Dynamic Optimization for Functional Reactive Programming using Generalized Algebraic Data Types

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK

Introduction

- Generalized Algebraic Data Types (GADTs) recently added to GHC.

Introduction

- Generalized Algebraic Data Types (GADTs) recently added to GHC.
- GADTs are a limited form of dependent types, closely related to inductive families.

Introduction

- Generalized Algebraic Data Types (GADTs) recently added to GHC.
- GADTs are a limited form of dependent types, closely related to inductive families.
- GADTs offer considerably enlarged scope for enforcing important important invariants statically.

Introduction

- Generalized Algebraic Data Types (GADTs) recently added to GHC.
- GADTs are a limited form of dependent types, closely related to inductive families.
- GADTs offer considerably enlarged scope for enforcing important important invariants statically.
- GADTs also offer the tantalizing possibility of writing more *efficient* programs.

This Talk

A case study on the applications of GADTs for performance optimizations in the context of Yampa:

This Talk

A case study on the applications of GADTs for performance optimizations in the context of Yampa:

- What kind of optimization possibilities do GADTs open up?

This Talk

A case study on the applications of GADTs for performance optimizations in the context of Yampa:

- What kind of optimization possibilities do GADTs open up?
- What is the impact, performance and other?

This Talk

A case study on the applications of GADTs for performance optimizations in the context of Yampa:

- What kind of optimization possibilities do GADTs open up?
- What is the impact, performance and other?

Results should be of interest also for other Domain-Specific Embedded Languages, especially arrow-based ones.

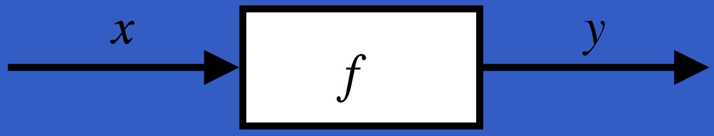
Yampa

Yampa is

- a domain-specific language for Functional Reactive Programming
- related to synchronous dataflow langauges and modelling and simulation langauges
- implemented as a self-optimizing, arrow-based Haskell combinator library.

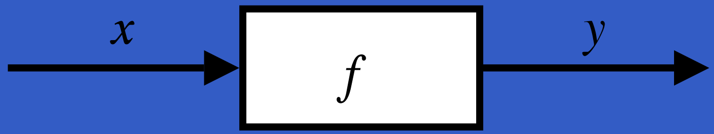
Signal functions

Key concept in Yampa: ***functions on signals***.



Signal functions

Key concept in Yampa: ***functions on signals***.

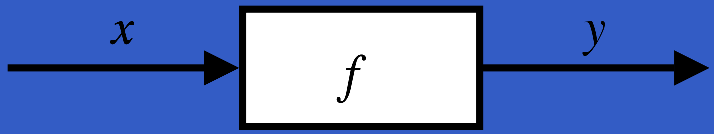


Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time}{\rightarrow}\alpha$$
$$x\ \texttt{::}\ \texttt{Signal}\ \alpha$$
$$y\ \texttt{::}\ \texttt{Signal}\ \beta$$
$$f\ \texttt{::}\ \texttt{Signal}\ \alpha{\rightarrow}\texttt{Signal}\ \beta$$

Signal functions

Key concept in Yampa: ***functions on signals***.



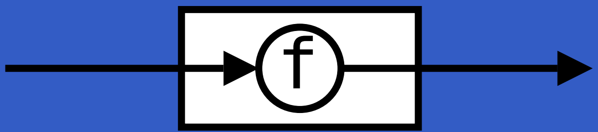
Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$$
$$x :: \texttt{Signal}\ \alpha$$
$$y :: \texttt{Signal}\ \beta$$
$$f :: \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

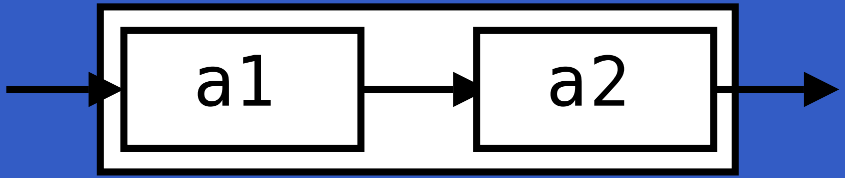
Signal function type:

$$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

Arrows: Lifting and Composition



`arr f`



`a1 >>> a2`

Type signatures in Yampa:

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

Optmimizing >>>: First Attempt (1)

The arrow identity law:

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

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:

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

How can this be exploited?

