Functional Languages Meet Vision and Robotics: FRP in Action

Gregory D. Hager
Department of Computer Science
Johns Hopkins University

John Peterson
Yale University

in collaboration with
Paul Hudak and Henrik Nilsson
Yale University

Izzet Pembeci and Xiangtian Dai
Johns Hopkins University

Copyright ©2002, Gregory D. Hager
All rights reserved.
Outline

- Motivation and History
- Brief introduction to FRP
- FRP-style systems for vision and robotics
- Some current and future applications
Motivation

XVision: 20,000 lines C++ Vision, Graphics, UI

ServoMatic: 20,000 lines C++ Control

XWindows: ∞ lines C Graphics, UI
How Best to Integrate?

• The “Unix” approach
  – \( x = \text{torobot(\text{control(\text{track(objects))})}} \) “Integration by for loop”
  – or callbacks
  – or worker functions

• The “Windows” approach
  – threads
  – threads
  – threads “Integration by appeal to a higher power”

A “language gap” between what we want to express in terms of component domain constructs and the “glue” we have to put components together

The common thread here is time flow - each sub-system has to advance its clock and communicate with other sub-systems.
Other Problems

• Typical recurring implementation chores
  – Writing loops to step forward discretely in time
  – Time slicing time-varying components that operate in parallel
  – Synchronizing updates across components
• Code reuse
  – Two pieces of code need to do *almost* the same thing, but not quite
  – *interconnection patterns* among components
• What’s correct?
  – The design doesn’t look at all like the code
  – Hard to tell if it’s a bug in the code, or a bug in the design

Declarative Programming!
Our Goal

To use modern programming language ideas in the design of domain-specific languages and environments for sensor-driven robot programming.

We make use of Functional Reactive Programming (FRP)
- Extends *any* programming language with a uniform notion of *time flow*
- Continuous time (control system) and discrete time (event oriented)
- Dynamic switching (generate new connection patterns on the fly)
- Fully explicit component interfaces
- Sequential and parallel process combination
- Explicit handle on resource use for some subsets of FRP

**FRP lets us describe what to do not how to do it**
Software Architectures

• A good DSL does not impose an architecture on the application. It captures architectural families and allows different architectures to be used at different levels of the system.

• Functional abstraction is used to capture architectural patterns.

• We use FRP as a *substrate* on which to build languages in domains that involve time flow.
Why FRP for Vision and Robotics?

- Continuous and discrete semantic domains
- Clear expression
  - programs are close to the design
  - programs express *what* we want to do, not *how* to do it
- Architecture neutral
  - create abstractions as needed
  - common “look-and-feel”
- Potentially rich component interfaces

*Modular Domain Specific Languages and Tools*, Paul Hudak, *ICSR 98*
History

• Conal Elliott (Microsoft Research) develops Fran, a language of interactive animations, an embedded language in Haskell.
• The “core” of Fran is separated and generalized, yielding FRP.
• FRP is used to build FROB for robot control and FVision for visual tracking.
• FROB is used to teach robotics at Yale using Scouts and a simulator.
• Serious performance problems in FRP delay general release and AFRP (for Arrowized FRP) is developed. AFRP is nearly ready to release. AFRP is implemented as a Haskell library.
• RT-FRP and E-FRP dialects are developed to address resource constraints. These languages are implemented via compilation and can be used in embedded systems.
• RaPID is developed at JHU to incorporate the FRP programming style into C++. Rapid is incorporated into XVision2.
• FROB is ported to AFRP and used in demos. This is not yet in release.
All of these examples are written in Haskell - a functional language. But Haskell isn’t essential to our methodology.

Paren not needed for function application: \( f \times y \)

Polymorphic types are an essential part of the system:

\[ f :: \text{Integer} \rightarrow \text{Float} \rightarrow \text{Bool} \]

f is a function with two arguments, Integer and Float, and returning Bool

Types may have parameters:

\[ g :: \text{SF Integer Bool} \]

SF is a type with two arguments

Type variables represent dependencies among parameters

\[ h :: a \rightarrow \text{SF} a a \]

h is a function from a value of any type (“a”) to a value of type SF a a, where the a’s in the output must be the same type as the input
FROB Basics

