reference to `anim1` besides the one given to the display function, the representations of `anim1` and `s1`, including memo tables will be reclaimed. The list cells in the time and value lists will also be eligible for reclamation in an efficient manner as animation display progresses.

Unfortunately, memory use is not always so well-behaved. One problematic case is that in which the definition of `anim1` is a top-level definition, also known as a "CAF" (constant applicative form). In that case, in the current Haskell implementations we know of, the behaviors will never be reclaimed.[5]

Another problematic situation for reclamation of memo tables arises when a behavior is retained in order to be restarted, as in the following example, which

[5] An upcoming release of the Glasgow Haskell Compiler fixes this problem.

makes a looping behavior out of `s1`, restarting every two seconds.[6]

```
anim2 :: ImageB
anim2 = stretch s2 pic
 where
   s1 = cube (interp 0.5 1.5 (cube time))
   s2 = s1 `untilB` timeIs 2 -=> later 2 s2
```

Here we have used the `later` operator to shift `s2` by two seconds. (Alternatively, the semantics of `untilB` could do this kind of shifting automatically. Although there are some technical difficulties, we intend to alter Fran in this way. The implementation issue discussed below remains, however.) In this case, `s1` cannot released, because it will be used every two seconds to generate a new list of scale values. For each memo table, an entry is added every two seconds, and the evaluated size of each entry grows through lazy evaluation by a time, a scale, and two cons cells every sample, e.g., 10 per second.

With sufficient cleverness it may be possible to reuse the same cache entry each time `s1` restarts. Such reuse would only be correct, however, if the suffix of the sample time list following the event occurrence, shifted back by an amount equal to the event time is equal to the original sample time list. This condition would hold for example, if the original time list started at zero and contained every 1/10 second, and the event occurred exactly at a multiple of 1/10 second. If, however, the sampling time sequence contains any irregularity at any time in the future, or if the event were something like `timeIs pi`, then there could be no reuse. Both of these problematic situations are common in practice.

For these reasons, it seems necessary for memo tables to have special support from the garbage collector, so that when there are no live pointers to an object other than memo table keys, then the object gets reclaimed and the memo table entries get deleted. This idea is described in [16] and has been in use in some Smalltalk, Lisp and Scheme implementations for quite a while, but as far as we know it has not yet been implemented in a Haskell run-time system. Some Fran programs leak space for exactly this reason.

A more serious, and more subtle, problem with memoization comes from Fran's `afterE` combinator. This combinator is essentially a dual to `snapshot`. The two signatures are as follows:

```
snapshot :: Event a -> Behavior b -> Event (a, b)
afterE   :: GBehavior bv => Event a -> bv -> Event (a, bv)
```

The idea of ``e `snapshot` b`` is to pair up the event data from each occurrence of `e` with snapshots of `b` values at the corresponding occurrence time. Dually, ``e `afterE` b`` gives access to the *residual* of `b` at each occurrence of `e`. Its purpose is to be able to coordinate with a concurrently running behavior after an event

[6] The event "`e -=> v`" occurs whenever `e` occurs and has value `v` at each occurrence. Syntactically, it binds more tightly than `` `untilB` ``.

occurrence, but fortunately its use can often be hidden inside of other event and behavior combinators. Because `afterE` need not actually sample, it can be used with "generalized behaviors", a notions that subsumes the actual behavior types.

Fran implements `afterE` in terms of the following more primitive "aging" function:

```
afterTimes :: GBehavior bv => bv -> [Time] -> [bv]
```

The idea is to create a list of times corresponding not only to occurrences of an event, but to closely spaced *non-occurrences* as well. This list is fed into `afterTimes`, which then produces a stream of updated versions of its generalized behavior argument. (Note: it may well be possible and desirable to hide `afterE` beneath a set of more abstract combinators, but the implementation issues remain.)

Now consider the interaction between `afterTimes` and memoization. Suppose we have a behavior `bv` with a memo entry for the time stream $t_1, t_2, \ldots$ and corresponding value stream $x_1, x_2, \ldots$, and we have another time stream $t'_1, t'_2, \ldots$ as an argument to `afterTimes`. How should we initialize the aged behaviors' memo tables? If we use `bv`'s memo table, we will have a space leak, and, moreover, these entries are not likely to get cache hits. If, on the other hand, we start with empty memo tables, we will end up repeating a lot of work. A crucial observation is that these aged behaviors are often sampled with *aged sample time streams*, i.e., suffixes of the time streams that have already been used to sample `bv` itself. Rather than reusing or discarding memo tables in their entirety, the current Fran implementation "ages" the tables, replacing each time- and value-stream pair with corresponding stream suffixes. For instance, if $t_2 < t'_1 \leq t_3$, then the first aged memo pair would be $t_3, t_4, \ldots$ and $x_3, x_4, \ldots$.

