FVision: A Declarative Language for Visual Tracking

John Peterson[1], Paul Hudak[1], Alastair Reid[2], and Greg Hager[3]

[1] Yale University, `peterson-john@cs.yale.edu` and `paul.hudak@yale.edu`
[2] University of Utah, `reid@cs.utah.edu`
[3] The Johns Hopkins University, `hager@cs.jhu.edu`

Abstract. Functional programming languages are not generally associated with computationally intensive tasks such as computer vision. We show that a declarative programming language like Haskell is effective for describing complex visual tracking systems. We have taken an existing C++ library for computer vision, called XVision, and used it to build FVision (pronounced "fission"), a library of Haskell types and functions that provides a high-level interface to the lower-level XVision code. Using functional abstractions, users of FVision can build and test new visual tracking systems rapidly and reliably. The use of Haskell does not degrade system performance: computations are dominated by low-level calculations expressed in C++ while the Haskell "glue code" has a negligible impact on performance.
FVision is built using functional reactive programming (FRP) to express interaction in a purely functional manner. The resulting system demonstrates the viability of mixed-language programming: visual tracking programs continue to spend most of their time executing low-level image-processing code, while Haskell's advanced features allow us to develop and test systems quickly and with confidence. In this paper, we demonstrate the use of Haskell and FRP to express many basic abstractions of visual tracking.

1 Introduction

Algorithms for processing dynamic imagery — video streams composed of a sequence of images — have reached a point where they can now be usefully employed in many applications. Prime examples include vision-driven animation, human-computer interfaces, and vision-guided robotic systems. However, despite rapid progress on the technological and scientific fronts, the fact is that software systems which incorporate vision algorithms are often quite difficult to develop and maintain. This is not for lack of computing power or underlying algorithms. Rather, it has to do with problems of *scaling* simple algorithms to address complex problems, *prototyping* and evaluating experimental systems, and effective *integration* of separate, complex, components into a working application.

There have been several recent attempts to build general-purpose image processing libraries, for example [9, 13, 8]. In particular, the Intel Vision Libraries[7] is an example of a significant software effort aimed at creating a general-purpose library of computer vision algorithms. Most of these efforts have taken the traditional approach of building object or subroutine libraries within languages such as C++ or Java. While these libraries have well designed interfaces and contain a large selection of vision data structures and algorithms, they tend not to provide language abstractions that facilitate dynamic vision.

The research discussed in this paper started with *XVision*, a large library of C++ code for visual tracking. XVision was designed using traditional object-oriented techniques. Although computationally efficient and engineered from the start for dynamic vision, the abstractions in XVision often failed to solve many basic software engineering problems. In particular, the original XVision often lacked the abstraction mechanisms necessary to integrate primitive vision components into larger systems, and it did not make it easy to *parameterize* vision algorithms in a way that promoted software reusability.

Rather than directly attacking these issues in the C++ world, we chose a different approach: namely, using *declarative programming* techniques. *FVision* is the result of our effort, a Haskell library that provides high-level abstractions for building complex visual trackers from the efficient low-level C++ code found in XVision. The resulting system combines the overall efficiency of C++ with the software engineering advantages of functional languages: flexibility, composability, modularity, abstraction, and safety.

This paper is organized as a short tour of our problem domain, punctuated by short examples of how to construct and use FVision abstractions. To put visual tracking into a more realistic context, some of our examples include animation code implemented in Fran, an animation system built using FRP[1]. Our primary goal is to explore issues of composibility: we will avoid discussion of the underlying primitive tracking algorithms and focus on methods for transforming and combining these primitive trackers. All of our examples are written in Haskell; we assume the reader is familiar with the basics of this language. See `haskell.org` for more further information about Haskell and functional programming. FRP is a library of types and functions written Haskell. The FRP library has been evolving rapidly; some function and type names used here may not match those in prior or future papers involving FRP. We do not assume prior experience with FRP in this paper. Further information regarding FRP can be found at `haskell.org/frp`.

2 Visual Tracking

Tracking is the inverse of animation. That is, animation maps a scene description onto a (much larger) array of pixels, while tracking maps the image onto a much simpler scene description. Animation is computationally more efficient when the scene changes only slightly from one frame to the next: instead of re-rendering the entire scene, a clever algorithm can reuse information from the previous frame and limit the amount of new rendering needed. Tracking works in a similar way: computationally efficient trackers exploit the fact that the scene changes only slightly from one frame to the next.