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

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

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
...
```

(annotation pointing to `sf`): $::$ `SF b c` $\neq$ `SF a c`

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)
```

$\stackrel{switch}{\Longrightarrow}$ `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 $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

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:

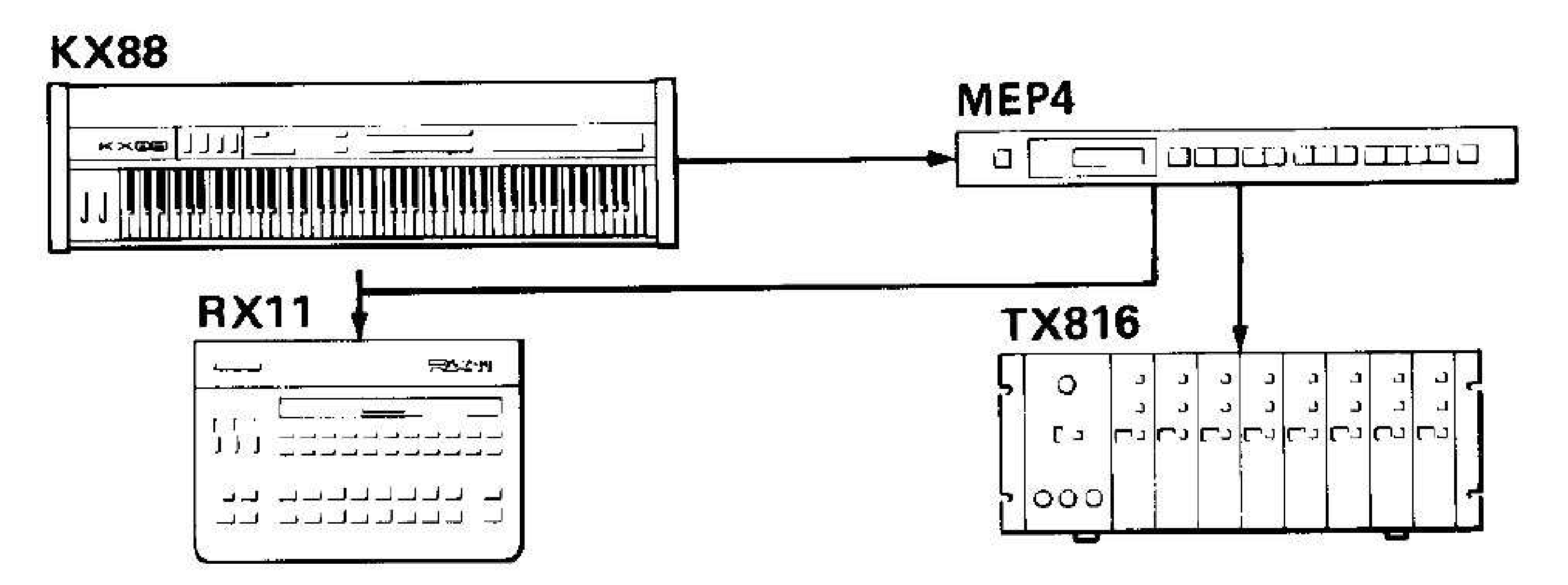
- Insensitive to bracketing.
- A number of “pre-composed” combinators no longer needed, thus simplifying the Yampa API (and implementation).
- Much better event processing.

