Dynamic Optimization for Functional Reactive Programming using Generalized Algebraic Data Types

Henrik Nilsson

School of Computer Science and Information Technology, University of Nottingham
Henrik.Nilsson@cs.nott.ac.uk

Abstract

A limited form of dependent types, called Generalized Algebraic Data Types (GADTs), has recently been added to the list of Haskell extensions supported by the Glasgow Haskell Compiler. Despite not being full-fledged dependent types, GADTs still offer considerably enlarged scope for enforcing important code and data invariants statically. Moreover, GADTs offer the tantalizing possibility of writing more efficient programs since capturing invariants statically through the type system sometimes obviates entire layers of dynamic tests and associated data markup. This paper is a case study on the applications of GADTs in the context of Yampa, a domain-specific language for Functional Reactive Programming in the form of a self-optimizing, arrow-based Haskell combinator library. The paper has two aims. Firstly, to explore what kind of optimizations GADTs make possible in this context. Much of that should also be relevant for other domain-specific embedded language implementations, in particular arrow-based ones. Secondly, as the actual performance impact of the GADT-based optimizations is not obvious, to quantify this impact, both on tailored micro benchmarks, to establish the effectiveness of individual optimizations, and on two fairly large, realistic applications, to gauge the overall impact. The performance gains for the micro benchmarks are substantial. This implies that the Yampa API could be simplified as a number of "pre-composed" primitives that were there mainly for performance reasons are no longer needed. As to the applications, a worthwhile performance gain was obtained in one case whereas the performance was more or less unchanged in the other.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.3.2 [*Programming Languages*]: Language Classifications—functional languages, dataflow languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—data types and structures, polymorphism

General Terms Languages, Performance

Keywords Functional programming, Haskell, arrows, combinator library, domain-specific languages, DSEL, reactive programming, FRP, Yampa, synchronous dataflow languages, GADT

ICFP '05 September 26–28, 2005, Tallinn, Estonia

1. Introduction

Combinator libraries have proven to be a very effective way of tailoring a functional programming language to address particular programming problems, such as parsing [30, 29], pretty-printing [17, 7], graphical user-interfaces [3], or even entire application domains, such as animation [12], vision [25], or robotics [23]. Part of their attraction is that programming with a well-designed combinator library can be very much like using a customized programming language that provides features and abstractions highly appropriate for the task at hand, but at a fraction of the implementation effort required for a stand-alone language implementation. Indeed, the term Domain-Specific Embedded Languages (DSEL) has been coined to refer to such libraries [15].

An equally important aspect of their attractiveness is that such libraries often can be implemented in a way that achieves levels of performance that are acceptable for many practical purposes. However, this may require quite sophisticated library implementations that carry out static and/or dynamic analysis and optimization, and thus a bit more effort on behalf of the library implementor. A good example is Swierstra's and Duponcheel's self-analyzing parser combinators [29]. Another example is Yampa [21][1], a domain-specific language for Functional Reactive Programming (FRP) in the form of a self-optimizing Haskell combinator library, which provides the setting for this paper.

Unfortunately, as many combinator library implementors have observed, the Hindley-Milner-based type systems of typical, modern functional languages, such as SML or Haskell 98, often tend to get in the way once one try to pursue the optimization approach in earnest. For example, this was noted in the Yampa paper, where it was pointed out that the efficient implementation of a particularly obvious optimization would require a simple form of dependent types [21, p. 62]. Baars and Swierstra [1] and Hughes [19] discuss the problems that the standard Haskell 98 system causes for optimizing DSEL implementations in more general terms. Additionally, they show how certain commonly implemented extensions of the Haskell 98 type system can be put to clever use to work around some of the limitations of the standard type system. Alas, for reasons to be discussed (see section 6), neither of these approaches would be ideal for Yampa.

To give a concrete example, let us consider a simplified version of the problem encountered in the optimizing Yampa implementation. The central abstraction in Yampa is that of a *Signal Function*. A signal function represents a simple, synchronous process, mapping an input signal to an output signal. However, for this example, one can think of signal functions just as a plain function. The type of a signal function is written $\texttt{SF}\,\alpha\,\beta$, where α is the type of the input and β is the type of the output. Yampa is structured using John Hughes' arrow framework [18]. This is an an abstract data

[1] Then called AFRP for *Arrowized Functional Reactive Programming*

type interface for function-like types, particularly suitable for types that represent process-like computations, such as Yampa's signal functions. Two central arrow combinators are `arr`, that constructs a (pure) arrow from an ordinary function (often referred to as *lifting*), and `>>>`, that forms a new arrow by composing two arrows, similar to ordinary function composition. In the context of Yampa, the type signatures of these two combinators are

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

In addition, the arrow framework specifies a set of algebraic laws that all instances of the framework must satisfy. One of the requirements is that the lifting of the identity function `id` must be the identity of arrow composition. That is, for an arbitrary arrow `f`:

```
arr id >>> f  =  f  =  f >>> arr id
```

It would of course be great if these two algebraic identities could be exploited in the definition of `>>>`, as this would eliminate the overhead of an arrow composition and of applying the identity function to the input or output of `f`. In an attempt to implement this for Yampa, one could imagine introducing a constructor for signal functions that represents `arr id`:

```
data SF a b = ...
            | SFId    -- Represents arr id
            | ...
```

The type `SF` is then made abstract by hiding all of its constructors, and an appropriately typed constant is added to the API as the only way of constructing a signal function represented by `SFId`:

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

Note that the constructor `SFId` itself has the more general type `SF a b`. The programmer would be asked to use `identity` in place of `arr id` if he or she wishes to take advantage of the optimizations.