Note that if we were to memoize Representation A or B instead of C, then it would become trickier to use a garbage collector to trim the memo tables. If the `Time` type is something like `Float` or `Double` (as in Fran's implementation), then we could easily keep a time/sample pair alive in a memo table by accident. If indeed these accidental retentions happen, a solution would be to introduce a data type to wrap around the time values.

3.4 The Problem of Synchrony

Memoization solves the problem of redundant sampling, but only when the different uses of a behavior are sampled with exactly the same time streams. There are, however, at least three situations in which we would want a behavior to be sampled with different time streams.

The first situation is the application of a time transformation. Consider the following example.

```
b :: RealB
```

```
b = b1 + timeTransform b1 (time / pi)
 where
   b1 = cube (interp 0.5 1.5 (cube time))
```

There is a second situation in which we might like to sample a behavior with two different rates, namely when different degrees of precision are needed. As an example, suppose we have an image `im1` moving in a complex motion path `mot`, with an overlaid, much smaller copy of itself:

```
im :: ImageB
im = stretch 0.1 im2 `over` im2
 where
   im2 = move mot im1
```

If we were sampling in order to stay within a error bound measured in 2D distance, rather than *temporal* rate, then the first use of `im2` would require less accuracy from `mot` and `im1` than the second use, because the spatial inaccuracies are reduced by a factor of ten.

A third situation calling for variable sampling rate is detection of events defined by boolean behaviors. As discussed in [11], interval analysis (IA) can be used in a robust nand efficient algorithm to detect occurrences of such events. The sampling patterns are adaptive to the nature of the condition's constituent behaviors.

In all of these cases, lack of synchrony disallows sharing of work between different sampling patterns. We do not have a solution to this problem. Note, however, that the values used for display in between samplings are necessarily approximate. (Fran uses a linearly interpolating engine [10].) As explored in Section 4.4 , this observation suggests sharing of work among non-synchronous sampling patterns.

3.5 Structural Optimizations

Fran's algebra of behaviors and events satisfies several algebraic properties that are exploited for optimization. Roughly speaking, these identities fall into the categories of "static" and "behavioral" properties.

By a "static" property, we mean one that is lifted to the level of behaviors directly from a property on static values. Examples include the following identity and distributive properties on images, where `over` is the (associative but not commutative) image overlay operation and `*%` applies a 2D transform to an image.

```
emptyImage `over` im  ==  im

im `over` emptyImage  ==  im

xf *% (im1 `over` im2)  ==  xf *% im1 `over` xf *% im2
```

Another useful property is following one for `condB`, which is the lifted form of if-then-else:

```
condB (constantB True ) b c  ==  b

condB (constantB False) b c  ==  c
```

By a "behavioral" property, we mean one that applies to behaviors over all types. For example, when "`$*`" is applied to constant behaviors, the result is a constant behavior, i.e.,

```
constantB f $* constantB x == constantB (f x)
```

As a consequence, a lifted n-ary function applied to n constant behaviors yields a constant behavior.

Another useful property is distributivity of lifted functions over reactivity. First consider a simple case.

```
lift1 f (b `untilB` e)  ==  lift1 f b `untilB` e ==> lift1 f
```

The event `e ==> h` occurs when `e` occurs, and its occurrence values result from applying the function `h` to the corresponding value from `e`. Syntactically, it binds more tightly than `` `untilB` ``.

Here is an obvious candidate for the general case:

```
fb $* (xb `untilB` e)  ==
  fb $* xb  `untilB` e ==> \ xb' ->
  fb $* xb'
```

and similarly for the case that `fb` is reactive. If both argument behaviors are reactive, then both rules may be applied sequentially (in either order). Similarly, if in the first rule, `xb` itself is reactive, applying the rule will give rise to another rule application as well.

