ITU FRP 2010

Lecture 5: The Yampa Implementation

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK

Outline

- The basic implementation approach
- Optimization
- Aggressive optimization using GADTs

A basic implementation: SF (1)

Each signal function is essentially represented by a ***transition function***. Arguments:

- Time passed since the previous time step.
- The current input value.

Returns:

- A (possibly) updated representation of the signal function, the ***continuation***.
- The current value of the output signal.

A basic implementation: SF (2)

```
type DTime = Double

data SF a b =
  SF {sfTF :: DTime -> a
               -> Transition a b}

type Transition a b = (SF a b, b)
```

The continuation encapsulates any internal state of the signal function. The type synonym `DTime` is the type used for the time deltas, > 0.

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

- Loops over the sampling points.

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

- Loops over the sampling points.
- At each sampling point:

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

- Loops over the sampling points.
- At each sampling point:
  - reads input sample and time from the external environment (typically I/O action)

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

- Loops over the sampling points.
- At each sampling point:
  - reads input sample and time from the external environment (typically I/O action)
  - feeds sample and time passed since previous sampling into the signal function's transition function

A basic impl.: `reactimate` (1)

The function `reactimate` is responsible for animating a signal function:

- Loops over the sampling points.
- At each sampling point:
  - reads input sample and time from the external environment (typically I/O action)
  - feeds sample and time passed since previous sampling into the signal function's transition function
  - writes the resulting output sample to the environment (typically I/O action).

A basic impl.: `reactimate` (2)

- The loop then repeats, but uses the continuation returned from the transition function on the next iteration, thus ensuring any internal state is maintained.

A basic implementation: arr

```
arr :: (a -> b) -> SF a b
arr f = sf
  where
    sf = SF {sfTF = \_ a -> (sf, f a)}
```

Note: It is obvious that `arr` constructs a ***stateless*** signal function since the returned continuation is exactly the signal function being defined, i.e. it never changes.

A basic implementation: >>>

For >>>, we have to combine their continuations into updated continuation for the composed arrow:

```
(>>>) :: SF a b -> SF b c -> SF a c
(SF {sfTF = tf1}) >>> (SF {sfTF=tf2}) =
    SF {sfTF = tf}
  where
    tf dt a = (sf1' >>> sf2', c)
      where
        (sf1', b) = tf1 dt a
        (sf2', c) = tf2 dt b
```

Note how ***same*** time delta is fed to both subordinate signal functions, thus ensuring synchrony.

A basic impl.: How to get started? (1)

What should the very first time delta be?

A basic impl.: How to get started? (1)

What should the very first time delta be?

- Could use 0, but that would violate the assumption of positive time deltas (time always progressing), and is a bit of a hack.

A basic impl.: How to get started? (1)

What should the very first time delta be?

- Could use 0, but that would violate the assumption of positive time deltas (time always progressing), and is a bit of a hack.
- Instead:
  - Initial SF representation makes a first transition given just an input sample.
  - Makes that transition into a representation that expects time deltas from then on.

A basic impl.: How to get started? (2)

```
data SF a b =
  SF {sfTF :: a -> Transition a b}

data SF' a b =
  SF' {sfTF' :: DTime -> a
                      -> Transition a b}

type Transition a b = (SF' a b, b)
```

`SF'` is internal, can be thought of as representing a “running” signal function.

Optmimizing >>>: First Attempt (1)

The arrow identity law:

```
arr id >>> a  =  a  =  a >>> arr id
```

Optmimizing >>>: First Attempt (1)

The arrow identity law:

```
arr id >>> a = a = a >>> arr id
```

How can this be exploited?

Optmimizing >>>: First Attempt (1)

The arrow identity law:

$$\texttt{arr id >>> a} = \texttt{a} = \texttt{a >>> arr id}$$

How can this be exploited?

1. Introduce a constructor *representing* `arr id`

```
data SF a b = ...
            | SFId
            | ...
```

Optmimizing >>>: First Attempt (1)

The arrow identity law:

```
arr id >>> a = a = a >>> arr id
```

How can this be exploited?