Dynamics (time variation) is fundamental to interaction:

\[\begin{align*}
v &= \sigma(v_{\text{max}}, f - \bar{d}^*) \\
\omega &= \sigma(\sin(\theta_{\text{max}}) \times v_{\text{curr}}, s - \bar{d}^*) - \dot{s}
\end{align*}\]
Dynamics (time variation) is fundamental to interaction:

\[
v = \sigma(v_{\text{max}}, f - d^*) \\
\omega = \sigma(\sin(\theta_{\text{max}}) \times v_{\text{curr}}, s - d^*) - \dot{s}
\]
Dynamics (time variation) is fundamental to interaction:

\[ v = \sigma(v_{\text{max}}, f - d^*) \]
\[ \omega = \sigma(\sin(\theta_{\text{max}}) \cdot v_{\text{curr}}, s - d^*) - \dot{s} \]
Dynamics (time variation) is fundamental to interaction:

\[ v = \sigma(v_{\text{max}}, f - d^*) \]

\[ \omega = \sigma(\sin(\theta_{\text{max}}) * v_{\text{curr}}, s - d^*) - \dot{s} \]
FROB Basics

Dynamics (time variation) is fundamental to interaction:

\[ v = \sigma(v_{\text{max}}, f - d^*) \]
\[ \omega = \sigma(\sin(\theta_{\text{max}}) \times v_{\text{curr}}, s - d^*) - \dot{s} \]
Dynamics (time variation) is fundamental to interaction:

\[
v = \sigma(v_{\text{max}}, f - \bar{d}^*)
\]

\[
\omega = \sigma(\sin(\theta_{\text{max}}) \ast v_{\text{curr}}, s - d^*) - \dot{s}
\]
Signal Functions

Components are encapsulated as signal functions.

- Integer → Float
- Float
- Integer → Bool

Input Signals: Integer, Float, Integer
Output Signals: Float, Bool
Signal Functions

Components are encapsulated as signal functions.

Type signature for this component:
\[ f :: SF (\text{Integer}, \text{Float}, \text{Integer}) (\text{Float}, \text{Bool}) \]
Components are encapsulated as signal functions.

Type signature for this component:
\[ f :: SF \text{ (Integer, Float, Integer)} \text{ (Float, Bool)} \]

```
f = proc (i, f, j) -> (g,b) where
    ...
```

Names for input and output signals
Components are encapsulated as signal functions.

Type signature for this component:
\[ f :: SF \text{ (Integer, Float, Integer)} \text{ (Float, Bool)} \]

\[
f = \text{proc } (i, f, j) \rightarrow (g, b) \text{ where } b = i > j \]
\[
\ldots
\]

Pointwise computations on instantaneous values
Components are encapsulated as signal functions.

Type signature for this component:
f :: SF (Integer, Float, Integer) (Float, Bool)

\[
f = \text{proc } (i, f, j) \rightarrow (g, b) \text{ where } \\
b = i > j \\
o1 \leftarrow c1 \leftarrow i1 \\
o2 \leftarrow c2 \leftarrow i2 \\
\ldots
\]
Signal Functions

Full definition of a signal function:

\[ f = \text{proc (i, f, j) \rightarrow (g,b) where} \]
\[ b = i > j \]
\[ o1 \leftarrow c1 \leftarrow i1 \]
\[ o2 \leftarrow c2 \leftarrow i2 \]
\[ g = o1 + o2 \]
\[ i1 = f \]
\[ i2 = f \]
A FROB Wall Follower

\[
\begin{align*}
v &= \sigma(v_{\text{max}}, f - d^*) \\
\omega &= \sigma(\sin(\theta_{\text{max}}) \cdot v_{\text{curr}}, s - d^*) - \dot{s}
\end{align*}
\]

wallFollow :: Distance -> SF (Velocity, Distance, Distance) (Velocity, RotVel)

wallFollow d_star = proc (v_curr, f, s) -> (v, omega)
where
\[
\begin{align*}
v &= \text{limit } v_{\text{max}} (f - d_{\text{star}}) \\
\dot{s} &= \text{derivative } s \\
\omega &= \text{rerror} - \dot{s} \\
\text{rerror} &= \text{limit } (v_{\text{curr}} \cdot \sin \theta_{\text{max}}) (d_{\text{star}} - s)
\end{align*}
\]
An event is a signal that occurs only at some times.

