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 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 invariants statically.
- GADTs also offer the tantalizing possibility of writing more efficient programs.
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.
Key concept in Yampa: *functions on signals*.

$$\signal{x} \xrightarrow{f} \signal{y}$$
Signal functions

Key concept in Yampa: *functions on signals*.

Intuition:

\[
\begin{align*}
\text{Signal } \alpha & \approx \text{Time} \rightarrow \alpha \\
x & :: \text{Signal } \alpha \\
y & :: \text{Signal } \beta \\
f & :: \text{Signal } \alpha \rightarrow \text{Signal } \beta
\end{align*}
\]
Signal functions

Key concept in Yampa: *functions on signals.*

Intuition:

\[
\text{Signal } \alpha \approx \text{Time} \rightarrow \alpha \\
x :: \text{Signal } \alpha \\
y :: \text{Signal } \beta \\
f :: \text{Signal } \alpha \rightarrow \text{Signal } \beta
\]

Signal function type:

\[
\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
\]
Arrows: Lifting and Composition

Type signatures in Yampa:

\[
\text{arr} :: (a \rightarrow b) \rightarrow \text{SF} a b \\
(>>>>) :: \text{SF} a b \rightarrow \text{SF} b c \rightarrow \text{SF} a c
\]
The arrow identity law:

```
arr id >>> a = a = a >>> arr id
```
Optmimizing >>>: First Attempt (1)

The arrow identity law:

\[ \text{arr id} \gggg a = a = a \gggg \text{arr id} \]

How can this be exploited?
Optmimizing >>>>: First Attempt (1)

The arrow identity law:

\[
\text{arr id} \gggg \ a = a = a \gggg \text{arr id}
\]

How can this be exploited?

1. Introduce a constructor representing \text{arr id}\n
   \[
   \text{data SF a b} = \ldots
   \]

   \[
   \mid \text{SFId}
   \]

   \[
   \mid \ldots
   \]
Optmimizing >>>: First Attempt (1)

The arrow identity law:

\[
\text{arr id} \gggg a = a = a \gggg \text{arr id}
\]

How can this be exploited?

1. Introduce a constructor *representing* \( \text{arr id} \)
   
   data \( \text{SF a b} \) = ...

2. Make \( \text{SF} \) abstract by hiding all its constructors.
3. Ensure `SFId` only gets used at intended type:

   identity :: SF a a
   identity = SFId
3. Ensure \texttt{SFId} only gets used at intended type:

\begin{verbatim}
identity :: SF a a
identity = SFId
\end{verbatim}

4. Define optimizing version of \texttt{>>>}:

\begin{verbatim}
(>>>) :: SF a b -> SF b c -> SF a c

... 
SFId >>> sf = sf 
...
\end{verbatim}
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`
   
   ...
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.
Instead of

```haskell
data SF a b = ...
    | SFId
    | ...
```
Instead of

```haskell
data SF a b = ... SFId ... :: SF a a ...
:: SF a b
```
Instead of

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

we define

```
data SF a b where
  ...
  SFId :: SF a a
  ...
```
Define optimizing version of \( \ggg \) \textit{exactly} as before:

\[
(\ggg) :: \text{SF}~a~b \rightarrow \text{SF}~b~c \rightarrow \text{SF}~a~c
\]
Define optimizing version of >>> exactly as before:

\[(\ggg\ggg) :: SF \ a \ b \ \to \ SF \ b \ c \ \to \ SF \ a \ c\]

\[
\ldots
\]

\[
\text{SFId} \ \ggg \ sf = sf
\]

\[
\ldots
\]
Define optimizing version of \( \ggg g \) exactly as before:

\[
(\ggg g) :: \text{SF} \ a \ b \to \text{SF} \ b \ c \to \text{SF} \ a \ c
\]

\[
\text{SFId} \ggg g \ sf = sf
\]

\[
:: \text{SF} \ a \ a
\]
Define optimizing version of `>>>` *exactly* as before:

\[
(\gggg) :: \text{SF} \ a \ b \to \text{SF} \ b \ c \to \text{SF} \ a \ c
\]

\[
\text{SFId} \ \gggg \ \text{sf} = \text{sf}
\]

\[
:: \text{SF} \ a \ a
:: \text{SF} \ a \ c
\]
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:

