Functional Reactive Programming

Cleanly Abstracted Interactivity

C++Now 2014

David Sankel

Why not zone out during this talk?

- Interactivity is quite common
- Functional Reactive Programming is a quite different take on interactivity.
- FRP changes the assumptions
  - Hard → Not Hard
  - Complex → Simple
  - Monolithic → Modular
- Based in math

Recap: Denotational Semantics Design

Vocabulary

Iterator Vocabulary

```
for (std::vector<int>::const_iterator
         i = v.begin();
     i != v.end(); ++i) {
  std::cout << *i << std::endl;
}
```

Foreach Vocabulary

```
for (int i : v) {
  std::cout << i << std::endl;
}
```

What do you see? (1)

What do you see? (2)

What do you see? (3)

What is a mouse position?

Time → Point2D

or

function<Point2D (Time)>

How would we implement that?

```
Point2D mousePos( Time t ) {
  //?
}
```

Example 3: First try

```
Drawing circleFollowsMouse(
    function<Point2D(Time)> mousePos,
    Time t) {
  return circleAt(mousePos(t));
}
```

Example 3: Up the ante

```
function<Drawing(Time)>
circleFollowsMouse(
    function<Point2D(Time)> mousePos)
{
  return [mousePos](Time t) {
    return circleAt(mousePos(t));
  };
}
```

Behaviors

Time → T for some type T

```
template <typename T>
using Behavior = function<T(Time)>;

Behavior<int> ≈ function<int(Time)>
Behavior<Drawing> ≈ function<Drawing(Time)>
```

Behaviors: handy utilities (1)

```
template <typename T>
Behavior<T> always(T value) {
  return [value](Time) {
    return value;
  };
}
```

Behaviors: handy utilities (2)

```
template <typename R, typename Args...>
Behavior<R>
map(function<R(Args...)> func,
    Behavior<Args>... behaviors);
```

Map example

```
Drawing drawOver(Drawing top,
                 Drawing bottom);
Behavior<Drawing> topBehavior = …;
Behavior<Drawing> bottomBehavior = …;

Behavior<Drawing> combined =
    map(drawOver, topBehavior,
        bottomBehavior);
```

Example 3: Revisit (1)

```
function<Drawing(Time)>
circleFollowsMouse(
    function<Point2D(Time)> mousePos)
{
  return [mousePos](Time t) {
    return circleAt(mousePos(t));
  };
}
```

Example 3: Revisit (2)

```
Behavior<Drawing> circleFollowsMouse(
      Behavior<Point2D> mousePos) {
   return [mousePos](Time t) {
      return circleAt(mousePos(t));
   };
}
```

Example 3: Revisit (3)

```
Behavior<Drawing> circleFollowsMouse(
      Behavior<Point2D> mousePos) {
   return map(circleAt, mousePos);
}
```

Example 2

```
Behavior<Drawing> spinningBall(
    Behavior<Point2D> mousePos) {
  //?
}
```

Example 2

```
Behavior<Drawing>
spinningBall(Behavior<Point2D>) {
  //?
}
```

Example 2

```
Behavior<Drawing>
spinningBall(Behavior<Point2D>) {
  Behavior<Point2D> spinningPoint =
      /*?*/;
  return map(circleAt,spinningPoint);
}
```

Example 2: spinningPoint (1)

```
Behavior<Point2D> spinningPoint = [](
     Time t) {
  return Point2D(/*?*/, /*?*/);
};
```

Example 2: spinningPoint (2)

```
Behavior<Point2D> spinningPoint = [](
    Time t) {
  return Point2D(
      50 * std::cos(t * 2 * π),
      50 * std::sin(t * 2 * π));
};
```

Example 2: Put it all together

```
Behavior<Drawing>
spinningBall(Behavior<Point2D>) {
  Behavior<Point2D> spinningPoint = [](
      Time t) {
    return Point2D(
        50 * std::cos(t * 2 * π),
        50 * std::sin(t * 2 * π));
  };
  return map(circleAt,spinningPoint;
}
```

Some Composition Operations

```
template <typename T>
Behavior<T> operator+(Behavior<T> lhs,
                      Behavior<T> rhs);

template <typename T>
Behavior<T> operator-(Behavior<T> lhs,
                      Behavior<T> rhs);
```

spinningBall for reuse

```
Behavior<Point2D> spinningPoint = [](
    Time t) {
  return Point2D(
      50 * std::cos(t * 2 * π),
      50 * std::sin(t * 2 * π));
};
Behavior<Point2D> spinningBall =
    map(circleAt, spinningPoint);
```