There is, however, an operational problem with a rule like the one above, namely that it holds onto the behavior `fb` while waiting for the event `e` to occur, thus causing a space leak. Then when `e` finally occurs, `fb` will get sampled starting at the occurrence time. If `fb` has meanwhile undergone many transitions, there may be a lot of catching up to do, which amounts to a "time leak". To fix both of these problems, use `afterE` to give quick access to the residual of `fb`.

```
fb $* (xb `untilB` e)  ==
  fb  $* xb  `untilB` e `afterE` fb ==> \ (xb',fb') ->
  fb' $* xb'
```

The identities above, while widely applicable, do not improve performance by themselves. Their merit is that they often enable other optimizations. For instance, consider the following animation:

```
b :: RealB
b = (time `untilB` timeIs 5 -=> 0) + b2
```

Applying the `$*/untilB` identity yields the following:

```
time + b2 `untilB` (timeIs 5 -=> 0) `afterE` b2 ==>
   \ (b1', b2') -> b1' + b2'
```

When the transition occurs at time 5, the new behavior will be `0 + b2'`, which can be simplified to `b2'`.

4 Future Work

Although we have explored several alternatives for representing continuous animation, there are many more worth trying. Below are a few that we have given some preliminary thought to, but have implemented only partially or not at all.

4.1 Memo elimination

Consider again the examples from Section 3.2 that motivated memoization:

```
interp b1 b2 a  =  (1 - a) * b1 + a * b2

doubleInterp b1 b2 b3 b4 a = interp b1 b2 a'
  where a' = interp b3 b4 a

cube b = b * b * b
```

If a behavior is represented as a function from either times or time lists then the repeated occurrences of `a` in `interp` and `b` in `cube` will cause redundant sampling, unless some kind of run-time memoization is used. Unfortunately, the memoization has overhead even when it turns out not to be helpful, i.e., when a behavior is not used more than once. For example, the definition of `interp` creates three unshareable behaviors, namely `1-a`, `(1-a)*b1`, and `a*b2`, and only one shareable behavior (the one returned). (The three "unshareable" behaviors are in fact shareable indirectly, because of the containing shareable behavior. However, the containing behavior's memoization ensures that the contained behaviors are not redundantly sampled.)

In simple situations like `interp`, lack of shareability is apparent from the souce code. Given Fran's nature as a Haskell library (or "embedded language" [8]), however, we do not see how to exploit non-shareability to eliminate run-time memoization overhead. Unfortunately, the problem is especially messy, because the memoization cannot itself be implemented in pure Haskell, since it requires (a) hidden mutable state (not showing through in an imperative monad), and (b) pointer equality to preserve laziness.

4.2 Compile-Time Memoization

Just as non-shareability tends to be apparent in behavior source code, sharing does as well. Considering again the definition of `interp` above, it is easy to guess that `a` will be sampled redundantly. Could a Haskell compiler without hardwired domain knowledge eliminate redundant sampling, thus eliminating run-time overhead?

It seems possible for a domain-independent Haskell compiler to perform this optimization by applying some simple, generally useful transformations. The first step is to do some compile-time elimination of overloading, by specializing the definition of `interp` to the case of behavior arguments, e.g., using the techniques in [18].[7]

```
interp :: Num c => Behavior c -> Behavior c -> Behavior c
       -> Behavior c

interp b1 b2 a =
  lift2 (+) (lift2 (*) (lift2 (-) (constantB 1) a)
                       b1)
            (lift2 (*) a b2)
```

The next transformation is to unfold the definition of `lift2`, "`$*`", and `constantB` . The details depend on the exact representation of behaviors. For the sake of exposition, we will take Representation C from Section 3.1, based on stream functions. Following the unfolding by a few pattern match eliminations, would result in roughly the following:

```
interp b1 b2 a =
  Behavior (\ ts ->
   zipWith (+) (zipWith (*) (zipWith (-) (repeat 1) (a `at` ts))
                            (b1 `at` ts))
               (zipWith (*) (a `at` ts) (b2 `at` ts)) )
```

The crucial aspect of this transformation is that it revealed the redundant sampling of `a` over the time stream `ts`. The next transformation then is common-subexpression elimination (which is somewhat tricky for lazy languages [6]).

```
interp b1 b2 a =
  Behavior (\ ts ->
    let as = a `at` ts in
      zipWith (+) (zipWith (*) (zipWith (-) (repeat 1) as)
                               (b1 `at` ts))
                  (zipWith (*) as (b2 `at` ts)) )
```

[7]For simplicity, the derivation given assumes a somewhat different definition of `lift2` and `lift3`, directly in terms of `zipWith` and `zipWith3`, respectively.