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

switch

arr g >>> arr f = arr (f . g)
```
Laws Exploited for Optimizations

General arrow laws:

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr} (g . f) = \text{arr} f >>> \text{arr} g\]
\[\text{arr} \text{id} >>> f = f\]
\[f = f >>> \text{arr} \text{id}\]

Laws involving \texttt{const} (the first is Yampa-specific):

\[sf >>> \text{arr} (\text{const} k) = \text{arr} (\text{const} k)\]
\[\text{arr} (\text{const} k) >>> \text{arr} f = \text{arr} (\text{const} (f k))\]
Laws Exploited for Optimizations

General arrow laws:

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (g \cdot f) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
\[f = f >>> \text{arr } \text{id}\]

Laws involving \texttt{const} (the first is Yampa-specific):

\[s f >>> \text{arr } (\text{const } k) = \text{arr } (\text{const } k)\]
\[\text{arr } (\text{const } k) >>> \text{arr } f = \text{arr } (\text{const } (f k))\]
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
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
data FunDesc a b where

  FDI :: FunDesc a a

  FDC :: b -> FunDesc a b

  FDG :: (a -> b) -> FunDesc a b
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
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)

\[
\begin{align*}
\text{fdComp} & :: \text{FunDesc } a \ b \rightarrow \text{FunDesc } b \ c \\
& \quad \rightarrow \text{FunDesc } a \ c \\
\text{fdComp } \text{FDI} \ \text{fd2} & = \ \text{fd2} \\
\text{fdComp } \ \text{fd1} \ \text{FDI} & = \ \text{fd1} \\
\text{fdComp } (\text{FDC } b) \ \text{fd2} & = \\
& \quad \ \text{FDC } ((\text{fdFun } \text{fd2}) \ b) \\
\text{fdComp } _{\_} (\text{FDC } c) & = \ \text{FDC } c \\
\text{fdComp } (\text{FDG } f1) \ \text{fd2} & = \\
& \quad \ \text{FDG } (\text{fdFun } \text{fd2} \ . \ f1)
\end{align*}
\]
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).
Yampa models *discrete-time* signals by lifting the *range* of continuous-time signals:

\[
\text{data Event } a = \text{NoEvent} \mid \text{Event } a
\]

*Discrete-time signal* \(=\) \(\text{Signal (Event } a)\).

Consider composition of pure event processing:

\[
f :: \text{Event } a \to \text{Event } b
g :: \text{Event } b \to \text{Event } c
\]

\[
\text{arr } f \gg \gg \text{arr } g
\]
Additional function descriptor:

```haskell
data FunDesc a b where
  ...
  FDE :: (Event a -> b) -> b
       -> FunDesc (Event a) b
```
Additional function descriptor:

```haskell
data FunDesc a b where 

... 

FDE :: (Event a -> b) -> b 
-> FunDesc (Event a) b
```
Additional function descriptor:

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

Extend the composition function:

```haskell
fdComp (FDE f1 f1ne) fd2 =
  FDE (f2 . f1) (f2 f1ne)
where
  f2 = fdFun fd2
```
Extend the composition function:

```haskell
fdComp (FDG f1) (FDE f2 f2ne) = FDG f
where
  f a =
    case f1 a of
    NoEvent -> f2ne
    f1a     -> f2 f1a
```
Extend the composition function:

\[
\text{fdComp} \ (\text{FDG} \ f1) \ (\text{FDE} \ f2 \ f2ne) = \text{FDG} \ f
\]

where

\[
f \ a = \begin{cases} \text{NoEvent} & \rightarrow f2ne \\ f1a & \rightarrow f2 \ f1a \end{cases}
\]
A general stateful event processor:

\[
\text{ep} :: (c \to a \to (c, b, b)) \to c \to b \\
\to \text{SF}\ (\text{Event } a)\ b
\]
A general stateful event processor:

\[
\text{ep} :: (c \to a \to (c, b, b)) \to c \to b \\
\to \text{SF} (\text{Event } a) \ b
\]

Composes nicely with stateful and stateless event processors!
A general stateful event processor:

\[ \text{ep} :: (\text{c} \to \text{a} \to (\text{c, b, b})) \to \text{c} \to \text{b} \to \text{SF (Event a) b} \]

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

\[ \text{data SF a b where} \]
\[ \quad \ldots \]
\[ \quad \text{SFEP} :: \ldots \]
\[ \quad \to (\text{c} \to \text{a} \to (\text{c, b, b})) \to \text{c} \to \text{b} \to \text{SF (Event a) b} \]
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?
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 (1)

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

- Yes, works as expected.
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.

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

![Diagram of MIDI equipment including KX88, RX11, MEP4, and TX816]
The MEP4
## Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$T_U$ [s]</th>
<th>$T_S$ [s]</th>
<th>$T_G$ [s]</th>
<th>$T_S/T_U$</th>
<th>$T_G/T_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Inv.</td>
<td>0.95</td>
<td>0.86</td>
<td>0.88</td>
<td>0.91</td>
<td>1.02</td>
</tr>
<tr>
<td>MEP</td>
<td>19.39</td>
<td>10.31</td>
<td>9.36</td>
<td>0.53</td>
<td>0.91</td>
</tr>
</tbody>
</table>
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