1. Introduce a constructor *representing* `arr id`

```
data SF a b = ...
            | SFId
            | ...
```

2. Make SF abstract by hiding all its constructors.

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

4. Define optimizing version of >>>:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf  = sf
...
```

Optmimizing >>>: First Attempt (2)

3. Ensure `SFId` only gets used at intended type:

```
identity :: SF a a
identity = SFId
```

4. Define optimizing version of >>>:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

```
:: SF b c ≠ SF a c
```

No optimization possible?

The type system does not get in the way of all optimizations. For example, for:

```
constant :: b -> SF a b
constant b = arr (const b)
```

the following laws can readily be exploited:

```
      sf >>> constant c = constant c
constant c >>> arr f    = constant (f c)
```

But to do better, we need GADTs.

Generalized Algebraic Data Types

GADTs allow

- individual specification of return type of constructors
- the more precise type information to be taken into account during case analysis.

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId
            | ...
```

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId        :: SF a b
            | ...
```

Optmimizing >>>: Second Attempt (1)

Instead of

```
data SF a b = ...
            | SFId
            | ...
```

:: SF a b

we define

```
data SF a b where
    ...
    SFId :: SF a a
    ...
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of >>> ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of >>> ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of >>> ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...

:: SF a a
```

Optmimizing >>>: Second Attempt (2)

Define optimizing version of >>> ***exactly*** as before:

```
(>>>) :: SF a b -> SF b c -> SF a c
...
  SFId >>> sf = sf
...
```

```
:: SF a a                :: SF a c
```

Other Ways?

There are other ways to implement this kind of optimisation (e.g. Hughes 2004). However:

Other Ways?

There are other ways to implement this kind of optimisation (e.g. Hughes 2004). However:

- GADTs offer a completely straightforward solution

Other Ways?

There are other ways to implement this kind of optimisation (e.g. Hughes 2004). However:

- GADTs offer a completely straightforward solution
- absolutely no run-time overhead.

Other Ways?

There are other ways to implement this kind of optimisation (e.g. Hughes 2004). However:

- GADTs offer a completely straightforward solution
- absolutely no run-time overhead.

The latter is important for Yampa, since the signal function network constantly must be monitored for emerging optimization opportunities:

```
arr g >>> switch (...) (\_ -> arr f)
```

$\overset{switch}{\Longrightarrow}$ `arr g >>> arr f` $=$ `arr (f . g)`

Laws Exploited for Optimizations

General arrow laws:

```
(f >>> g) >>> h = f >>> (g >>> h)
    arr (g . f) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
```

Laws involving `const` (the first is Yampa-specific):

```
 sf >>> arr (const k) = arr (const k)
arr (const k)>>>arr f = arr (const(f k))
```

Laws Exploited for Optimizations

General arrow laws:

```
(f >>> g) >>> h = f >>> (g >>> h)
    arr (g . f) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
```

Laws involving `const` (the first is Yampa-specific):

```
 sf >>> arr (const k) = arr (const k)
arr (const k)>>>arr f = arr (const(f k))
```

Implementation (1)

```
data SF a b where
  SFArr ::
    (DTime -> a -> (SF a b, b))
    -> FunDesc a b
    -> SF a b
  SFCpAXA ::
    (DTime -> a -> (SF a d, d))
    -> FunDesc a b->SF b c->FunDesc c d
    -> SF a d
  SF ::
    (DTime -> a -> (SF a b, b))
    -> SF a b
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

Recovering the function from a `FunDesc`:

```
fdFun :: FunDesc a b -> (a -> b)
fdFun FDI     = id
fdFun (FDC b) = const b
fdFun (FDG f) = f
```

Implementation (2)

```
data FunDesc a b where
    FDI :: FunDesc a a
    FDC :: b -> FunDesc a b
    FDG :: (a -> b) -> FunDesc a b
```

Recovering the function from a `FunDesc`:

```
fdFun :: FunDesc a b -> (a -> b)
fdFun FDI     = id
fdFun (FDC b) = const b
fdFun (FDG f) = f
```

Implementation (3)

```
fdComp :: FunDesc a b -> FunDesc b c
          -> FunDesc a c
fdComp FDI fd2 = fd2
fdComp fd1 FDI = fd1
fdComp (FDC b) fd2 =
    FDC ((fdFun fd2) b)
fdComp _ (FDC c) = FDC c
fdComp (FDG f1) fd2 =
    FDG (fdFun fd2 . f1)
```

Events

Yampa models ***discrete-time*** signals by lifting the ***range*** of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Discrete-time signal $=$ `Signal` (`Event` α).