Now the redundant sampling of `a` is gone, but another inefficiency has shown itself, namely the temporary creation and consumption of non-shareable intermediate lists. At this point, we would like to perform deforestation [24] to get something like the following.

```
interp b1 b2 a =
  Behavior (\ ts ->
    let b1s = b1 `at` ts
        b2s = b2 `at` ts
        as  = a  `at` ts
        f b1 b2 a = (1 - a) * b1 + a * b2
    in
        zipWith3 f b1s b2s as
```

General deforestation is a difficult optimization and is not present in compilers. The Glasgow Haskell Compiler [19] performs a simpler version that works in many cases [15], but not with functions like `zipWith` that consume two or more lists synchronously. Since the list functions `map`, `zipWith`, `zipWith3`, etc., are so commonplace in lazy functional programming, however, it seems reasonable to embed sufficient expertise about them into a compiler.

Note how much this final form of `interp` resembles the original form. With just a little more massaging, it becomes the following:

```
interp b1 b2 a =
  Behavior (\ ts -> zipWith3 f (b1 `at` ts)
                               (b2 `at` ts)
                               (a  `at` ts) )
 where
   f b1 b2 a = (1 - a) * b1 + a * b2
```

which is equivalent to a `lift3`'d version of the static version of `interp`:

```
interp = lift3 interpS
 where
   interpS b1 b2 a = (1 - a) * b1 + a * b2
```

This form could perhaps be arrived at by more direct means if a domain-independent compiler could be extended with domain-specific optimizations. In this case, the optimizations would involve various compositions of the lifting functionals.

Could the technique of "compile-time memoization", as we have just outlined, eliminate `all` redundant sampling? It appears not, since it depends on inlining definitions to see the repeated use of a behavior. In even moderately complex animation programs, the repeated use of behaviors can be hidden inside of several layers of definition. Inlining definitions exhaustively would cause unacceptable code bloat.[8] Perhaps the missing piece of the puzzle is some way for the compiler to factor each definition into a composition of an outer part to be inlined and an inner part to remain opaque. Ideally, the outer part would be small, but contain the crucial aspect of sampling. In the example of `interp`, the outer part might be all but the inner definition of `f`. (As always, the question remains: how could a domain-independent compiler choose the right factoring?)

The `interp` example is a fairly easy one, since it involves behaviors constructed solely through lifting (including `constantB`, which is zero-ary lifting). Reactivity and time transformation raise issues of their own.

4.3 Functions vs data structures

Arya did early work on temporally discrete modeled animation, representing animations simply as lists [1, 2]. Incremental value construction may be done as in our representation C. Moreover, a list representation does not have the problem of redundant sampling, because almost all of the computation work goes into the construction of the animation representation, rather than into sampling.

To simplify comparison of Arya's discrete model to our continuous one, here is how one would define some of the behavior constructors.

```
type Behavior a = [a]     -- possibly infinite lists

constantB = repeat        -- repeat x = [x, x, x, ...]

lift1 = map
lift2 = zipWith      -- map binary function over two lists
lift3 = zipWith3     -- map ternary function over three lists
-- etc
```

Time transformation is problematic. It would somehow have to estimate values in between the samples in a pre-time-transformed value list. For applicable types, one might try some form of interpolation, as in [2], but doing so is pure invention.

With the list representation the *logical* per-sample work is trivial – just extract and use the head and then continue with the tail. In contrast, the *operational* per-sample work is sometimes cheap and sometimes expensive; laziness delays the actual computation of the list elements until its first consumption, and other consumptions are virtually without cost.

In contrast with the list representation, in all of the continuous behavior implementations described above, the value computations logically belong to sampling, making sampling an expensive operation. Behavior construction, on the other hand, is logically very cheap, just wrapping up some other behavior

[8] We thank Simon Peyton Jones for pointing out this problem.

sampling functions in a new closure. Unfortunately, the caching done by a lazy language implementation only benefits the very cheap construction operation, but not the expensive sampling operation.

These observations about the operational differences between the list-based representation for discrete behaviors and the function-based representations for continuous behaviors suggest looking for a data structure for behaviors such that the sampling operation cost is a small fixed amount, independent of the complexity of a behavior's construction. Relative to the continuous behavior representations above, we must shift work from sampling to construction. Because of laziness, one consumer of a behavior will drive the actual computation, and the others will reap the benefits.

Important point: the sampling cost has to be *really* small, including a small constant factor.

