Submitted to the 1999 International Conference on Software Engineering

Prototyping Real-Time Vision Systems: An Experiment in DSL Design

Alastair Reid, John Peterson, Greg Hager, Paul Hudak
Yale University
P.O Box 208285
New Haven, CT 06520
{reid-alastair, peterson-john, hager-greg, hudak-paul}@cs.yale.edu

ABSTRACT
We describe the enhancement of XVision, a large library of C++ code for real-time vision processing, into FVision (pronounced "fission"), a fully-featured domain-specific language embedded in Haskell. The resulting prototype system substantiates the claims of increased modularity, effective code reuse, and rapid prototyping that characterize the DSL approach to system design. It also illustrates the need for judicious interface design: relegating computationally expensive tasks to XVision (pre-existing C++ components), and leaving modular compositional tasks to FVision (Haskell). At the same time, our experience demonstrates how Haskell's advanced language features (specifically parametric polymorphism, lazy evaluation, higher order functions and automatic storage reclamation) permit a rapid DSL design that is itself highly modular and easily modified. Overall, the resulting hybrid system exceeded our expectations: visual tracking programs continue to spend most of their time executing low level image-processing code, while Haskell's advanced features allow us to quickly develop and test small prototype systems within a matter of a few days and to develop realistic applications within a few weeks.

Keywords
Domain-specific languages, Functional programming, Modularity, Code reuse, Computer vision, Haskell.

1 INTRODUCTION

Real-time computer vision is an area that is at a critical juncture. Inexpensive cameras, digitizers, and high-performance video devices are now plentiful, and the processing power of most PC's and workstations has reached the point where they can perform many image processing functions which historically required specialized hardware [7]. Software exploiting vision has not, however, advanced at a comparable rate. We assert that this is not due to a lack of algorithms or computing power, but rather that little is yet known about the effective software abstractions and tools in this domain.

There have been several attempts to build general-purpose image processing libraries [14, 18, 13]. Most have taken a traditional approach to system design using a language such as C++ or Java to build suitable libraries, based on well designed interfaces, that capture system functionality in a modular way. *XVision* is such a library designed for a specialized subset of real-time computer vision tasks, in particular real-time tracking. The interfaces were designed with the usual trade-offs of performance and functionality, and many successful vision applications

Yet even with XVision, building an application is not always easy. There is often a need for better composition and abstraction facilities than exist in the current version. Furthermore, programmer productivity is a particular problem: the best way of constructing a particular vision system is often found only by extensive prototyping, combining elements from a variety of techniques. Thus, the traditional programming/debugging/testing cycle is quite long. It is further hampered by the fact that it is often difficult to determine whether a given system malfunction is due to a programming error, or a conceptual problem with the underlying methodology.

This has led us to investigate *domain-specific languages* (DSLs) as a way of augmenting our existing libraries with the composition and abstraction mechanisms needed in this field, and to give us stronger guarantees about program correctness. With the DSL approach, a special purpose language is developed to provide just the right glue and abstraction mechanisms to make composition and parameterization easy and natural for the domain of interest.

In this paper we describe our experiences designing and implementing a DSL called *FVision*, using XVision as the source of primitive operations. Although designing and implementing a DSL can itself be a difficult task (language design is difficult!), we avoided this problem by building FVision as an *embedded* DSL in the functional language *Haskell*. FVision programs are perfectly valid Haskell programs, but certain syntactic, static, and dynamic language features in Haskell give FVision the look and feel of an entirely new language.

Just the process alone of designing FVision clarified what the primitive operations should be, and resulted in a stream-lining of the XVision libraries to yield only its essence. Our work shows that constructing an effective DSL from an existing library is not a matter of simply "turning the crank," but rather requires a significant re-engineering effort to achieve an effective domain-specific language. The proven benefits of the DSL, though, make this effort worthwhile. These benefits include:

- Flexibility. The ability to quickly experiment with and evaluate a large variety of solutions is a necessary process when building complex vision-based systems.
- Modularity and abstraction. Programming abstractions are designed which are natural to the domain, but which are not feasible in the current technology of Java or C++ programming. The resulting clarity and compactness makes explicit various ideas that are left implicit in most vision systems, and facilitates the description of the underlying algorithms in a concise and semantically clear fashion.
- Efficiency. The low-level operations which dominate execution time remain in the C++ domain. The FVision glue is usually not a significant part of the execution time.
- Safety. The FVision type system ensures that modules are composed correctly; we avoid using dynamic typing or other techniques that may fail at execution time.