Consider an animation of two cubes moving under 3D transformations `t1` and `t2`. These transformations translate, scale, and rotate the cube into location within the scene. In Fran, the following program plays this animation:

```
scene :: Transform3B -> Transform3B -> GeometryB
scene t1 t2 = cube1 ‘unionG‘ cube2
 where cube1 = unitCube ‘transformG‘ t1
       cube2 = unitCube ‘transformG‘ t2
```

Rendering this animation is a process of generating a video (i.e. image stream) that is a composition of the videos of each cube, each of those in turn constructed from the individual transformations `t1` and `t2`. In computer vision we process the image stream to determine location and orientation of the two cubes, thus recovering the transformation parameters `t1` and `t2`.

We accomplish this task by using knowledge of the scene structure, as captured in a *model*, to combine visual tracking primitives and motion constraints into an “observer.” This observer processes the video input stream to determine the motion of the model. We assume that the behavior of objects in the video is somehow “smooth”: that is, objects do not jump suddenly to different locations in the scene. There are also a number of significant differences between vision and animation:

– Tracking is fundamentally uncertain: a feature is recognized with some measurable error. These error values can be used resolve conflicts between trackers: trackers that express certainty can “nudge” other less certain trackers toward their target.
– Efficient trackers are fundamentally history sensitive, carrying information from frame to frame. Animators generally hide this sort of optimization from the user.
– Animation builds a scene “top down”: complex objects are decomposed unambiguously into simpler objects. A tracker must proceed “bottom up” from basic features into a more complex object, a process which is far more open to ambiguity.

The entire XVision system consists of approximately 27,000 lines of C++ code. It includes generic interfaces to hardware components (video sources and displays), a large set of image processing tools, and a generic notion of a “trackable feature.” Using this as a basis, XVision also defines several *trackers*: specialized modules that recognize and follow specific features in the video image. XVision includes trackers for features such as edges, corners, reference images, and areas of known color. These basic tracking algorithms were re-expressed in Haskell using basic C++ functions imported via GreenCard, a tool for importing C code into Haskell.

2.1 Primitive Trackers

Primitive trackers usually maintain an underlying state. This state defines the location of the feature as well as additional status information such as a confidence measure. The form of the location is specific to each sort of tracker. For a color blob it is the area and center; for a line it is the two endpoints.

Figure 1 illustrates this idea conceptually for the specific case of an SSD (Sum of Squared Differences) tracking algorithm [2]. This algorithm tracks a region by attempting to compute an image motion and/or deformation to match the current appearance of a target to a fixed reference. The steps in the algorithm are:

1. Acquire an image region from the video input using the most recent estimate of target position and/or configuration. In addition, reverse transform (warp) it. The acquired region of interest is generally much smaller than the full video frame. Pixels are possibly interpolated during warping to account for rotation or stretching.
2. Compute the difference between this image and the reference image (the target).
3. Determine what perturbation to the current state parameters would cause the (transformed) current image to best match the reference.
4. Use this data to update the running state.

As this process requires only a small part of the original video frame it is very efficient compared to techniques that search an entire image. It also makes use of the fact that motion from frame to frame is small when computing the perturbation to the current state. As a consequence, it requires that the target move relatively consistently between frames in the image stream: an abrupt movement may cause the tracker to lose its target.

In XVision, trackers can be assembled into *hierarchical constraint networks* defined by geometric knowledge of the object being tracked (the model). This knowledge is typically a

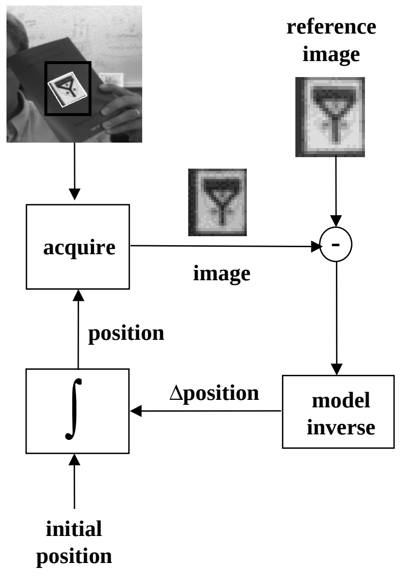

Fig. 1. Figure of XVision feedback loop.

relationship between different points or edges in the object's image, such as the four corners of a square. If one corner is missing in the image (perhaps due to occlusion) then the positions of the other three define the expected location of the missing corner. This allows a disoriented tracker to resynchronize with its target. Although XVision includes object-oriented abstractions for the construction of hierarchical constraint networks, these abstractions had proven difficult to implement and limited in expressiveness. In the remainder of the paper we describe a rich set of abstractions for tracker composition.