4.4 Interpolation and Restricted Sample Time Streams

The function-based representations explored in Section 3 try to support the generation of sample values for arbitrary sample times. Of course, in general, behaviors are not actually constant in between these sample times. In the early implementations of Fran, however, as in most implementations of animation, behavior values were shown as constant until new values were computed. In the current implementation, behaviors are sampled at roughly ten times per second and then interpolated by a fast, specialized engine, running in its own thread [10]. The sprite engine performs image motion, stretching and compositing (overlaying) at roughly video refresh rate, which is indistinguishable from varying continuously. (Ideally, the rate would be exactly equal to the video refresh rate. The sprite engine can easily achieve this rate with visually simple animations, but not the desired regularity, due to operating system scheduling and locking issues.)[9]

Since we can rely on interpolation to fill in the gaps between computed behavior samples, we may relax our demands on behavior sampling. Instead of supporting arbitrary time streams, we might allow only a special class of them. In particular, we might allow only the streams S_k containing all integer multiples of 2^k, for each integer k, i.e., $S_k = \{n2^k \mid n \in \mathbb{Z}\}$. The benefit of such a restriction is that it allows us to represent a behavior not as a function, but as a data structure that contains all of the S_k. As pointed out in Section 4.3, the mechanics of lazy evaluation then work to our advantage. Computation of streams or portions of streams with no clients does not occur, while sharing among the various clients of a stream or portion happens automatically.

Now consider the question of an efficient representation of the set of value streams. In order to choose a representation, we must know what operations

[9]In the current implementation of Fran, when a behavior has a large discontinuity, interpolation is visually disturbing. For this reason, discontinuities should be communicated to the sprite engine explicitly so that it can be presented faithfully.

will be performed on it and with what frequency. These operations would seem to be the following, in order of decreasing frequency:

- interpolation between members of a single stream;
- extraction of successive members of a single stream;
- construction via lifted functions, time transformation, `untilB`, etc.; and
- shifting from one S_k to another, midway in enumeration.

One use of the last operation is that the frame rate at which an animation is presented may have to adapt to changes in the animation's complexity, or to changes in the load on a computer due to other processes. Another use comes from time transformation. For example, an animation transformed for "slow-in, slow-out" gets stretched near its middle, and so would need to be sampled at a higher rate there than near its beginning and end.

An obvious representation for the set of sample streams is a bidirectionally (doubly) infinite list of bidirectionally infinite lists. We would use doubly infinite lists because we need S_k for all integers k, both positive and negative, as well as both positive and negative multiples of 2^k.

```
type BiList   a = ([a],[a])    -- bidirectionally infinite list
type Behavior a = BiList (BiList a)
```

(If S_k contains only the *positive* multiplies of 2^k, then the inner `BiList a` may be simplified to `[a]`.)

A problem with this representation is that it is awkward to shift from one S_k to another, mid-stream. If we hold onto the other value lists from their beginning, then when it becomes necessary to shift, there may be a large amount of catching up to do, resulting in a space-time leak. Alternatively, one could advance in all of the value streams simultaneously. However, lazy evaluation, which is necessary since there are infinitely many value streams, would postpone their advancement, making the leak worse rather than better. Not only would the streams be retained, but also the cascading advancement computations.

4.5 Interval Trees

Another representation for the multi-resolution behaviors introduced above is an infinite binary tree. Each node represents a single interval with a single value and contains two subtrees with more refined approximations.

```
data ITree a =
  ITree Time DTime      -- Start time and duration
        a               -- value at start of interval
        (ITree a)       -- refinement for left half
        (ITree a)       -- refinement for right half
```

There is, however, a lot of redundancy in this representation, since the start times and durations of the subtrees are easily computed from those of the parent, and the left subtree's start value is the same that of its parent. We can remove all of this redundancy by moving the start time, duration and start value into a non-recursive data type, and augmenting the recursive one with a value at the interval's midpoint:

```
data ITree a = ITree Time DTime a (ITreeRec a)

data ITreeRec a =
  ITreeRec (ITreeRec a)     -- refinement for left half
           a                -- value at midpoint
           (ITreeRec a)     -- refinement for right half
```