This paper provides an overview of the FVision approach to computer vision and compares it with XVision. We address techniques used to embed a DSL in Haskell, including transforming monolithic C++ components into highly parameterized, purely functional Haskell objects. To achieve this we rely critically on Haskell's parametric polymorphism, lazy evaluation, higher order functions, type classes, and garbage collection. Finally, we evaluate our approach, comparing program development using FVision to that using XVision.

2 THE DOMAIN: REAL-TIME VISION

XVision is an application and hardware-independent set of tools for visual feature tracking. Conceptually, XVision can be viewed as the inverse of an animation system. Whereas in animation the goal is to quickly combine a set of graphics primitives into an "animator" that produces a desired video output stream, in XVision the goal is to combine visual tracking primitives and motion constraints into an an "observer" for a video input stream.

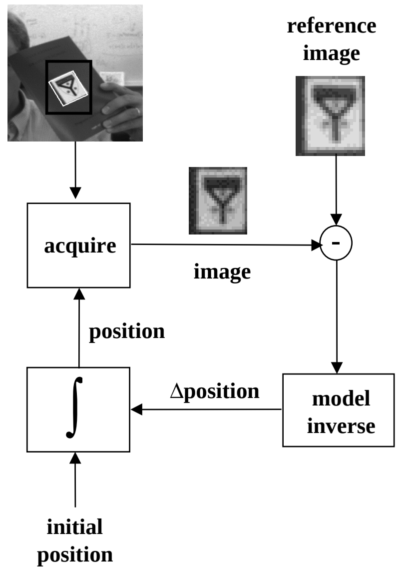

Figure 2: Figure of XVision feedback loop.

The existing XVision system libraries consist of approximately 27,000 lines of C++ code organized as shown in Figure 1. In particular, XVision defines generic interfaces to hardware components (video sources and displays), contains a large set of image processing tools, and defines a generic notion of a "trackable feature." Using this as a basis, XVision then defines several *trackers*: specialized modules that recognize and follow specific features in the video image. XVision includes trackers that follow the position of a line, corner, an area of color, and a variety of other similar image artifacts.

The existing XVision system is organized around two programming abstractions. The first abstraction is to view each feature as a *state-based* object and to define tracking as a feedback process on the state of the feature. The state, which usually consists of the location of the feature plus some additional status information, consolidates and defines the information content of each feature as a single consistent entity. The notion of the feedback loop is important as it captures the idea that the state of the feature at the current time is in fact a small perturbation on what it was in the previous frame.

Figure 2 illustrates this idea conceptually for the specific case of an SSD (Sum of Squared Difference) tracking al-

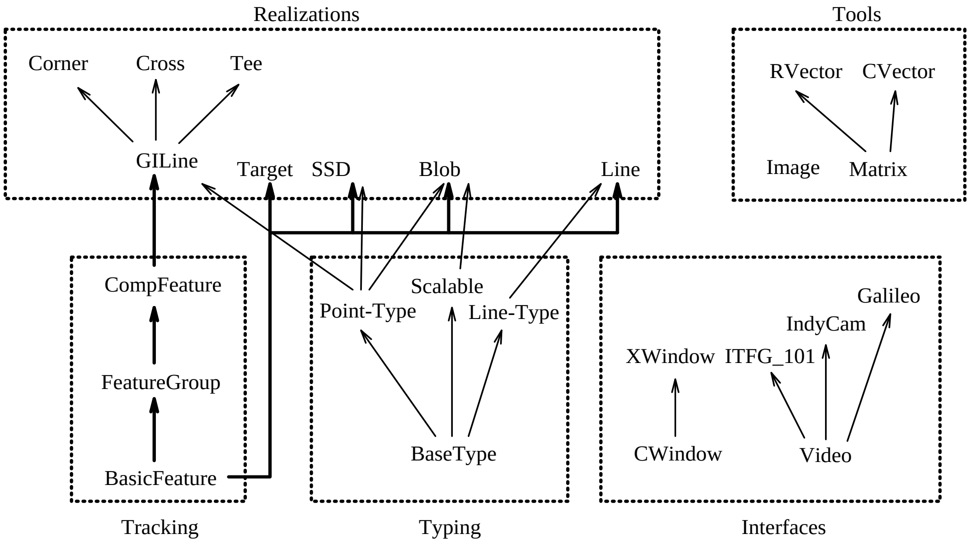

Figure 1: Software Layout.

gorithm [6]. 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 and deform an image based on the previous state, (2) compute the difference between this image and the reference image (the target), and (3) perform some arithmetic to determine what perturbation to the current state parameters would cause the (deformed) current image to best match the reference.