3 Abstractions for Visual Tracking

A camera converts a continuously changing scene into a discrete stream of images. In previous work we have defined trackers in terms of standard *stream processing* combinators [11]. Here these combinators are subsumed by FRP. FRP supports inter-operation between continuous time systems and discrete time (stream processing) systems. This allows FVision to combine with animation systems such as Fran or robotics systems such as Frob[10].

Before examining the construction of trackers, we start by demonstrating the use of a tracker in conjunction with animation. This function processes a video stream, of type `Video clk`, and generates an animation in which a red dot is drawn over a tracked image. The type `Video` is defined thusly:

```
type Video clk = CEvent clk Image
```

The `CEvent clk a` in FRP denotes a stream of values, each of type `a`, synchronized to clock `clk`. This clock type allows FVision to detect unintentional clock mismatches. Since none of the code in this paper is tied to a specific clock the `clk` argument to `CEvent` will always be uninstantiated.

The user must first define the reference image to be tracked by using the mouse to select a rectangular area around the target in the video image. The rectangular area is marked by pressing the mouse to indicate the top-left corner, then dragging and releasing the mouse at the bottom-right corner. As the mouse is being dragged, an animated rectangle is drawn over the video image. Once the mouse is released, the rectangle is replaced by a red dot centered on the rectangle and an SSD tracker is created to move the dot through successive frames.

```
followMe :: Video clk -> PictureB
followMe video =
  videoB `untilB`
   (lbp `snapshot_` mouse) ==>
     \corner1 -> rectangle (lift0 corner1) mouse `over` videoB
       `untilB`
       ((lbr `snapshot_` (pairB mouse videoB) ==>
         \(corner2, image) ->
          let tracker = ssdTracker (getImage image corner1 corner2)
              mid = midPoint corner1 corner2
              b = runTrackerB videoB mid mid tracker
          in (redDot `transform2B` tracker)
    `over` videoB ))
 where videoB = stepper nullImage video  -- convert image stream to behavior
       redDot = ...           -- draw a red dot
       rectangle c1 c2 = ... -- draw a rectangle
```

The above code can be read: "Behave as the video input until the left mouse button is pressed, at which time a snapshot of the mouse position is taken. Then draw a rectangle whose top left-hand corner is fixed but whose bottom right-hand corner is whatever the current mouse position is. Do this until the left mouse button is released, at which point a snapshot of both the mouse position and the video are taken. A tracker is initialized with the midpoint of the two corners as the initial location and the snapshot image as the reference. Use the output of the tracker to control the position of a red dot drawn over the video image." For example, if you draw a rectangle around a face the tracker can the follow this face as it moves around in the camera's field of view. This tracker is not robust: it may lose the face, at which point the red dot will cease to move meaningfully.

Functions such as `untilB` and `snapshot_` are part of FRP. By convention, types and functions suffixed with "B" deal with behaviors: objects that vary continuously with time. Type synonyms are used to abbreviate `Behavior Picture` as `PictureB`. Some functions are imported from XVision: the `getImage` function extracts a rectangular sub-image from the video stream. This image serves as a reference image for the SSD (Sum Squared Difference) tracker. Once the reference image is acquired, the tracker (the `ssdTracker` function) defines a behavior that follows the location of the reference image in the video stream. The `runTrackerB` function starts the tracker, pointing it initially to the selected rectangle, defining the transformation used in the animation.

3.1 Types for Tracking

The goal of this research is to define trackers in a compositional style. Following the principals of type directed design, we start with some type definitions. A tracker is composed of two parts: an *observer* which acquires and normalizes some subsection of the video image, and a *stepper* which examines this sub-image and computes the motion of the tracker.

```
type Observer observation a = (a, Image) -> observation
type Stepper measure observation a = (a, observation) -> measure a
```

The observer takes the present location, `a`, of the tracker and the current frame of video and returns an observation: usually one or more sub-images of the frame. The location may be designated using a single point (the `Point2` type), as used in color blob tracking, or perhaps by a point, rotation, and scale (represented by the `Transform2` type). Observers may also choose sample at lower resolutions, dropping every other pixel for example. In any case, the type `a` is determined by the particular observer used.

The stepper adjusts the location of the tracked feature based on the current location and the observation returned by the stepper. The stepper may also compute additional values that measure accuracy or other properties of the tracker. We choose to make the measure a type constructor, `measure a`, rather than a separate value, `(measure, a)`, so as to use overloading to combine measured values. XVision defines a variety of steppers, including the SSD stepper, color blob steppers, edge detectors, and motion detectors.

Measurement types are defined to be instances of the `Valued` class. This extracts the value from its containing measurement type:

```
class Valued c where
  valueOf :: c a -> a
```

Measurement types are also in the `Functor` class, allowing modification the contained value.