Events carry a value; we write “Event a” as the type of an event carrying type “a”.

A signal function that uses an event has a type such as

\[ f :: SF (Event a) a \]

Here \( f \) reads a signal of events carrying type “a” and produces a continuous output also of type “a”

Functions on events:

- \( .|: Event a \rightarrow Event a \rightarrow Event a \) Merging of event streams
- \( \text{tag} :: Event a \rightarrow b \rightarrow Event b \) Changing the value of an event
- \( \text{edge} :: SF \text{Bool} (Event ()) \) Watch for an edge in a boolean signal. (()) = “void”.
Basic Signal Functions

integral :: Fractional a => SF a a

hold :: a -> SF (Event a) a
Switching allows the network of components to be dynamically altered.

\[
\text{until} :: \text{SF } a \text{ b} \rightarrow \text{SF } a \text{ (Event (SF a b))} \rightarrow \text{SF } a \text{ b}
\]

- Initial component
- Switching event defines a new component
- Overall circuit has type SF a b

There are many different ways to do switching; AFRP contains a number of different switching constructs.
But We Need More
But We Need More
But We Need More
But We Need More

- Turn Left
- No Wall
- Follow Left
- Turn Right
- Wall left
- Blocked
- Free
- No Wall
- Turn Left
But We Need More
But We Need More

[Diagram of a maze with nodes for 'Wall Follow Left', 'Turn Left', 'Turn Right', 'Blocked', 'Free', 'No Wall', 'Wall left']
But We Need More
But We Need More
But We Need More
A *task* couples a behavior with a termination event. In its simplest form, we combine a behavior and an event into a task:

\[
\text{mkTask} :: \text{SF} \ a \ (b, \ \text{Event} \ c) \rightarrow \text{Task} \ a \ b \ c
\]

Continuous value defined by task

Value returned at end of task

\[
\text{wallTask} \ d_{\star} = \text{mkTask}
\]

\[
\begin{align*}
\text{proc} \ (v_{\text{curr}}, \ f, \ s) & \rightarrow (\text{wc}, \ e) \quad \text{where} \\
\text{wc} & \leftarrow \text{wallFollow} \ d_{\star} \leftarrow (v_{\text{curr}}, \ f, \ s) \\
\text{eBlk} \ k & \leftarrow \text{edge} \leftarrow (f \leq d) \\
\text{eWall} & \leftarrow \text{edge} \leftarrow (s \geq 2*d) \\
e & = \text{eBlk} \ k \ `\text{tag}\` \ \text{Blocked} \mid \text{eWall} \ `\text{tag}\` \ \text{NoWall}
\end{align*}
\]
Using Tasks

```haskell
patrol d_star =
  do status <- wallTask d_star
    case status of
      NoWall   -> turnLeft
      Blocked  -> turnRight
  patrol d_star
```
Tasks and Customizing Behavior

A simple task algebra:

**Primitive tasks:**
- Infinite task (never terminates): `constT :: b -> Task a b c`
- Empty tasks (terminates immediately): `return :: c->Task a b c`

**Operators on a task:**
- `timeOut :: Task a b c -> Time -> Task a b (Maybe c)`
- `abortWhen :: Task a b c -> SF a (Event d) -> Task a b (Either c d)`
- `withSF :: Task a b c -> SF a d -> Task a b (c,d)`
- `withMyResult :: (a -> Task a b) -> Task a b`

Flexible customization of a small fixed library of primitives
The “Bug” Algorithm

followWall :: Distance -> Task RobotInput RobotOutput ()
driveTo    :: Point2 -> Task RobotInput RobotOutput Bool
atPlace    :: Point2 -> SF RobotInput (Event ())
atMin      :: SF (a, Double) a
place      :: RobotInput -> Point2

bug goal   = do finished <- driveTo goal
               if not finished then do
               goAround goal
               bug goal
               else return ()

goAround goal = do closestPoint <- findClosestPlace goal
circleTo closestPoint

circleTo p    = followWall 0.20 `abortWhen` atPlace p

circleOnce    = do initp <- robotPlace
                   followWall 0.20 `timeOut` 5.0
                   circleTo initp

closestPlace goal = proc i -> p where
                   p <- atMin <- (place i, distance (place i) goal)

findClosestPlace goal = circleOnce `withSF` closestPlace goal
FROB views the outside world as supplying and consuming “records” of data. Records are assembled by operators that “combine” output data.