The second major abstraction in XVision is that of combining simple features to form more complex tracking systems, which result in *hierarchical constraint networks*. Figure 3 shows the feature network for a clown face animation using SSD trackers as its input. At the image level, SSD tracking primitives operating on images localize the eyes and mouth. For each eye and mouth, there are in fact two trackers, one for an open eye (or mouth) and one for a closed one. Status information (essentially how well the reference image matches the current image) determine the status (open or closed) of their respective targets. Subsequent levels of the system combine this information into a consistent representation of the pose and status of the face. The animation (frames of which are shown at the right of Figure 3) results by "slaving" graphics drawing primitives to the state of each tracking primitive.

3 FROM XVISION TO FVISION

Despite the success of XVision as a substrate for application development, it was clear that the existing design was often too inflexible for the type of experimental programming involved in developing vision-based systems. Thus, our initial plan was to simply import XVision tracking primitives as DSL components and to capture only one of the XVision abstractions, hierarchical composition, with the DSL. At this stage, we started to replicate the XVision C++ object hierarchy in Haskell. This fell short in a number of ways:

- The use of subclassing to extend existing classes is difficult to replicate outside of the C++ type system.
- The original C++ code made extensive use of implicit object state. This led to code which could not take advantage of Haskell's purely functional nature.
- The C++ classes were very course-grained: the structure of the underlying algorithms was hidden inside the classes. This prevented experimentation with the structure of the algorithms.
- The C++ objects did not take advantage of Haskell's polymorphic type system.

It quickly became clear to us that we should instead recreate the basic structure of the XVision trackers directly in Haskell instead of importing the entire tracker as a highly complex, indivisible black box. In our second attempt, we imported the non-tracking-specific core components of XVision—namely the interfaces to the outside world and the image processing tools—and recreated in Haskell much of what had been completely encapsulated, monolithic object definitions in C++. In particular, we could now easily capture the core abstraction of a tracking cycle within the DSL, and thereby

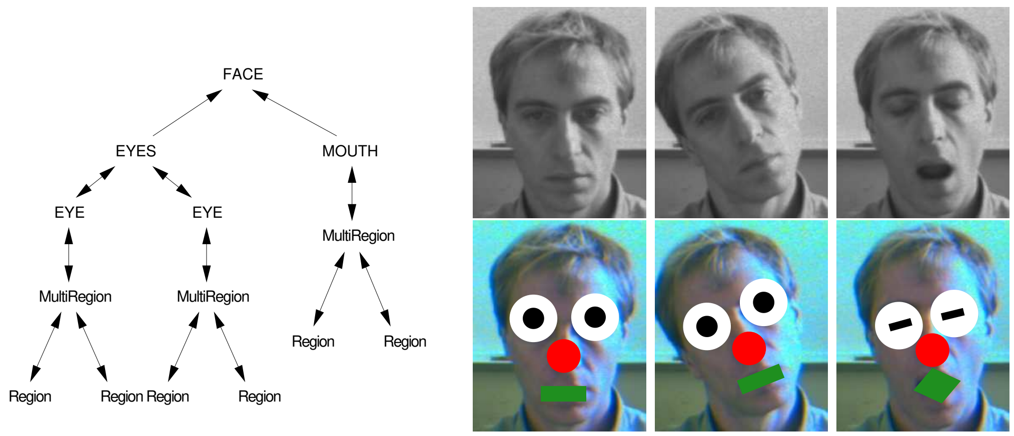

Figure 3: (Left) The tracking network used for face tracking as it is defined in XVision. (Right) The output of the "clown face" tracker. The upper row of images shows the raw video, and the lower row of images shows the graphics produced by the tracker overlaid onto the live video.

experiment with new tracking algorithms by using programming abstractions even at that level.

Another improvement in this second effort was to replace the feedback loops previously hidden within the trackers by an abstraction defining a sequential set of values; i.e. a *pipeline*. This pipeline abstraction served as a basis for the translation into a much more idiomatic and useful DSL version of the trackers.