The `Residual` type used by the SSD tracker in an example of a measurement:

```
data Residual a =
  Residual { a :: resValue, residual :: Float }

instance Valued Residual where
  valueOf = resValue
```

Combining an observer and a stepper yields a tracker: a mapping from a video stream onto a stream of measured locations.

```
type Tracker measure a = Stepper measure Image a
```

Note that `Tracker` is a refinement of the `Stepper` type. Trackers are constructed by combining an observer with a stepper:

```
mkTracker ::  Observer observation a -> Stepper measure observation a ->
              Tracker measure a
mkTracker o s = \(loc, image) -> let ob = o (loc, image) in s (loc, ob)
```

3.2 A Primitive Tracker

We can now assemble a primitive FVision tracker, the SSD tracker. Given a reference image, the observer pulls in a similar sized image from the video source at the current location. The stepper then compares the image from the current frame with the reference, returning a new location and a residual. This particular tracker uses a very simple location: a 2-D point and an orientation. The SSD observer is an XVision primitive:

```
grabTransform2 :: Size -> Observer Image Transform2
```

where `Size` is a type defining the rectangular image size (in pixels) of the reference image. The position and orientation of the designated area, as defined in the `Transform2`, are used to interpolate pixels from the video frame into an image of the correct size.

The other component of SSD is the stepper: a function that compares a reference image with the observed and determines the new location of the image. The type of the stepper is

```
ssdStep :: Image -> Stepper Residual Image Transform2
```

where `Image` argument is the reference image. A detailed description of this particular stepper is found in [11]. Now for the full SSD tracker:

```
ssdTracker :: Image -> STracker Residual Transform2
ssdTracker image =
 mkTracker (grabTransform2 (sizeOf image)) (ssdStep image)
```

Before we can use a tracker, we need a function that binds a tracker to a video source and initial location:

```
runTracker :: Valued measure =>
   Video clk -> a -> Tracker measure a -> CEvent clk a
runTracker video a0 tracker = ma where
 locations  = delay a0 aStream
  ma        = lift2 (,) locations video ==> tracker
  aStream   = ma ==> valueOf
```

The `delay` function delays the values of an event stream by one clock cycle, returning an initial value, here `a0`, on the first clock tick.

We can also run a tracker to create a continuous behavior, `Behavior b`.

```
runTrackerB :: Valued measure =>
   Video clk -> measure a -> Tracker measure a -> CEvent clk a
runTrackerB video ma0 trk =
  stepper ma0 (runTracker video (valueOf ma0) trk)
```

In this function, we need a measured initial state rather than an unmeasured one since the initial value of the behavior is measured.

The `clk` in the type of `runTracker` is not of use in these small examples but is essential to the integrity of multi-rate systems. For example, consider an animation driven by two separate video sources:

```
scene :: Video clk1 -> Video clk2 -> PictureB
```

The type system ensures that the synchronous parts of the system, trackers clocked by either of the video sources, are used consistently: no synchronous operation may combine streams with different clock rates. By converting the clocked streams to behaviors, we can use both video sources to drive the resulting animation.

3.3 More Complex Trackers

Consider an animator that switches between two different images:

```
scene :: Transform2B -> BoolB -> PictureB
scene place which = transform2 place (ifB which picture1 picture2)
```

A tracker for this scene must recover both the location of the picture, `place` (a 2-D transformation) and the boolean that selects the picture, `which`. Previously, we inverted the transformation for a fixed picture. Here we must also invert the `ifB` function to determine the state of the boolean. We also we wish to retain the same compositional program style used by the animator: our tracking function should have a structure similar to this `scene` function.

The composite tracker must watch for both images, `picture1` and `picture2` at all times. To determine which image is present, we examine the residual produced by SSD, a measure of the overall difference between the tracked image and the reference image. We formalize this notion of "best match" using the `Ord` class:

```
instance Ord (Residual a) where
  r1 > r2 = residual r1 < residual r2
```

This states that smaller residuals are better than large ones.

The `bestOf` function combines a pair of trackers into a tracker that follows whichever produces a better measure. The trackers share a common location: in the original scene description, there is only one transformation even though there are two pictures. The resulting values are augmented by a boolean indicating which of the two underlying trackers is best correlated with the present image. This value corresponds to the `which` of the animator. The projection of the measured values onto the tracker output type are ignored: this combines the internal tracker states instead of the observed values seen from outside.

```
bestOf :: (Functor measure, Ord measure) =>
  Tracker measure a -> Tracker measure a -> Tracker measure (a, Bool)
bestOf t1 t2 =
 \((loc, _), v) -> max (fmap (\x -> (x, True)) (t1 (loc, v)))
                       (fmap (\x -> (x, False)) (t2 (loc, v)))
```

The structure of `bestOf` is simple: the location (minus the additional boolean) is passed to both tracker functions. The results are combined using `max`. The `fmap` functions are used to tag the locations, exposing which of the two images is presently on target.