Events

Yampa models ***discrete-time*** signals by lifting the ***range*** of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Discrete-time signal $=$ `Signal` (`Event` α).

Consider composition of pure event processing:

```
f :: Event a -> Event b
g :: Event b -> Event c

arr f >>> arr g
```

Optimizing Event Processing (1)

Additional function descriptor:

```
data FunDesc a b where
    ...
    FDE :: (Event a -> b) -> b
                -> FunDesc (Event a) b
```

Optimizing Event Processing (1)

Additional function descriptor:

```
data FunDesc a b where
  ...
  FDE :: (Event a -> b) -> b
             -> FunDesc (Event a) b
```

Optimizing Event Processing (1)

Additional function descriptor:

```
data FunDesc a b where
    ...
    FDE :: (Event a -> b) -> b
               -> FunDesc (Event a) b
```

Extend the composition function:

```
fdComp (FDE f1 f1ne) fd2 =
    FDE (f2 . f1) (f2 f1ne)
    where
      f2 = fdFun fd2
```

Optimizing Event Processing (2)

Extend the composition function:

```
fdComp (FDG f1) (FDE f2 f2ne) = FDG f
  where
    f a =
      case f1 a of
        NoEvent -> f2ne
        f1a     -> f2 f1a
```

Optimizing Event Processing (2)

Extend the composition function:

```
fdComp (FDG f1) (FDE f2 f2ne) = FDG f
  where
    f a =
      case f1 a of
        NoEvent -> f2ne
        f1a     -> f2 f1a
```

Optimizing Stateful Event Processing

A general stateful event processor:

```
ep :: (c -> a -> (c,b,b)) -> c -> b
      -> SF (Event a) b
```

Optimizing Stateful Event Processing

A general stateful event processor:

```
ep :: (c -> a -> (c,b,b)) -> c -> b
      -> SF (Event a) b
```

Composes nicely with stateful and stateless event processors!

Optimizing Stateful Event Processing

A general stateful event processor:

```
ep :: (c -> a -> (c,b,b)) -> c -> b
      -> SF (Event a) b
```

Composes nicely with stateful and stateless event processors!
Introduce explicit representation:

```
data SF a b where
  ...
  SFEP :: ...
    -> (c -> a -> (c, b, b)) -> c -> b
    -> SF (Event a) b
```

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of >>>:

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of >>>:

- Completely unoptimized: 15 lines

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of >>>:

- Completely unoptimized: 15 lines
- Some optimizations (current): 45 lines

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of >>>:

- Completely unoptimized: 15 lines
- Some optimizations (current): 45 lines
- GADT-based optimizations: 240 lines

Cause for Concern

Code with GADT-based optimizations is getting large and complicated:

- Many more cases to consider.
- Larger size of signal function representation.

Example: Size of >>>:

- Completely unoptimized: 15 lines
- Some optimizations (current): 45 lines
- GADT-based optimizations: 240 lines

Is the result really a performance improvement?

Micro Benchmarks (1)

A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

Micro Benchmarks (1)

A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.

Micro Benchmarks (1)

A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.
- No significant performance overhead.

Micro Benchmarks (1)

A number of Micro Benchmarks were carried out to verify that individual optimizations worked as intended:

- Yes, works as expected.
- No significant performance overhead.
- Particularly successful for optimizing event processing: additional stages can be added to event-processing pipelines with almost no overhead.

Micro Benchmarks (2)

Most important gains:

- Insensitive to bracketing.
- A number of “pre-composed” combinators no longer needed, thus simplifying the Yampa API (and implementation).
- Much better event processing.