To illustrate the flavor of our approach, we describe below four key parts of the FVision system in detail: pipelines, the *SSD stepper*, the *SSD tracker* and the *clown face* demo described in the previous section. Although space limitations preclude the explanation of every syntactic detail, we feel that the examples in most cases are self-explanatory, which is some indication of the naturalness of the DSL design.

Pipelines

Pipelines provide a *declarative* view of the iterative loops used in XVision. Specifically, they allow the definition of iterative networks of computation based on pure functions that operate on pipelines. These are functions in the mathematical sense of the word: they have no state; the result of function application does not depend on how the function has been used in the past. Pure functions are an essential feature of FVision and offer many advantages in a DSL framework:

- System specifications, including those for computer vision, are often described in mathematical terms. Translating those specifications into a DSL that resembles the domain-specific mathematics is thus relatively easy.
- An equally important specification method is *flow diagrams*, such as used in signal processing, but these too are stateless. Any given flow diagram (even ones with loops) can be converted easily into a set of mutually recursive FVision equations (indeed, they are isomorphic).
- Reasoning about, analyzing, and transforming programs is generally easier for programs that do not rely on global state.
- Understanding components based on pure functions is easier since their interface to the rest of the program is explicit rather than implicit.

Furthermore, it is often the case that pipelines are conceptually *infinite* in length: the Haskell substrate on which FVision is built easily supports this through *lazy evaluation*. Infinite pipelines are quite common in FVision programming, but the user need not worry about problems with termination.

In FVision, the type of a "pipeline" containing values of type `T` is written `Pipe T`. For example, the type `Pipe Float` denotes a pipeline of floating point numbers and `Pipe Image` denotes a pipeline of images. Note that the type constructor `Pipe` is *polymorphic*: that is, each pipeline can contain a different type of value.

A simple pipeline is written `pipe [x,y,z]` where `x`, `y`, and `z` are the elements in the pipeline. The elements must all have the same type, but otherwise may be images, floating-point numbers, or whatever.

FVision also supplies a rich set of functions for the construction, combination, and de-structuring of pipelines. Indeed, it is often the case that we have a function that operates on images or floating-point numbers, say, and we wish to "lift" it to operate on *pipelines* of images or floating-point numbers. This is one place where polymorphic higher-order functions really shine: instead of redefining these functions to operate on the pipelines, we simply provide a family of polymorphic "lifting operators" to do this for us:

```
pipe0 :: a              -> Pipe a
pipe1 :: (a -> b)       -> (Pipe a -> Pipe b)
pipe2 :: (a -> b -> c) ->
                (Pipe a -> Pipe b -> Pipe c)
...
```

`pipe0` takes a function of 0 arguments (aka a constant) and turns it into a "constant pipeline" which always contains the same value; `pipe1` takes a function of 1 argument and turns it into a function which takes a pipeline of arguments and returns a pipeline of results; `pipe2` takes a function of 2 arguments and turns it into a function which takes 2 pipelines of arguments and returns a pipeline of results; etc. For example, `pipe2 (*)` creates a version of the multiplication function (written as `(*)` in FVision) which operates on two pipelines of numbers, yielding a pipeline of products:

```
(*) 2 3    ==>    6
pipe2 (*) (pipe [2,3,4]) (pipe [3,4,5])
           ==> pipe [6,12,20]
pipe2 (*) (pipe0 2) (pipe [3,4,5])
           ==> pipe [6,8,10]
```

Pipelines may be split or joined using these functions:

```
joinPipe  :: Pipe a -> Pipe b -> Pipe (a, b)
splitPipe :: Pipe (a, b) -> (Pipe a, Pipe b)
```

For example:

```
joinPipe (pipe [1,2,3]) (pipe [4,5,6])
   ==> pipe [(1,4),(2,5),(3,6)]
splitPipe (pipe [(1,4),(2,5),(3,6)])
   ==> (pipe [1,2,3], pipe [4,5,6])
```

Another way to combine pipelines is to "multiplex" them: the `multiplex` function merges two pipes, using a third pipe as a switch; in essence this is just the conditional `if` function "lifted" into the pipeline domain:

```
multiplex ::
    Pipe Bool -> Pipe a -> Pipe a -> Pipe a
multiplex = pipe3 cond
 where
  cond x y z = if x then y else z
```

Pipelines may also include interactions with the outside world. IO actions in FVision have type `IO a` for some type `a`. The family of functions:

```
pipeIO0 :: IO a         -> Pipe a
pipeIO1 :: (a -> IO b) -> (Pipe a -> Pipe b)
...
```

yield pipelines which execute an IO action at each iteration of the pipeline. For example, `acquire v sz pos` is an IO action which acquires an image of size `sz` at position `pos` from a video device `v`; therefore `pipeIO1 (acquire v sz)` is a function which acquires a sequence of images (of fixed size) at a sequence of different positions in the video frame.