This same code can be used on steppers as well as trackers; only the signature restricts `bestOf` to use trackers. This signature is also valid:

```
bestOf :: (Functor measure, Ord measure) =>
  Stepper measure observation a ->  Stepper measure observation a ->
  Stepper measure observation (a, Bool)
```

Thus steppers are composable in the same manner as trackers. This is quite useful: by composing steppers rather than trackers we perform one observation instead of two. Thus the user can define a more efficient tracker when combining trackers with a common observation.

Higher-order functions are a natural way to express this sort of abstraction in FVision. In C++ this sort of abstraction is more cumbersome: closures (used to hold partially applied functions) must be defined and built manually.

3.4 Adding Prediction

We may to improve tracking accuracy by incorporating better location prediction into the system. When tracking a moving object we can use a linear approximation of motion to more accurately predict object position in the next frame. A prediction function has this general form:

```
type Predictor a = Behavior (Time -> a)
```

That is, at a time t the predictor defines a function on times greater than t based on observations occurring before t.

Adding a predictor to `runTracker` is simple:

```
runTrackerPred :: Valued measure =>
   Video clk -> Tracker measure a -> Predictor a -> CEvent clk a
runTrackerPred video tracker p =
  withTimeE video `snapshot` p
      ==> \((v,t), predictor) -> tracker (predictor t, v)
```

The FRP primitive `withTimeE` adds an explicit time to each frame of the video. Then `snapshot`, another FRP primitive, samples the predictor at the current time and adds sampled values of the prediction function to the stream.

This is quite different from `runTracker`; there seems to be no connection from output of the tracker back to the input for the next step. The feedback loop is now outside the tracker, expressed by the predictor.

Using prediction, a tracking system looks like this:

```
followImage :: Video clk -> Image -> Point2 -> CEvent clk Point2
followImage video i p0 =
  let ssd = ssdTracker i
      p = interp2 p0 positions
      positions = runTrackerPred video p ssd

interp2 :: Point2 -> CEvent clk Point2 -> Predictor Point2
```

The `interp2` function implements simple linear prediction. The first argument is the initial prediction seen before the initial interpolation point arrives. This initial value allows the `p0` passed to `interp2` to serve as the initial observed location.

4 Generalized Composite Trackers

We have already demonstrated one way to compose trackers: `bestOf`. Here, we explore a number of more general compositions.

4.1 Trackers in Parallel

An object in an animation may contain many trackable features. These features do not move independently: their locations are related to each other in some way. Consider the following function for animating a square:

```
scene :: Transform2B -> PictureB
scene t = transform2 t (polygon [(0,0), (0,1), (1,1), (1,0)])
```

In the resulting animation, trackers can discern four different line segments - one for each edge of the square. The positions of these line segments are somewhat correlated: opposite edges remain in parallel after transformation. Thus we have a level of redundancy in the trackable features. Our goal is to exploit this redundancy to make our tracking system more robust by utilizing relationships among tracked objects.

A composite tracker combines trackers for individual object features into a tracker for the overall object. The relationship between the object and its features is represented using a pair of functions: a *projection* and an *embedding*. These functions map between the model state (parameters defining the overall object) and the states of the component trackers. The projection function maps a model state onto a set of component states and the embedding function combines the component states into a model state. This function pair is denoted by the following type:

```
type EPair a b = (a -> b, b -> a)
```

We now build a composite tracker that combines the states of two component trackers. In this example, we define a corner tracker using two component edge trackers. Edge trackers are implemented using the following XVision stepper:

```
edgeStepper :: Stepper Sharpness Image LineSeg
```

The location maintained by the tracker is a line segment, denoted by the `LineSeg` type. This tracker observes an `Image` and measures the quality of tracking with the `Sharpness` type. This `Sharpness` type has the same structure as the `Residual` type but is mathematically distinct. To combine two line segments into a corner, we find the intersection of the underlying lines (possibly outside the line segments) and then "nudge" the line segment to this point. This is crucial since the edge trackers tend to creep away from the corner. The underlying geometric types are as follows:

```
type LineSeg = (Point2, Vector2)
type Corner = (Point2, Vector2, Vector2)
```