Micro Benchmarks (2)

Most important gains:

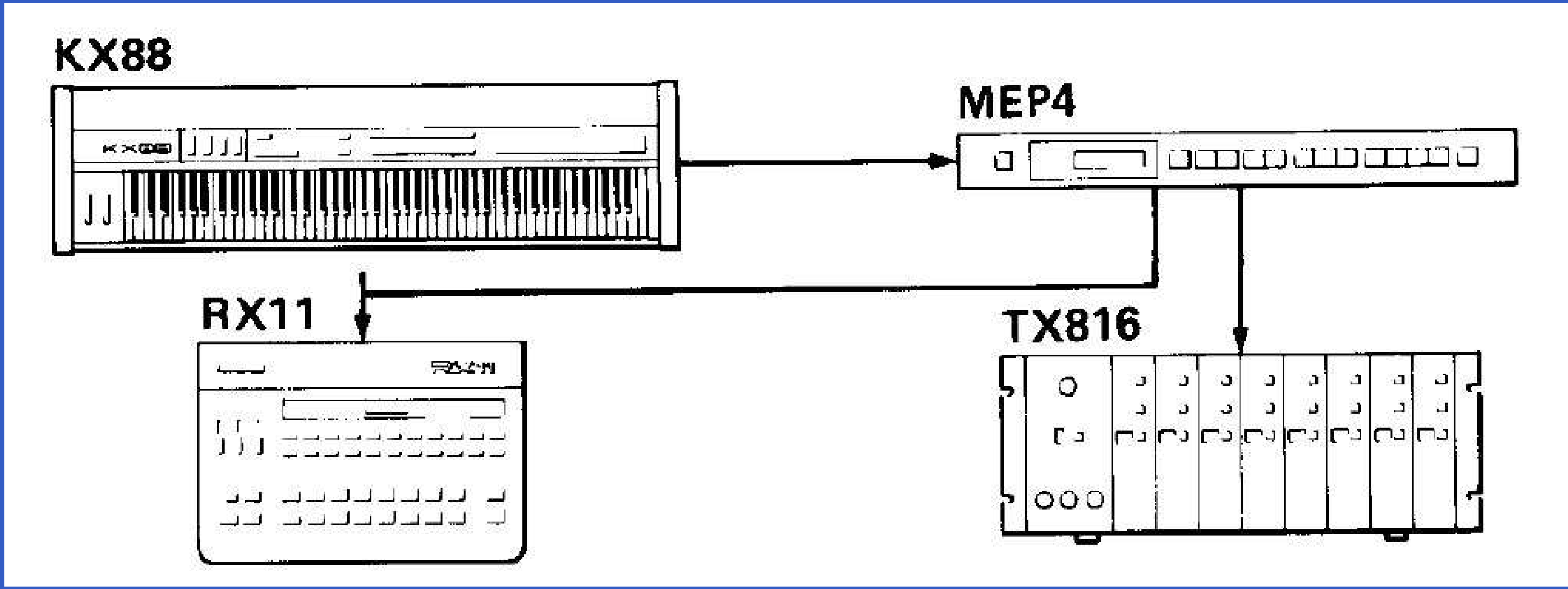
- Insensitive to bracketing.
- A number of “pre-composed” combinators no longer needed, thus simplifying the Yampa API (and implementation).
- Much better event processing.

But what about overall, system-wide performance impact? ***Does it make a difference???***

Benchmark 1: Space Invaders

Benchmark 2: MIDI Event Processor

High-level model of a MIDI event processor programmed to perform typical duties:



The MEP4

STUDIO
QUAD 4
midi express
USB
EXPRESS XT
YAMAHA
MEP4
Cheetah
MULTI TIMBRAL SYNTHESIZER MODULE
Oberheim
Matrix-1000
Volume
Power
OFF
ON
YAMAHA

Results

Benchmark	T_{U} [s]	T_{S} [s]	T_{G} [s]	$T_{\mathrm{S}}/T_{\mathrm{U}}$	$T_{\mathrm{G}}/T_{\mathrm{S}}$
Space Inv.	0.95	0.86	0.88	0.91	1.02
MEP	19.39	10.31	9.36	0.53	0.91

Reading

- Henrik Nilsson. Dynamic Optimization for Functional Reactive Programming using Generalized Algebraic Data Types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 54–65, Tallinn, Estonia, September, 2005.