Another Behavior

```
Behavior<Drawing>
spinningBallFollowsMouse(
    Behavior<Point2D> mousePos) {
  return map(circleAt,
             mousePos + spinningPoint);
}
```

Functional Reactive Programming History

Conal Elliot

Paul Hudak

- 1997. Functional Reactive Animation. Elliott and Hudak. (First FRP paper)
- 2001. Genuinely Functional User Interfaces. Courtney and Elliott. (AFRP)
- 2002. Functional Reactive Programming, Continued. Nilsson, Courtney, and Peterson (Yampa is born)
- 2003. The Yampa Arcade. Courtney, Nilsson, Peterson.
- 2009. Push-pull functional reactive programming. Elliott.

FRP Implementation Problems

- Poor and often unpredictable consumption of space.
- Lacking dynamic collection capabilities.
- Subtle implementations wrt. Laziness
- Complex to use with imperative libraries.

FRP Implementation Problems

- Poor and often unpredictable consumption of space.
- Lacking dynamic collection capabilities.
- Subtle implementations wrt. Laziness
- Complex to use with imperative libraries.

Haskell Problems!

How can we solve these problems?

???
???
???
???
???
???
???
???
???
???

How can we solve these problems?

??
??
??
??
??
??
??
??
??
??

Use C++

Use C++

Specifying Behavior in C++

Dai, Hager, and Peterson

2002

- Translation of Haskell FRP syntax to C++.
- Subtle space leaks/awkward dynamic collections remained.