We force the length of the vector defining a line segment to remain constant during tracking, allowing the use of a fixed size window on the underlying video stream. The projection and embedding functions are thus:

```
cornerToSegs :: Corner -> (LineSeg, LineSeg)
cornerToSegs (corner, v1, v2) = ((corner, v1), (corner, v2))

segsToCorner :: (LineSeg, LineSeg)  -> Corner
segsToCorner (seg1@(_,v1), seg2@(,v2)) = (segIntersect seg1 seg2, v1, v2)
```

Next we need a function to combine two trackers using a projection / embedding pair.

```
join2 :: (Joinable measure, Functor Measure) =>
  Tracker measure a -> Tracker measure b -> EPair (a,b) c -> Tracker measure c
join2 t1 t2 (fromTup, toTup) =
  \(c, v) -> let (a,b) = toTup c
                 ma = t1 (a,v)
                 mb = t2 (b,v)
              in fmap fromTup (joinTup2 (ma, mb))
```

The structure of this function is the same as the `bestOf` function defined earlier. There is a significant addition though: the type class `Joinable`. Here we create a measured object from more than one measured sub-objects. Thus we must combine the measurements of the sub-objects to produce an overall measurement. The `Joinable` class captures this idea:

```
class Joinable l where
 joinTup2 :: (l a,l b) -> l (a, b)
 joinTup3 :: (l a,l b,l c) -> l (a, b, c) -- and so on

instance Joinable Sharpness where ...
```

The joinTup2 function joins two measured values into a single one, combining the measurements in some appropriate way. Joining measurements in a systematic manner is difficult; we will avoid addressing this problem and omit instances of Joinable.

Another way to implement joining is to allow the embedding function to see the underlying measurements and return a potentially different sort of measurement:

```
join2m :: Tracker measure a -> Tracker measure b ->
          ((measure a, measure b) -> measure2 c, c -> (a, b)) ->
          Tracker measure2 c
```

This can be further generalized to allow all of the component trackers to use different measurements. However, in most cases we can hide the details of joining measured values within a type class and spare the user this extra complexity.

Now for the corner tracker:

```
trackCorner :: Tracker Sharpness LineSeg -> Tracker Sharpness LineSeg ->
               Tracker Sharpness Corner
trackCorner l1 l2 = join2 l1 l2 (segsToCorner, cornerToSegs)
```

The join2 function is part of a family of joining functions, each integrating some specific number of underlying trackers.

The corner tracker incorporates "crosstalk" between the states of two trackers but does not have to deal with redundant information. We now return to tracking a transformed square. Given four of these corner trackers, we now compose them into a square tracker. The underlying datatype for a square is similar to the corner:

```
type Square = (Point2, Point2, Point2)
```

We need specify only three points; the fourth is functionally dependent on the other three. This type defines the image of a square under affine transformation: from this image we can reconstruct the transformation (location, rotation, scaling, and shear). Our problem now is to map four tracked corners onto the three points defining the Square type. There are many possibilities: for example, we could throw out the point whose edges (the two vectors associated with the corner) point the least towards the other corners, probably indicating that the corner tracker is lost. Here, we present a strategy based on the Sharpness measure coming from the underlying trackers.

The only significant difference between the previous example and this one is in the embedding function. We need to combine measured values in the embedding; thus the tracker is defined using join4m. First, we need to lift a function into the domain of measured values. Using the Joinable class we define the following:

```
jLift3 :: Joinable m => (a -> b -> c -> d) -> (m a -> m b -> m c -> m d)
jLift3 f = \x y z -> let t = joinTup3 x y z in
                       fmap (\(x',y',z') -> f x' y' z') t
```

Using this, we build a function that generates a measured square from three measured points:

```
mkSquare :: Sharpness Point2 -> Sharpness Point2 -> Sharpness Point2) ->
            Sharpness Square
mkSquare = jLift3 (\x y z -> (x,y,z))
```

Now we generate all possible squares defined by the corners, each using three of the four edge points, and choose the one with the best Sharpness measure using max:

```
bestSquare :: (Sharpness Point2, Sharpness Point2, Sharpness Point2,
               Sharpness Point2) -> Sharpness Square
bestSquare (v1, v2, v3, v4) =
  mkSquare v1 v2 v3 `max` mkSquare v1 v2 v4 `max`
  mkSquare v1 v3 v4 `max` mkSquare v2 v3 v4
```

In summary, the family of `join` functions capture the basic structure of the parallel tracker composition. While this strategy occasionally requires somewhat complex embedding functions this is exactly where the underlying domain is also complex. Also, we can use overloading to express simple embedding strategies in a concise and readable way.