Each application needs a customized input and output routine to interface with system hardware.
We use classes (same as interfaces in Java) to generalize over families of devices.

A typical function:

```
getMessage :: (Radio i) => SF i (Event String)
```
**Type Examples**

**Typical Frob library code (Scout robot)**

**Interface for robots with local odometry:**

```haskell
class Odometry i where
  position :: i -> Point2
  heading   :: i -> Angle
```

**Definition of Scout as a type:**

```haskell
data Scout = << definition of Scout parameters >>
```

**Binding of Scout to the odometry interface:**

```haskell
instance Odometry Scout where
  position = << hardware specific code >>
```

FVision: FRP for Vision

How should we put this into FRP?
• **Prediction**
  – prior states predict new appearance

• **Image rectification**
  – generate a “normalized view”

• **Offset computation**
  – compute error from nominal

• **State update**
  – apply correction to fundamental state
Image Stabilization

type Src s = SF s Image
type Stepper s = SF Image (Delta s, Error)
type Tracker s = SF s (s, Error)

basicTracker :: Stepper s -> Src s -> s -> Tracker s

ssd :: Src Pos -> Pos -> Tracker Pos

loopSF :: a -> (b -> a) -> SF a b -> SF a b
loopSF \( p0 \) feedback sf = proc p -> b
where
    pos' <- delay p0 <- pos
    b <- sf <- pos'
    pos = feedback b
Feature Composition

- **Corners composed of two edges**
  - edges provide one positional parameter and one orientation.
  - two edges define a corner with position and 2 orientations.
  - relationship defined by a projection-embedding pair
An Abstraction: Tracker Algebra

Clownface

Eyes

Eye

BestSSD

Eye

BestSSD

Mouth

BestSSD

Image Processing
An Abstraction: Tracker Algebra
An Abstraction: Tracker Algebra

A Stream Processor

Clownface

Eyes
Eye
BestSSD
Eye
BestSSD
Eye
BestSSD
Eyes

Image Processing

Mouth
MatchSSD
BestSSD
Mouth

Clownface
Tracker Algebra

type EPair a1 a2 a = (Splitter a a1 a2, Joiner a1 a2 a)

composite2 :: EPair a1 a2 a -> Tracker a1 -> Tracker a2
                -> Tracker a

winnersp s = (s,s)
winnerjn (s1,s2) = if (err s1 < err s2) s1 else s2

bestof f g = composite2 (winnersp, winnerjn) f g

bestSSD (i1, i2) src p0 = bestof (ssd i1 src p0)
                           (ssd i2 src p0)
Tracker Algebra

type Splitter a a1 a2 = a -> (a1,a2)
type Joiner a1 a2 a = (a1, a2) -> a
type EPair a1 a2 a = (Splitter a a1 a2, Joiner a1 a2 a)

composite2 :: EPair a1 a2 a -> Tracker a1 -> Tracker a2 -> Tracker a

winnersp :: Splitter a a a
winnersp s = (s,s)

winnerjn :: Joiner ((a, Error), (a, Error)) (a, Error)
winnerjn (s1,s2) = if (err s1 < err s2) s1 else s2

bestof :: Tracker a -> Tracker a -> Tracker a
bestof f g = composite2 (winnersp, winnerjn) f g

type Best a = (a,a) -- a good one and a bad one

bestSSD :: Best Image -> Src a -> a -> Tracker a
bestSSD (i1, i2) src p0 = bestof (ssd i1 src p0) (ssd i2 src p0)
type Splitter a a1 a2 = a -> (a1,a2)
type Joiner a1 a2 a = (a1, a2) -> a
type EPair a1 a2 a = (Splitter a a1 a2, Joiner a1 a2 a)

composite2 :: EPair a1 a2 a -> Tracker a1 -> Tracker a2
           -> Tracker a

winnersp :: Splitter a a a
winnersp s = (s,s)

winnerjn :: Joiner ((a, Error), (a, Error)) (a, Error)
winnerjn (s1,s2) = if (err s1 < err s2) s1 else s2

bestof :: Tracker a -> Tracker a -> Tracker a
bestof f g = composite2 (winnersp, winnerjn) f g