Note that the `ITreeRec` type represents a behavior over a doubly-open interval, but does not tell a behavior's value at either the start or end of its interval (except in taking the limit of approaching midpoint values). The `ITree` type adds a start value, and so represents a behavior over a left-closed, right-open interval. An entire behavior, however, covers a left-closed, right-infinite interval. Fortunately, such an interval can be constructed by concatenating an infinite sequence of left-closed, right-open intervals, as long as (a) the start time of each member of the sequence equals the end time of the previous member, and (b) the endpoint sequence goes to infinity in the limit (i.e., the *series* of durations. This observation suggests a representation for behaviors:

```
-- Behavior as list of trees covering contiguous intervals.
newtype Behavior a = Behavior [ITree a]
```

There remains the criterion for dividing up the original semi-infinite interval. One possibility is a sequence of doubling durations, e.g., $[0,1)$, $[1,3)$, $[3,7)$, $[7,15)$, ... The benefit of this doubling approach is that it leads to a kind of logarithmic search "outwards", just as the approach of halving finite intervals leads to logarithmic search "inwards".

There is some redundancy in valid behavior representations. Because each `ITree` is followed by another, the durations of each may be computed by subtraction, and so may be removed:

```
data ITree a = ITree Time a (ITreeRec a)
```

Using interval analysis (IA) [22], one could choose an examination depth based on required precision. By combining symbolic differentiation and IA, one could then use second derivative bounds to determine error bounds on linear interpolations, borrowing results from the theory of piecewise approximation of curves [14].

The idea of multi-resolution representations based on binary trees has been used for images in computer graphics. In particular, a "MIP map" is a sequence of copies of a discrete image, each half the width and height of the predecessor, and stored in a clever manner for quick access [25]. In order to eliminate spatial aliasing (masquerading of high frequencies as low ones), each copy is prefiltered at the time of its construction, so that simple linear interpolation suffices for high quality display. Because aliasing is a problem for time as well as space, the same sort of filtering may be worthwhile in the interval tree representations discussed above. As far as we know, recursive representations like MIP maps have not been extended to infinitely large synthetic images. We intend to try such a representation for Fran images in the future.

4.6 Non-Reactive Normal Forms

As discussed in Section 3.5, lifted functions distribute over `untilB`. Repeatedly applying this distributivity property normalizes a reactive behavior to "non-reactive head normal form", which is `b ‘untilB‘ e`, where `b` is nonreactive (inserting the non-occuring event `neverE` if the behavior is non-reactive to begin with). Once the event `e` occurs, producing a new behavior, we again head-normalize it to get a new non-reactive behavior and another event.

This kind of normalization may be helpful, because it brings together larger compositions of non-reactive behaviors, which are easier to analyze. In particular, as described in [11], we can compose interval versions of the lifted functions making up the non-reactive head in order to perform efficient and robust detection of predicate events.

As pointed out in Section 3.1, in order to get the sampling incrementality we need, a good deal of behavior or list construction is done on each sampling. With a head-normalized behavior, this per-sampling construction would be unnecessary, since non-reactive behaviors can be sampled efficiently without incrementality.

We can take the idea of a non-reactive normal form even further. First note that in a reactive behavior `b1 ‘untilB‘ e1`, only the first occurrence of the event `e1` is relevant to the behavior's meaning. That occurrence has a time t_2 and a value, which in normal form is `b2 ‘untilB‘ e2`. Again, the only relevant aspect of `e2` is its first occurrence. Semantically, this list of event first occurrences constitutes an event in itself. This observation suggests another normal form, namely `switcher b e`, where `b` is non-reactive and `e` is an event over non-reactive behaviors. The Fran `switcher` function is currently implemented recursively in terms of `untilB`, as follows.[10]

```
-- Assemble a behavior piecewise from an initial one and an event
switcher :: GBehavior bv => bv -> Event bv -> bv
switcher b0 e = b0 ‘untilB‘ withRestE e ==> uncurry switcher
```

[10] The event `withRestE e` occurs when e does, and its value is value from `e`, paired with the residual of `e`. In the implementation of `switcher`, the new behavior and residual event are then passed recursively to `switcher`.

4.7 Constraint Propagation

Given non-reactive head normal form, we can then think of a reactive behavior as piece of state (a "constraint variable") holding only non-reactive behaviors. In this light, repeated head normalization is reminiscent of propagation of dependencies ("uni-directional constraints"). Scholz and Bokowski's PIDGETS++ system [21] works in this way. The TBAG system combined multi-directional constraint propagation with non-reactive behaviors [12]. The major difference between the approaches, aside from uni- vs multi-directionality, is inversion of control. In constraint propagation, changes are *pushed* toward constraint variables that depend on them (transitively). Head normalization, in contrast, *pulls* changes. It is probably more efficient to somehow use the push approach, or a hybrid of the two approaches, in the implementation of reactive behaviors.