4.2 Trackers in Series

Another basic strategy for combining trackers is combine slow but robust "wide field" trackers with fast but fragile "narrow-field" trackers to yield an efficient robust tracking network. The structure of this type of tracker does not correspond to an animator since this deals with performance rather than expressiveness. Switching between different trackers is governed by measures that determine whether the tracker is "on feature" or not. Consider the following three trackers:

– A motion detector that locates areas of motion in the full frame.
– A color blob tracker that follows regions of similarly colored pixels.
– A SSD tracker targeted at a specific image.

Our goal is to combine these trackers to follow a specific face with an unknown initial location. The motion detector finds an area of movement. In this area, the blob tracker finds a group of flesh-colored pixels. Finally, this blob is matched against the reference image. Each of these trackers suppresses the one immediately proceeding it: if the SSD tracker is "on feature" there is no need for the other trackers to run.

The type signatures of these trackers are relatively simple:

```
motionDetect :: Tracker SizeAndPlace ()
blob         :: Color -> Tracker SizedAndOriented Point2
ssd          :: Image -> Tracker Residual Transform2
```

The `motionDetect` tracker is an example of a *stateless* tracker. That is, it does not carry information from frame to frame. Instead, it looks at the entire frame (actually a sparse covering of the entire frame) at each time step. Since there is no location to feed to the next step, all of the information coming out of `motionDetect` is in the measure. For the blob tracker we get both a size and an orientation, the axis that minimizes distance to the points.

To compose trackers in series, we use a pair of state projection functions. This is similar to the embedding pairs used earlier except that there is an extra `Maybe` in the types:

```
type SProjection m1 a1 m2 a2 = (m1 a1 -> Maybe s2, m2 a2 -> Maybe s1)
```

These functions lead up and down a ladder of trackers. At every step, if in the lower state we go "up" if the current tracker can produce an acceptable state for the next higher tracker. If we are in the higher state, we drop down if the current tracker is not in a suitable situation.

The tracker types reflect the union of the underlying tracker set. To handle measures, we need a higher order version of `Either`:

```
data EitherT t1 t2 a = LeftT (t1 a) | RightT (t2 a)

instance (Valued t1, Valued t2) => Valued (EitherT t1 t2) where
  valueOf (LeftT x) = valueOf x
  valueOf (RightT x) = valueOf x
```

Now we combine two trackers in series:

```
tower :: Tracker m1 a1 -> Tracker m2 a2 -> SProjection m1 a1 m2 a2 ->
         Tracker (EitherT m1 m2) (Either a1 a2)
tower low high (up, down) =
  \(a, v) -> case a of
     Left a1 -> let ma1 = low (a1, v) in
      case up ma1 of
        Nothing -> LeftT (fmap Left ma1)
        Just a2 ->
          let ma2 = high (a2, v) in
             case down ma2 of
              Nothing -> RightT (fmap Right ma2)
              Just _  -> LeftT (fmap Left ma1)
     Right a2 -> let ma2 = high (a2, v) in
       case down ma2 of
        Nothing -> RightT (fmap Right ma2)
        Just a1  -> LeftT (fmap Left (low ma1))
```

This calls each of the sub-trackers no more than once per time step. The invariants here are that we always attempt to climb higher if in the lower state and that we never return a value in the higher state if the `down` function rejects it.

Before using the `tower` function, we must construct the state projections. Without showing actual code, they function as follows:

– Move from `motionDetect` to `blob` whenever the size of the area in motion is greater than some threshold (normally set fairly small). Use the center of the area in motion at the initial state in the blob tracker.
– Always try to move from blob to SSD. Use the blob size and orientation to create the initial transformation for the SSD tracker state.
– Drop from `ssd` to `blob` when the residual is greater than some threshold. Use the position in the transformation an the initial state for `blob`.
– Drop from `blob` to `motionDetect` when the group of flesh-toned pixels is too small.

The composite tracker has the following structure:

```
faceTrack :: Image ->
  Tracker (EitherT (EitherT SizeAndPlace SizedAndOriented) Residual)
          (Either (Either () Point2) Transform2)
faceTrack image =
  tower (tower motionDetect blob (upFromMD, downFromBlob))
        ssd onlyRight (upFromBlob, downFromSSD)
 where
   upFromMD mt =
      if mArea mt > mdThreshold then Just mCenter mt else Nothing
   downFromBlob mt =
      if blobSize mt < bThreshold then Just (blobCenter mt) else Nothing
  upFromBlob mt =
     Just (translate2 (blobCenter mt)
             `compose2`
           rotate2 (blobOrientation mt))
  downFromSSD mt =
    if residual mt > ssdthreshold
        then Just (origin2 `transform2` (valueOf mt))
        else Nothing
```

The `onlyRight` function is needed because the composition of motion detection and blob tracking yields an `Either` type instead of a `blob` type. The `onlyRight` function (not shown) keeps the ssd tracker from pulling on the underlying tracker when it is looking for motion rather than at a blob.