Feedback loops often require a delay to hold a value from one time step to the next. The `delay` function delays the values in the pipeline by one step, using a given initial value for the first element in the pipe:

```
delay :: a -> Pipe a -> Pipe a
```

Feedback is how state is expressed in control loops, and thus this delay function is how we express stateful trackers. For example, if the tracker applies a function `step` to generate a new state from the old on each iteration, then the following function yields an iterated tracker:

```
iterate ::
    (a -> a -> a) -> a -> Pipe a -> Pipe a
iterate combine x0 xs =
  let p = delay x0 (pipe2 combine p xs)
  in p
```

Note that the pipeline `p` is defined recursively; indeed, it is an example of an infinite pipe.

A particularly useful function built using `iterate` is `integral` which computes the running total of the values in its input pipe:

```
integral :: (Num a) => a -> Pipe a -> Pipe a
integral x0 xs = iterate (+) x0 xs
```

In summary, the pipeline abstraction takes advantage of many Haskell features: polymorphic typing (in the `Pipe` type and the `pipe<n>` functions), higher-order functions (in the `pipe<n>` functions and `iterate` function) and lazy evaluation (in the `multiplex`, `iterate` and `integral` functions).

The SSD Stepper

The SSD tracker follows a reference image as it moves in the video stream. In XVision, this tracker is defined using a complex object structure containing many different methods and internal state. The inner structure of SSD contains a loop that acquires a region within the image, compares it with the reference image, and then adjusts the apparent location of the image to account for movement that has taken place since the previous frame.

In FVision, the SSD tracker is broken into two parts: the *stepper* and the *tracker*. The stepper is a pure function whose FVision code is given in figure 4. This code is a direct transcription of the SSD algorithm into FVision, and is readable to anyone familiar with the underlying algorithm, but is otherwise unimportant here. More important is the type signature, which declares that `ssdStep` takes two inputs, both images (the reference image is the first parameter), and returns a delta (direction to move the "camera" to adjust the current image to match the reference image) and the residual (an estimate of the closeness of the match between the area under the camera and the reference).

The other component of SSD is the tracker. Once we implemented the pipeline abstraction, it proved to be trivial to implement an SSD tracker, as shown in figure 5. All we had to do was translate the flow diagram in figure 2 from the graphical syntax into a textual syntax using the pipeline abstraction to represent the lines in the diagram. (The translation is not quite exact: the stepper incorporates both the image difference and model inverse steps of Figure 2.) Once again, the type signature reveals the basic operation of `ssdTrack`: given a video stream, the initial position of the tracked feature, and an image of the tracked feature, this function returns two pipelines: a sequence of points and a sequence of residuals.

The cyclic dependencies in the tracking algorithm (as expressed in the flow diagram) are directly reflected in the dependencies between the variables in the `let` expression (in FVision, the definitions introduced in a `let` expression are mutually recursive). Lazy evaluation ensures that at each step evaluation will occur in the proper order; i.e. by demand: first the image is acquired from the video device at the current position, then the SSD stepper computes a delta from the current position, and then this delta is added to the current position. The `integral` function serves the essential role of delaying each computation by one step. That is, it uses the delta computed in the previous iteration to compute the present value of the integral.

More Complex Trackers

To show the compositional nature of FVision, and thus its ability to scale, a more complex tracker based on SSD is shown in figure 6. This tracker is used as part of the clown face program mentioned earlier. It tracks eye position using two different reference images: an open eye and a closed eye. The tracker compares the current image with reference images for both an open and a closed eye, choosing to move the tracker using the delta associated with the image with the smallest residual (error value) associated with it.

This tracker fuses the results of two SSD sub-trackers, one for each image. Both trackers share a common state (the current position) and the image is continuously compared against both reference images, using the image most closely matching the current image to guide the tracker. The result includes a pipeline of booleans, indicating which of the two images is currently being tracked.

We can further abstract this fusion by replacing the image parameters with arbitrary trackers. Thus any tracker returning a delta and residual can be combined with a similar tracker to yield the composite tracker. Higher-order functions are a natural way to express this sort of abstraction in FVision. In C++ this sort of abstraction is much more cumbersome: closures (used to hold partially applied functions, such as `ssdStep openIm`) must be defined and built manually.