Use C++

Specifying Behavior in C++

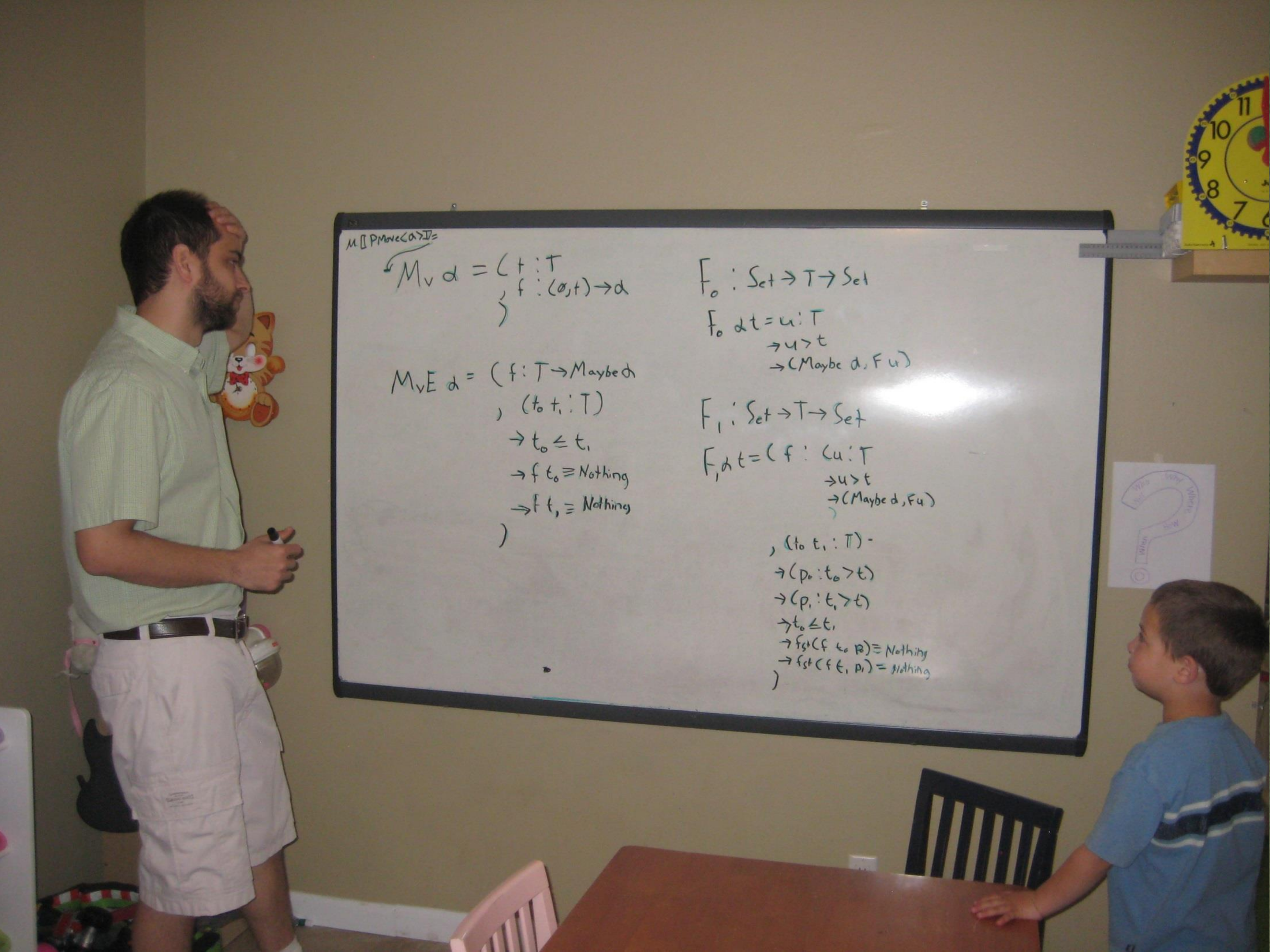
Dai, Hager, and Peterson

2002

- Translation of Haskell FRP syntax to C++.
- Subtle space leaks/awkward dynamic collections remained.

???

???



$\mathcal{M}[\![\mathrm{PMove}\langle\alpha\rangle]\!] =$

```
$M_v\,\alpha = (t : T$
          $, f : (\sigma, t) \to \alpha$
          $)$

$M_vE\,\alpha = (f : T \to \mathrm{Maybe}\,\alpha$
             $, (t_0\ t_1 : T)$
               $\to t_0 \leq t_1$
               $\to f\,t_0 \equiv \mathrm{Nothing}$
               $\to f\,t_1 \equiv \mathrm{Nothing}$
             $)$

$F_0 : \mathrm{Set} \to T \to \mathrm{Set}$

$F_0\,\alpha\,t = u : T$
            $\to u > t$
            $\to (\mathrm{Maybe}\,\alpha, F\,u)$

$F_1 : \mathrm{Set} \to T \to \mathrm{Set}$

$F_1\,\alpha\,t = (f : (u : T$
                    $\to u > t$
                    $\to (\mathrm{Maybe}\,\alpha, F\,u)$
                    $)$

            $, (t_0\ t_1 : T) \cdot$
            $\to (p_0 : t_0 > t)$
            $\to (p_1 : t_1 > t)$
            $\to t_0 \leq t_1$
            $\to \mathrm{fst}\,(f\ t_0\ p_0) \equiv \mathrm{Nothing}$
            $\to \mathrm{fst}\,(f\ t_1\ p_1) = \mathrm{Nothing}$
            $)$
```

sfrp

A C++ Functional Reactive Programming library derived directly from the original semantics into C++.

Wormhole Example

Wormhole example

```
Behavior<Drawing>
circleGrow(Behavior<Point2D> mousePos) {
  Behavior<bool> inCircle = /*?*/;
  Behavior<float> circleRadius = /*?*/;
  return map(circleWithRadiusAt,
             circleRadius,
             always(Point2D(0.0, 0.0)));
}
```

Wormhole synopsis

```
template <typename T> struct Wormhole {
  Wormhole(const T &value);

  sfrp::Behavior<T>
  outputBehavior() const;

  sfrp::Behavior<T> setInputBehavior(
      const Behavior<T> &inputBehavior)
      const;
};
```

Wormhole usage

```
Wormhole<int> hole(0); // '0' initial
                       // value.
// use 'hole.outputBehavior()‘
// ...
Behavior<int> finalBehavior =
    hole.setInputBehavior(/*...*/);
```

Growing circle with wormhole

```
Behavior<Drawing>
circleGrow(Behavior<Point2D> mousePos) {
  Wormhole<float> circleRadiusWormhole(10);
  Behavior<bool> inCircle =
      map([](Point2D pos, float radius)
              ->bool {
            float distToCenter = std::sqrt(
                 pos.x * pos.x + pos.y * pos.y);
            return distToCenter < radius;
          },
          mousePos,
          circleRadiusWormhole .outputBehavior());
  Behavior<float> circleRadius =
    circleRadiusWormhole.setInputBehavior(/*?*/);
  //...
}
```

Wormholes

- Allows mutual dependencies between behaviors.
- Time shift property handy (integration, smoothing, etc.)