The output of this tracker would normally be filtered to remove states from the "backup" trackers. That is, the ultimate result of this tracker would probably be `Behavior (Maybe Point2)` rather that `Behavior Point2`. Thus when the composite tracker is hunting for a face rather than on the face this will be reflected in the output of the tracker.

5 Performance

Programs written in FVision tend to run at least 90% as fast as the native C++ code, even though they are being run by a Haskell interpreter. This can be attributed to the fact that the bottleneck in vision processing programs is not in the high-level algorithms, as implemented in Haskell, but in the low-level image processing algorithms written in C++. As a result, we have found that FVision is a realistic alternative to C++ for prototyping or even delivering applications. While there are, no doubt, situations in which the performance of Haskell code may require migration to C++ for efficiency, it is often the case that the use of a declarative language to express high-level organization of a vision system has no appreciable impact on performance. Furthermore, the Haskell interpreter used in our experiment, Hugs, has a very small footprint and can be included in an application without seriously increasing the overall size of vision library.

6 Related Work

We are not aware of any other efforts to create a declarative language for computer vision, although there does exist a DSL for writing video device drivers [12] which is at a lower level than that this work.

There are many on tools for building domain-specific languages such as FVision from scratch, but most relevant are previous efforts of our own on *embedded* DSL's [5, 4] that use an existing declarative language as the basic framework. General discussions of the advantages of programming with pure functions are also quite numerous; two of particular relevance to our work are one using functional languages for rapid prototyping [3] and one that describes the power of higher-order functions and lazy evaluation as the "glue" needed for modular programming[6].

7 Conclusions

FVision has proven to be a powerful software engineering tool that increases productivity and flexibility in the design of systems using visual tracking. As compared to XVision, the original C++ library, FVision reveals the essential structure of tracking algorithms much more clearly.

Some of the lessons learned in this project include:

1. Visual tracking offers fertile ground for the deployment of declarative programming technology. The underlying problems are so difficult that the payoff in this domain is very high. FVision is significantly better for prototyping tracking-based applications than the original XVision system.
2. The process creating FVision uncovered interesting insights that were not previously apparent even to original XVision developers. Working from the "bottom up" to develop a new language forces the domain specialists to examine (or re-examine) the underlying domain for the right abstractions and interfaces.
3. The principal features of Haskell, a rich polymorphic type system and higher-order functions, were a significant advantage in FVision.
4. FRP provides a rich framework for inter-operation among the various system components. By casting trackers in terms of behaviors and events we were able to integrate them smoothly into other systems.

This work was supported by NSF grant CCR-9706747 in experimental software systems.

References

[1] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[2] G. D. Hager and P. N. Belhumeur. Efficient region tracking of with parametric models of illumination and geometry. To appear in IEEE PAMI., October 1998.

[3] P. Henderson. Functional programming, formal sepcification, and rapid prototyping. *IEEE Transactions on SW Engineering*, SE-12(2):241–250, 1986.

[4] P. Hudak. Building domain specific embedded languages. *ACM Computing Surveys*, 28A:(electronic), December 1996.

[5] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[6] J. Hughes. Why functional programming matters. Technical Report 16, Programming Methodology Group, Chalmers University of Technology, November 1984.

[7] Intel vision libraries. `http://developer.intel.com/research/mrl/research/cvlib/`.

[8] R.E. Kahn, M.J. Swain, P.N. Prokopowicz, and R.J. Firby. Gesture recognition using Perseus architecture. In *Proc. IEEE Conf. Comp. Vision and Patt. Recog.*, pages 734–741, 1996.

[9] J.L. Mundy. The image understanding environment program. *IEEE EXPERT*, 10(6):64–73, December 1995.

[10] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of PADL 99: Practical Aspects of Declarative Languages*, pages 91–105, Jan 1999.

[11] A. Reid, J. Peterson, P. Hudak, and G. Hager. Prototyping real-time vision systems. In *Proceedings of ICSE 99: Intl. Conf. on Software Engineering*, May 1999.

[12] C. Consel S. Thibault, R. Marlet. A domain-specific language for video device drivers: From design to implementation. In *Proceedings of the first conference on Domain-Specific Languages*, pages 11–26. USENIX, October 1997.

[13] The Khoros Group. *The Khoros Users Manual.* The University of New Mexico, Albuquerque, NM, 1991.