4 IMPLEMENTATION ISSUES

We hope that the previous section provides convincing evidence that FVision is a valuable DSL for computer vision. In this section we address some of the practical issues involved in designing and implementing FVision as an embedded DSL in Haskell, and connecting it to a large C++ library, XVision.

The Domain Vocabulary

A key part of building a domain-specific language for vision processing is providing the basic vocabulary necessary to express operations in the domain efficiently and recognizably. This consists of the primitive data types that domain experts want to use (principally, in our case, images and matrices) and a useful set of operations to create, manipulate, and output values of those types. An essential tool in this task was our foreign function interface generator *GreenCard* [16] which made it easy to make C++ values and operations look like Haskell values and operations to the FVision programmer.

Partly from habit and partly because we were anticipating the pipeline abstraction, we wanted the domain-specific operations to be "pure" (free from, and unaffected by, side effects) and "lazy" (performed only if and when required). Making the XVision library look this way required a little work:

- Although many operations were already pure, C++ programmers rarely document such facts. Thus, it became necessary to become familiar with the implementation of as well as the interfaces to the

```
ssdStep :: Image -> Image -> (Delta,Double)
ssdStep refIm r =
  let m         = vectorsToMatrix [smoothDx refIm, smoothDy refIm]
      m_t       = transpose m
      m'        = inverse (m_t * m) * m_t
      error     = imageToVector (compressXY (refIm - r))
      delta     = m' `multVector` error
      residual  = norm (error - m `multVector` delta)
  in (matrixToDelta delta, residual)
```

Figure 4: The SSD stepper

```
ssdTrack :: Video -> Pos -> Image -> (Pipe Pos, Pipe Double)
ssdTrack video initialPosition refIm =
  let
    image             = pipeIO1 (acquire video (sizeOf refIm)) position
    (delta, residual) = splitPipe (pipe1 (ssdStep refIm) image)
    position          = integral initialPosition delta
  in (position, residual)
```

Figure 5: The SSD tracker

```
eye :: Video -> Pos -> Image -> Image -> (Pipe Pos, Pipe Bool)
eye video initialPosition openIm closedIm =
  let
    image                         = pipeIO1 (acquire video (sizeOf openIm)) posn
    (openDelta,   openResidual)   = splitPipe (pipe1 (ssdStep openIm)   image)
    (closedDelta, closedResidual) = splitPipe (pipe1 (ssdStep closedIm) image)
    isOpen                        = pipe2 (<) openDelta closedDelta
    delta                         = multiplex isOpen openDelta closedDelta
    posn                          = integral initialPosition delta
  in (posn, isOpen)
```

Figure 6: The eye tracker

underlying image processing library.

- Although it is easy to make pure operations lazy, it becomes almost impossible to reason about the lifetime of objects created by lazy operations, and thus manual storage management (using C++'s `new` and `delete`) is infeasible. Fortunately, GreenCard provides mechanisms that let Haskell's garbage collector take over the task of managing C++ objects from the programmer: when a C++ object is returned to Haskell from a C++ function, it is added to a list of objects managed by Haskell; and when Haskell no longer requires an object it is managing, it calls `delete` to release the object.

However, we did not try to make all operations pure, since some operations such as acquiring an image or drawing an image on the screen are necessarily "impure": purity and laziness are merely *design guidelines* and it is not helpful to be too dogmatic about them.

We also discovered the need for many standard image/matrix operations which C++ programmers would just code "inline." For example, one of our trackers needs to compute an image mask by applying a threshold to an image. Though this is a common enough, general purpose operation often used in XVision applications, XVision does not provide a function to do this directly. One way to add this function would be to add operations for manipulating individual pixels in an image and then code the function in Haskell; another way to add this function would be to code it in C++ and add it to XVision. We chose the second way for two reasons:

1. Crossing the boundary from Haskell into C++ is relatively expensive (perhaps the cost of 10 or 20 function calls in C++) so, for efficiency reasons, we wanted to avoid this boundary crossing overhead on tight loops.

2. Most of the new image processing functions we wanted to add were general purpose functions that we thought XVision *ought* to provide. It seemed that the best way to avoid reinventing the wheel was to make sure that everyone knew where the wheels were stored: in the XVision library.

