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.
- 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.

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 languages and modelling and simulation languages
- implemented as a self-optimizing, arrow-based Haskell combinator library.
Signal functions

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

\[
\begin{array}{c}
\text{x} \\
f \\
y
\end{array}
\]

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:

\[
\text{SF} \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta
\]

Optimizing >>>: First Attempt (1)

The arrow identity law:

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

How can this be exploited?

1. Introduce a constructor representing \(\text{arr id}\)

   \[
   \text{data SF} \ a \ b = \ldots
   \]

   \[
   \mid \text{SFId}
   \]

   \[
   \mid \ldots
   \]

2. Make \(\text{SF}\) abstract by hiding all its constructors.

Optimizing >>>: First Attempt (2)

3. Ensure \(\text{SFId}\) only gets used at intended type:

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

   \[
   \text{id} = \text{SFId}
   \]

4. Define optimizing version of \(\ggg\):

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

   \[
   \ldots
   \]
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.

Optimizing >>>: Second Attempt (1)

Instead of
```
data SF a b = ...
```
we define
```
data SF a b where
  ...
  SFIId :: SF a a
  ...
```

Optimizing >>>: Second Attempt (2)

Define optimizing version of >>> exactly as before:
```
(>>>) :: SF a b -> SF b c -> SF a c
  ...
```

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)
  \hspace{0.5cm} \text{switch}
  \hspace{1cm} arr g >>> arr f = arr (f \ . \ g)
```
Laws Exploited for Optimizations

General arrow laws:

\[(f >>> g) >>> h = f >>> (g >>> h)\]

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

\[\text{arr} id >>> f = f\]

f = f >>> \text{arr} id

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

\[\text{sf} >>> \text{arr} (\text{const} k) = \text{arr} (\text{const} k)\]

\[\text{arr} (\text{const} k) >>> \text{arr} f = \text{arr} (\text{const} (f k))\]

Implementation (1)

data SF a b where

\[
\begin{align*}
\text{SFArr} &:: \quad \text{(DTime} \to a \to (\text{SF} a b, b)) \to \text{FunDesc} a b \\
\text{SFCpAXA} &:: \quad (\text{DTime} \to a \to (\text{SF} a d, d)) \to \text{FunDesc} a b \to \text{SF} a d \\
\text{SF} &:: \quad (\text{DTime} \to a \to (\text{SF} a b, b)) \to \text{SF} a b
\end{align*}
\]

Implementation (2)

data FunDesc a b where

\[
\begin{align*}
\text{FDI} &:: \quad \text{FunDesc} a a \\
\text{FDC} &:: \quad b \to \text{FunDesc} a b \\
\text{FDG} &:: \quad (a \to b) \to \text{FunDesc} a b
\end{align*}
\]

Recovering the function from a FunDesc:

\[
\begin{align*}
\text{fdFun} &:: \quad \text{FunDesc} a b \to (a \to b) \\
\text{fdFun FDI} & = \text{id} \\
\text{fdFun} (\text{FDC} b) & = \text{const} b \\
\text{fdFun} (\text{FDG} f) & = f
\end{align*}
\]

Implementation (3)

\[
\begin{align*}
\text{fdComp} &:: \quad \text{FunDesc} a b \to \text{FunDesc} b c \to \text{FunDesc} a c \\
\text{fdComp FDI} & = \text{fd2} \\
\text{fdComp} & = \text{fd1} \\
\text{fdComp} (\text{FDC} b) & = \text{fd2} \\
\text{fdComp} (\text{FDG} f1) & = \text{fd2}
\end{align*}
\]
Events

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

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

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

Consider composition of pure event processing:

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

\[
\text{arr } f \gg\gg \text{arr } g
\]

Optimizing Event Processing (1)

Additional function descriptor:

\[
\text{data FunDesc } a b \text{ where} \\
\quad \text{...FDE :: (Event } a \rightarrow b) \rightarrow b \\
\quad \rightarrow \text{FunDesc (Event } a \big) b
\]

Extend the composition function:

\[
f d\text{Comp } (\text{FDE } f_1 f_{1ne}) f d_2 = \text{FDE } (f_2 \circ f_1) (f_2 f_{1ne}) \\
\text{where} \\
f_2 = f d\text{Fun } f d_2
\]

Optimizing Event Processing (2)

Extend the composition function:

\[
f d\text{Comp } (\text{FDG } f_1) (\text{FDE } f_2 f_{2ne}) = \text{FDG } f \\
\text{where} \\
f a = \text{case } f_1 a \text{ of} \\
\quad \text{NoEvent } \rightarrow f_{2ne} \\
\quad f_1 a \quad \rightarrow f_2 f_{1a}
\]

Optimizing Stateful Event Processing

A general stateful event processor:

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

Composes nicely with stateful and stateless event processors!

Introduce explicit representation:

\[
\text{data SF } a b \text{ where} \\
\quad \text{...SFEP :: ...} \\
\quad \rightarrow (c } \rightarrow a \rightarrow (c, b, b)) \rightarrow c \rightarrow b \\
\quad \rightarrow \text{SF (Event } a \big) b
\]
Cause for Concern

Code with GADT-based optimizations is getting large and complicated:
- Many more cases to consider.
- Larger size of signal function representation.

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:
- 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???
Benchmark 2: MIDI Event Processor

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

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.