In general, a stateful representation of reactive behaviors might be considerably more efficient than the stateless representations we have implemented. The main obstacle to pushing state chages is that behaviors must be time-transformable, which means different uses of the same behavior may require different states. A possible solution again comes from normalization. Time transformation distributes over lifted functions and reactivity. (Integration is more complex.) If there are enough of these distribution laws, then time transformation can be normalized away.

4.8 Translation

Fran has enough similarity to Esterel [4] and Lustre [5] to consider translation. Like Fran, these languages are based on a synchronous, deterministic model of concurrency. Esterel is imperative while Lustre is functional. Both have discrete notions of time, but it might be possible to translate a specification of concurrent behaviors into a parameterized specification of the result of sampling those behaviors over an arbitrary time stream (or snapshotting by an arbitrary event).

5 Conclusions

Modern software and hardware technology are temporally discrete in nature and so encourage discrete software models. In the context of animation, a continuous approach is more natural because it more closely reflects the real-world behaviors being modeled. In this paper, we have explored several functional implementations of continuous animation and some problems that arise. Some of these problems are rather subtle and became apparent only through costly trial and error. We have also considered many more ideas, some of which may turn out to be of practical value. We hope that their discussion here will motivate further work in pursuit of the goals of efficiently-executing, naturally-specified interactive animation.

6 Acknowledgements

Discussions with Sigbjorn Finne, Simon Peyton Jones, and Tony Daniels helped to clarify operational properties of the representations discussed in this paper. Extensive comments from anonymous reviewers greatly improved the presentation.

References

[1] Kavi Arya. A functional approach to animation. *Computer Graphics Forum*, 5(4):297–311, December 1986.

[2] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.

[3] John Backus. Can programming be liberated from the von Neumann style? *Comm. ACM*, 8:613–641, 1978.

[4] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[6] Olaf Chitil. Common subexpression elimination in a lazy functional language. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages*, St. Andrews, Scotland, September 10–12, 1997.

[7] Byron Cook and John Launchbury. Disposable memo functions (extended abstract). In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, page 310, Amsterdam, The Netherlands, June 1997.

[8] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *The Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997. USENIX. WWW version at `http://www.research.microsoft.com/~conal/papers/dsl97/dsl97.html`.

[9] Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, pages 18–33, July 1998. Expanded version with animated GIFs: `http://www.research.microsoft.com/~conal/fran/{tutorial.htm,tutorialArticle.zip}`.

[10] Conal Elliott. From functional animation to sprite-based display (expanded version). Technical Report MSR-TR-98-28, Microsoft Research, July 1998. `http://www.research.microsoft.com/~conal/papers/spritify-/long.ps`.

[11] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997.

[12] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida)*, pages 421–434. ACM Press, July 1994.

[13] John Peterson et. al. Haskell 1.3: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996. Current WWW version at `http://-haskell.org/report/index.html`.

[14] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3(4):295–311, 1986.

[15] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

[16] John Hughes. Lazy memo functions. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 129–146. Springer Verlag, September 1985.

[17] Adobe Systems Incorporated. *POSTSCRIPT Language: Tutorial and CookBook*. Addison-Wesley Publishing Company, Inc, 1991.

[18] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[19] Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, July 93.

[20] Arch D. Robinson. The Illinois functional programming interpreter. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 64–73. ACM, ACM, June 1987.

[21] Enno Scholz and Boris Bokowski. PIDGETS++ - a C++ framework unifying postscript pictures, gui objects, and lazy one-way constraints. In *Conference on the Technology of Object-Oriented Languages and Systems (TOOLS USA 96), Santa Barbara, California*. Prentice-Hall, 1996. `http://www.inf.fu-berlin.de/ scholz/tools-published.ps.gz`.

[22] John M. Snyder. Interval analysis for computer graphics. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 121–130, July 1992.

[23] Simon Thompson. A functional reactive animation of a lift using Fran. Technical Report 5-98, University of Kent, Computing Laboratory, May 1998. `http://www.cs.ukc.ac.uk/pubs/1998/583/index.html`.

[24] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[25] Lance Williams. Pyramidal parametrics. In *Proceedings of SIGGRAPH '83*, pages 1–11, July 1983.