type Best a    = (a,a)  -- a good one and a bad one

bestSSD :: Best Image -> Src a -> a -> Tracker a
bestSSD (i1, i2) src p0 = bestof (ssd i1 src p0) (ssd i2 src p0)
Clown Face

\[
\begin{align*}
\text{trackMouth } v & = \text{bestSSD mouthIms } (\text{newsrcI } v (\text{sizeof mouthIms})) \\
\text{trackLEye } v & = \text{bestSSD leyeIms } (\text{newsrcI } v (\text{sizeof leyeIms})) \\
\text{trackREye } v & = \text{bestSSD reyeIms } (\text{newsrcI } v (\text{sizeof reyeIms})) \\
\text{trackEyes } v & = \text{composite2 } (\text{split}, \text{join}) \ (\text{trackLEye } v) \ (\text{trackREye } v) \\
\text{where} \\
\text{split} & = \text{segToOrientedPts} \quad \text{--- some geometry} \\
\text{join} & = \text{orientedPtsToSeg} \quad \text{--- some more geometry} \\
\text{trackClown } v & = \text{composite2 } \text{concat2 } (\text{trackEyes } v) \ (\text{trackMouth } v)
\end{align*}
\]
Clown Face

drawMouth v = drawbest mouthIms v
drawLEye v = drawbest leyeIms v
drawREye v = drawbest reyeIms v
drawEyes v = composite2 split (drawLEye v) (drawREye v)
  where
    split = segToOrientedPts  --- some geometry

drawClown v = composite2 concat2 (drawEyes v) (drawMouth v)
Tracking = Animation and its Inverse

\[
\begin{align*}
\text{trackMouth } v &= \text{bestSSD mouthIms (newsrcI } v \text{ sizeof mouthIms)} \\
\text{trackLEye } v &= \text{bestSSD leyeIms (newsrcI } v \text{ sizeof leyeIms)} \\
\text{trackREye } v &= \text{bestSSD reyeIms (newsrcI } v \text{ sizeof reyeIms)} \\
\text{trackEyes } v &= \text{composite2 (split, join) (trackLEye } v \text{) (trackREye } v) \\
&\quad \text{where} \\
&\quad \text{split } = \text{segToOrientedPts} \quad \text{--- some geometry} \\
&\quad \text{join } = \text{orientedPtsToSeg} \quad \text{--- some more geometry} \\
\text{trackClown } v &= \text{composite2 concat2 (trackEyes } v \text{) (trackMouth } v) \\
\text{drawMouth } v &= \text{drawbest mouthIms } v \\
\text{drawLEye } v &= \text{drawbest leyeIms } v \\
\text{drawREye } v &= \text{drawbest reyeIms } v \\
\text{drawEyes } v &= \text{ani.composite2 split (drawLEye } v \text{) (drawREye } v) \\
&\quad \text{where} \\
&\quad \text{split } = \text{segToOrientedPts} \\
\text{drawClown } v &= \text{ani.composite2 split2 (drawEyes } v \text{) (drawMouth } v) \\
\end{align*}
\]
Vision-Based Animation


FVision: A Declarative Language for Visual Tracking, John Peterson, Greg Hager, Paul Hudak, and Alastair Reid, PADL 01
– Is FRP really tied to Haskell?
– Can we get similar functionality but live in a mainstream language?
– Faster importing and use of native libraries

We have implemented a version of FRP directly in C++.

Most of the functionality of the Haskell-based AFRP system can be captured in C++. 
RaPID Implementation

- **Datatypes**
  - Discrete behaviors
  - Events
  - Combinations thereof as dataflow graphs

- **Operators**
  - Algebraic equations that construct a graph
  - Allows direct, type transparent lift from C function types

- **Execution**
  - specialized “lazy” evaluation by walking graph
  - specialized garbage collector just for graph elements
JHU Modular Light-Weight Vision System

- Pentium X @ Y GHz
- SRI SVS (Small Vision System)
- Omnidirectional Camera
- Cardbus Firewire Interface for Linux
- XVision2 image processing library
- Redesigned from ground-up using FRP
A Test: A Human-in-the-Loop Navigation Architecture

- Task Oriented Control
- Non-Task Oriented Control
- Execution system
- Sensors
- Effectors
- Human Interface