In this way the rigid separation between the domain-specific language and the domain-specific operations helped clarify what operations we needed.

The final factor that influenced the design process grew from the dynamics of the project, a collaboration between vision researchers and functional programming researchers. As naïve users, the functional programmers would try to do things that didn't make any sense in the computer vision world, and then complained to the vision researchers when they obtained strange results. For example, adding two color images doesn't make any sense since the pixels are represented by 32 bit numbers: any overflow in one color field "spills over" into another color field resulting in strangely colored images. These problems prompted the development of a more precise type system for images that keeps color images separate from gray-scale images. This type system was quickly prototyped in Haskell by giving functions types which were more restrictive than in C++. Having found that this type system catches many trivial errors but doesn't interfere with the programmer too much, we plan to express the type system directly in the C++ class hierarchy.

After our failed first attempt at importing XVision into Haskell, one thing we *did not* try to do was to import XVision's high level abstractions into Haskell. Reasons for this include:

- Since our goal was to *redesign* XVision's high level abstractions, we did not want to buy into the existing abstractions for fear that they would make it hard to invent more appropriate abstractions or would make it harder to prototype new ideas.

- The low level objects and operations have simple, well understood, *obvious* interfaces; whereas the high level objects and operations have much more complex interfaces. It just wasn't obvious what the essence of the high level objects should be.

- The high level objects make more use of the C++ class hierarchy. This could probably be mimicked in Haskell, but it wasn't obvious how. Nor was it obvious that the existing class hierarchy was the *Right Design* rather than just what was convenient to code in C++.

Virtual Cameras and Displays

The only reason we provide the `pipeIO<n>` functions in the pipeline library is to let us acquire images from the video devices. These functions were a relatively late addition to the library: in earlier versions of FVision, opening a video device returned a pipe of images and `ssdTrack` used a `subImage` operation to acquire a small part of that image. We found this version much simpler to reason about but were forced to abandon it because of severe performance problems.

Our video device drivers run in the operating system kernel and capture images into one of a small number of memory buffers shared with the (user mode) program. Since there are only a few shared memory buffers, we have to copy the image into unshared memory before putting it into the pipe. Since the video device generates 30 frames per second of 1.5Mbytes each, our early applications spent most of their time copying memory around. This was particularly galling since a typical image processing application only examines a few regions of perhaps 1kbyte each.

Our solution was to add the `pipeIO<n>` functions and the `acquire` functions so that we could view a single *physical* camera (represented by a pointer to a C++ object) as a number of separate *virtual* cameras each providing a pipe of *subimages* of the full camera image.

Similarly, we plan to add support to let us view each *physical* window on our desktop as a collection of *virtual* windows each displaying relevant images and data from inside an FVision pipeline. This should be easy to do using standard functional programming technology [5] and will solve a problem observed in both FVision and XVision: when a complex application starts, it typically opens a dozen small windows on the screen, each window being randomly positioned on the screen according to the window manager's whim.

5 ASSESSMENT

In many ways, the development of FVision has been an experiment in software engineering and DSL design which has exceeded our expectations in terms of scope, performance, simplicity and usability. In the remainder of this section we discuss each of these issues in turn.

Performance

One of the reasons we had initially chosen to interface to XVision at a high level was our belief that Haskell would force us to pay too high a performance cost. It turns out that this supposition was unfounded. In fact, we have found that programs written in FVision run at least 90% as fast as the native C++ code, even though they are (currently) being run interpreted. This (surprising!) discovery can be attributed to the fact that the bottleneck in vision processing programs is not in the high-level algorithms (which we prototyped in Haskell) but in the low-level image processing algorithms (which we imported from C++). As a result, we have found that FVision is in fact 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 Haskell 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 application without seriously increasing the overall size of vision library.

Choice of Scope

As noted above, our original goals were to incorporate much of the "high-level" XVision code and to use it as a "black box." In retrospect, the initial attempts at doing this required as much or more effort than moving down a level and developing the complete XVision system itself within the DSL. This can largely be attributed to the fact that the lower-level operations of XVision not only fit well within the Haskell programming paradigm, but they also have simpler interfaces which were straightforward to incorporate.

Programmer Productivity

Enlarging the scope of the DSL had an additional advantage of allowing us to us to quickly explore the design space for visual tracking as well as the implications of the design far faster than a C++ prototype would have.