And

- No space leaks
- No subtle time leaks (delay insertion)

Follows from derivation of semantics into C++

Interaction with imperative code (1)

Push based behaviors

```
template <typename T>
pair<Behavior<optional<T> >,
     function<void(const T &)> >
trigger();
```

Pull based behaviors

```
// 'valuePullFunc' always called with
// increasing values.
template <typename T>
Behavior<T> fromValuePullFunc(
    function<T(Time)> valuePullFunc);
```

Interaction with imperative code (2)

Simulating behaviors

```
template <typename T> struct Behavior {
  // Must be called with increasing time
  // values.
  T pull(const double time) const;
  //...
};
```

sfrp Implementation Solutions

- Optimal and predictable consumption of space.
- Dynamic collection capabilities.
- Clear implementation (no laziness).
- Simple to use with imperative libraries.

sfrp: Industrial Stength FRP

Case Study:

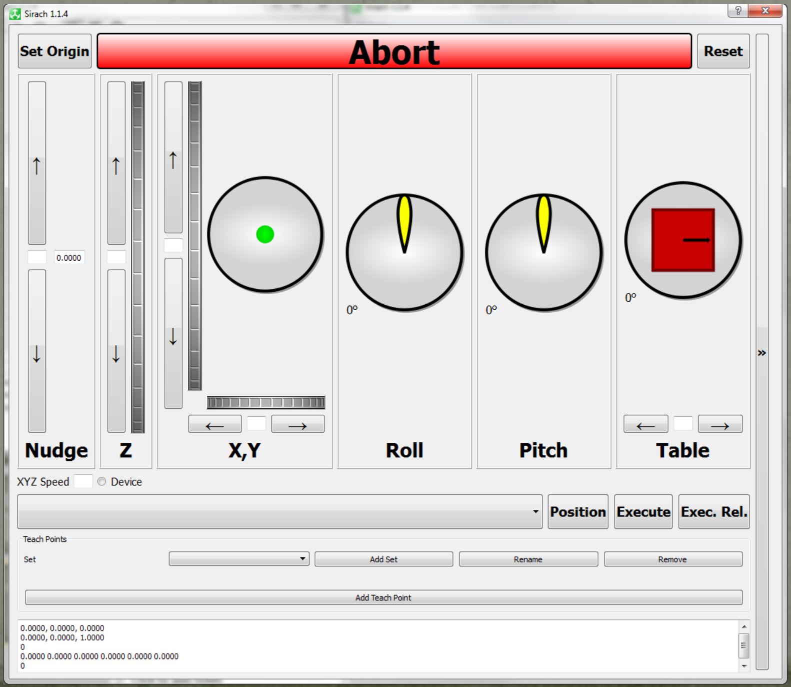
Sandia National Labs
6-Axis Layered Manufacturing Robot
2010-2012

Software Requirements:

- Real-time inverse kinematics
- Limited motion ranges
- Motor Speed limits
- Tangential accuracy subordinate to positional accuracy
- Real time adjustments of path during build

6-Axis Layered Manufacturing Robot

- All requirements met
- 2,500 lines of sfrp-specific code
- Qt widgets used as behaviors
- Challenging even with FRP's level of abastraction



```
0.0000, 0.0000, 0.0000
0.0000, 0.0000, 1.0000
0
0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0
```

6-Axis Layered Manufacturing Robot

Surprise nudge control requirement:

- Real-time adjustment of tip offset
- Change of apparent geometry of robot
- Feature was added in less than a day.

Nudge amount defined as behavior based on widgets and then used as adjustment of the driver specified geometry:

```
    const sfrp::Behavior<Point1D> nudgeB = /**/;
```

All speed and other constraints needed no adjustments.

sfrp: Functional Reactive Programming in C++

When to use:

- Robotics
- Computer Animation
- Games
- Simulations
- Anything with interactivity, especially complex interactivity.

Benefits:

- Cleanly abstracted (semantics: range for vs. iterators)
- Practical (language choice, implementation path)
- Composible (like legos!)

Cleanly Abstracted Interactivity

Lots more to learn:

- Events: behaviors with specific occurrences
- Mixing events and behaviors
- Behaviors of behaviors
- Integration

Clone it:

https://github.com/camio/CppNow2014

https://github.com/camio/sbase