Signals
“Control Programs”
Events
Stereo Vision
Fast Ground Plane Segmentation
Obstacle Detection

Problem to solve:
Distinguish between relevant obstacles (B,D) and irrelevant (A,C) obstacles
Obstacles
Tracking
RaPID Approach and Obstacle Avoidance

display = disp \ll (\text{liftB(drawLines)} \ll= \text{footprint})
  \ll (\text{liftB(drawState)} \ll= \text{state})
  \ll (\text{liftB(drawTarget)} \ll= \text{target})
  \ll \text{guess} \ll= \text{video(*vid)}

vel = \text{switchB( ScoutVel(0), events )};

\text{events} = \text{stopE} \mid \text{turnE} \mid \text{moveE} \mid \text{trackE};

\text{trackE} = (\text{filterE(lambda(c,c==' '))(disp.key())})

\text{ThenConstB}

\text{track\_behavior(target,footprint)};

\text{state} = \text{delayB(ScoutState(0))} \ll= \text{drive} \ll= \text{vel};

(\text{display, state}).\text{run}();
Haskell/FRP, Color Tracking and Obstacle Avoidance

Haskell with color tracking and obstacle avoidance

```
track_follow =
do tr <- colorTracker
  let st = proc inp -> do
    (x,y,w,h) <- tr -< fviImage inp
    ls <- arr (obsLines stereoV) -< (\x->0) inp
    (v,t) <- arr driveVector -< driveWithOA ls (mid (x,y,w,h))
    returnA -< ddSetVelTR v t `coCompose`
        fvgoOverlayRectangle (x,y) ((x+w), (y+h)) Blue
        `coCompose` toLines ls
    mkTask (st &&& fvgiLBP)
  nullT
```
Complete System In Action (Haskell)
Stereo vs. Color

x: -11.56, y: 1.65, theta: 6.20
depth: 1374.67
x: 1076.04, y: 842.38, h: -0.0422, wx: 0.0589, wy: 0
Stereo vs. Color
Advantages of Frob

• Conceptual *and* formal correctness
  – Domain proof + language semantics = program proof!

• Programs in the target domain are:
  – more concise
  – quicker to write
  – easier to maintain
  – can be written by non-experts

  \{ \text{Contribute to higher programmer productivity} \}

• “All-in-one” (extensible!) package
  – abstractions can be used to express your favorite architecture
    • subsumption
    • behaviors
    • TDL/Colbert/Saphira
    • ....
FRP Evaluation

- **Performance**
  - CPU not a problem
  - Space/GC also not a problem

- **Code size**
  - XVision = 21K core + 11K l
  - FVision = 21K core + 700 interface + 2K

- **Development times**
  - FVision: weekend;
  - HIL system: a couple of days (plus lots of interfaces, etc.)

- **Abstractions/Interfaces**
  - RaPID or GHC make interfacing “easy”
  - Comparative analysis suggests we can mimic many architectures
Lessons Learned

No one software architecture is capable of meeting all the needs of vision and robotics; different types and levels of system use different architectures

- **FRP is architecture neutral**
  - many “paradigms” are easy to express in FRP

- **Transformation Programming**
  - FRP/Frob makes it easy to transparently refine programs

- **Prototyping**
  - “time-based” languages lead to compact (and sometimes) novel expression of code

- **Deep interfaces**
  - need to integrate components at many different levels depending on application goals

- **It’s hard to kick the habit once you’ve got it ...**
What’s Next?

• **Scalability and abstractions**
  - Vision-based robot tour guide (Pembeci)
  - VICS
  - HMCS

• **FRP followons (Zhanyong Wan, Yale U.)**
  - RT-FRP (real-time FRP)
    • bounded space and time
  - E-FRP (event-driven FRP)
    • subset of RT-FRP
    • compilation to efficient C++
Some New Projects That May Use FRP

- Medical robotics: Center for Computer Integrated Surgical Systems and Technology (CISST)
- Visual Interaction Cues (VICs)
FRP-Relevant links

• Haskell:
  – http://haskell.org

• FRP
  – http://haskell.org/FRP

• CIRL lab
  – http://www.cs.jhu.edu/CIRL

• XVision2
  – http://www.cs.jhu.edu/CIRL/XVision2