The Pipe library is a good example. A simplistic prototype of pipes had been developed for XVision. It consisted of 2400 lines of C++ code written over several months. The code was designed as an "add on" to the existing XVision, although it was hoped that the use of data-flow processing would eventually impact the development of other aspects of the system.

Designing the Pipe library in FVision took only two days of programmer time, and consists of just 200 lines of FVision code. This can be attributed largely to the ability to describe pipes as lazy lists, and the use of polymorphism to import basic image operations into pipes. Furthermore, we could now explore the implications of pipes for the remainder of the system (in particular SSD) completely within the DSL. As a result, we now have a much more concrete notion of how a redesigned XVision (based on the pipe abstraction) would look.

Flexibility and Useability

One of the problems we have found with XVision over the past few years is that, although the software abstractions work reasonably well, the domain of real-time vision resists simple encapsulation. Most of the tracking methods can in fact be "tuned" or modified in a large variety of ways. Making all of the possibilities available via a generic interface across the various modalities has proven cumbersome (and was one reason why our initial prototype was more difficult to construct).

The DSL, in particular the development of pipes, has not only clarified much of the design of the system, but it has also made it easier to expose the inner workings of individual algorithms. As a result, the composition of new tracking systems is much simpler.

Finally, the pipeline model is a good basis for parallel execution on shared memory multiprocessors or even on a loosely coupled collection of processors.

6 RELATED WORK

We are not aware of any other efforts to create a special-purpose language for computer vision, although there does exist a DSL for writing video device drivers [17]. That work is at a level quite a bit lower than that at which we are working, but it is conceivable to use it as a substrate for our own work.

There are papers too numerous to mention on tools for building DSL's from scratch, but most relevant is previous efforts of our own on *embedded* DSL's [11, 10]. Previous examples of DSL's embedded in Haskell include Fran [3, 4], a language for functional reactive animations, and ActiveHaskell [15], a DSL for scripting COM components. These share much with FVision in their use of Haskell as a vehicle for expressing abstraction and modularity. General discussions of the advantages of programming with pure functions are also quite numerous; two of particular relevance to our work are one that describes the use of functional languages for rapid prototyping [9], and one that describes the power of higher-order functions and lazy evaluation as the "glue" needed for modular programming[12].

Pipelines are very similar to the notion of *streams* in the functional programming community, addressed by any good textbook on Haskell (e.g. [2]). The use of streams in signal processing and operating systems contexts dates back many years [8]. Streams have also been proposed as a basis for functional animation [1].

7 CONCLUSIONS

A domain-specific language is a powerful software engineering tool that increases productivity and flexibility in complex applications where ordinary program libraries are less effective. Creating a full-fledged DSL from a library was more difficult than expected but the results were well worth the investment. Lessons learned about DSL design from this project include:

1. The level of interface between native code and the DSL is a crucial choice in developing an effective system. Sometimes, this involves going deeper into the domain than one might expect.

2. The process of DSL design can uncover interesting insights which may not be apparent even to domain specialists. Working from the "bottom up" to develop a language forces both the domain specialists and the DSL specialists to examine (or re-examine) the underlying domain for the right abstractions and interfaces.

3. Performance, even for soft real-time applications, can be acceptable provided care is taken in the interfaces.

4. Haskell served well as a basis for the embedded DSL. The principal features of Haskell, a rich polymorphic type system and higher-order functions, were a significant advantage in the DSL. Adding a small Haskell interpreter to the system did not significantly increase its size or degrade performance.

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

REFERENCES

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

[2] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, New York, 1988.

[3] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages.* USENIX, October 1997.

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

[5] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Glasgow Functional Programming Workshop*, Ullapool, July 1995.

[6] 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.

[7] G. D. Hager and K. Toyama. The "XVision" system: A general purpose substrate for real-time vision applications. *Comp. Vision, Image Understanding.*, 69(1):23–27, January 1998.

[8] P. Henderson. Purely functional operating systems. In *Functional Programming and Its Applications: An Advanced Course*, pages 177–192. Cambridge University Press, 1982.

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

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

[11] 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.

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

[13] 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.

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

[15] Simon Peyton-Jones, Erik Meijer, and Dan Leijen. Scripting COM components in Haskell. In *Proceedings of 5th International Conference on Software Reuse*, pages 224–233. IEEE/ACM, 1998.

[16] SL. Peyton Jones, T. Nordin, and A. Reid. Greencard: a foreign-language interface for Haskell. In *Proc Haskell Workshop*, Amsterdam, June 1997.

[17] 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.

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