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


DigiTech
STUDIO
QUAD 4
ULTRAPATCH PRO
midi express
EXPRESS XT
USB
YAMAHA
MEP4
Cheetah
MS6 MULTI TIMBRAL SYNTHESIZER MODULE
Oberheim
Matrix-1000
Volume
YAMAHA
A5000

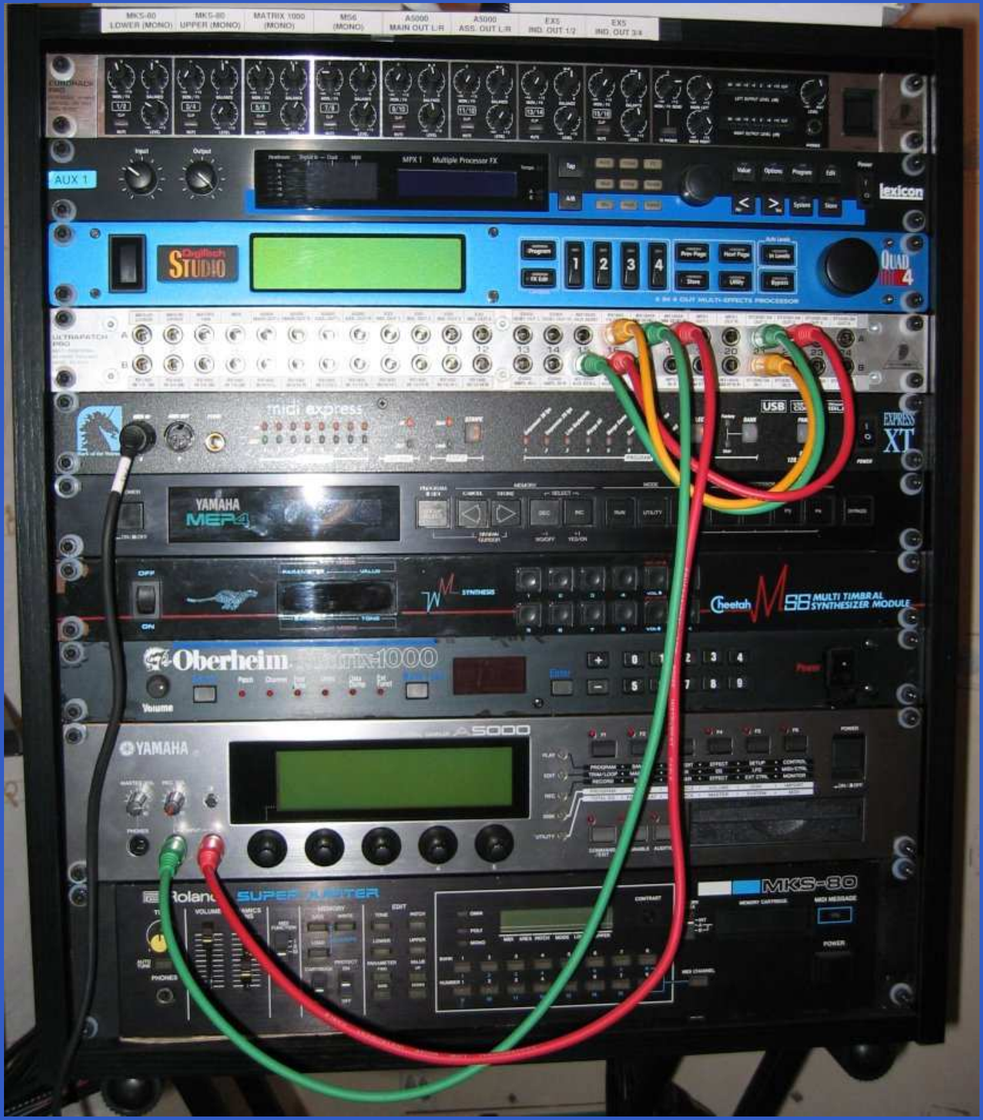
Results

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

Conclusions

- GADTs are powerful and easy-to-use.
- GADTs made a better Yampa implementation possible.
- Overall performance improvement lower than what was initially hoped for, but still worthwhile for certain kinds of applications.

Finally: Behind the Scenes



Finally: Behind the Scenes