The above algebraic identities can now we exploited in the definition of `>>>`. For example, the following fragment of the definition of `>>>` captures the first of the two algebraic identities above:

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

The problem is now obvious. The defining equation above is only well-typed if the type of the second argument `sf` is the same as the overall return type of `>>>`. That would be the case if the type variables `a` and `b` always were instantiated to the *same* type when the first argument is `SFId`. And indeed, as long as the only way to introduce `SFId` is through the type-constrained constant `identity`, that will be the case. Unfortunately, the type of the constructor `SFId` itself, which is all that matters when type checking the equation above, is too general to enforce that constraint. In fact, the Haskell 98 type system does not provide any way to give `SFId` a sufficiently constrained type, nor any way to exploit such information in individual branches of function definitions or case expressions if it were possible.

However, the recent addition of Generalized Algebraic Data Types (GADTs) [26] to the list of Haskell extensions supported by the Glasgow Haskell Compiler (GHC) gives programmers a lot more freedom. GADTs are a limited form of dependent types, offering a considerably enlarged scope for capturing and thus enforcing important code and data invariants statically. In particular, GADTs are just what is needed to address the problem discussed above since the key idea is to allow constructors that have more specific types than usual, and to take that extra type information into account in individual case branches. GADTs would no doubt also offer an interesting alternative to the methods described by Baars and Swierstra [1] and Hughes [19].

This paper is a case study on the applications of GADTs in the context of Yampa. It has two aims. Firstly, to explore what kind of optimizations that GADTs make possible in this context. It turned out that GADTs are expressive enough to enable the implementation of optimizations well beyond what was originally envisioned. Much of this should also be relevant for other domain-specific embedded language implementations, in particular arrow-based ones. Secondly, to quantify the actual performance impact those optimizations can have, as the performance gains of GADT-based optimizations are not as clear-cut as it might first appear. The problem is that the optimizations do add to the size and complexity of the combinator library. This could have a negative performance impact which might offset the gains from the optimizations.

With this in mind, performance figures are given both for small, tailored, benchmarks, to establish the effectiveness of individual optimizations, and for two fairly large, realistic applications, to gauge their overall impact. The performance gains for the small benchmarks are substantial. This is encouraging in itself, and also implies that the Yampa API could be simplified as a number of "pre-composed" primitives that were there mainly for performance reasons are no longer needed. As to the applications, the first was written well before GADTs were added to GHC and thus without GADT-based optimizations in mind. The other is mainly an event processing application, as it turned out that GADTs enabled a number of interesting optimizations in that area. A worthwhile performance gain was obtained for the event processing one, whereas the performance was more or less unchanged for the other.

The rest of the paper is organized as follows. Section 2 gives the necessary background on arrows, Yampa, and GADTs. Section 3 reviews the the current, *simply* optimized, Yampa implementation, and shows that these optimizations do have a positive performance impact that is sufficiently large to make a substantial difference at the system level for non-trivial applications. This was always the assumption, but had not been confirmed by measurements before. Section 4 then shows how GADTs can be used to optimize Yampa further. The effectiveness of the GADT-based optimization efforts is evaluated in section 5. Related work is discussed in section 6. Section 7, finally, gives conclusions.

2. Technical Background

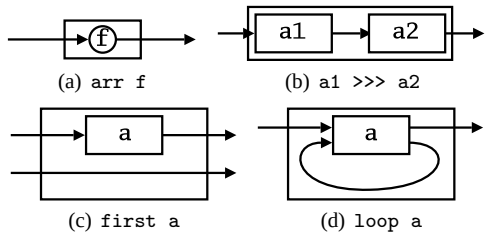
The introduction briefly outlined the arrow framework, Yampa, Generalized Algebraic Data Types, and their respective roles in this paper. This section gives a somewhat more thorough presentation of these topics in the interest of making this paper self-contained. Nothing here is new, and a reader who is familiar with any (or all) of these can probably skim or even skip past the subsection(s) in question without loss of continuity.

2.1 Arrows

The arrow framework, introduced by John Hughes [18], is an abstract data type interface for function-like types. It is particularly suitable for types that represent process-like computations, and that is why it is interesting in the context of Functional Reactive Programming and Yampa. Since arrows can be be seen as computations, arrows are related to monads, but arrows are more general in that they can handle more kinds of computations.

A type constructor of arity two, together with three operations, `arr`, `>>>`, and `first`, form an arrow, provided certain algebraic laws hold (see below). In Haskell, this, except for the algebraic requirements, is captured by the following class definition:

```
class Arrow a where
```



(a) `arr f` (b) `a1 >>> a2`

(c) `first a` (d) `loop a`

Figure 1. The core arrow combinators.

```
arr   :: (b -> c) -> a b c
(>>>) :: a b c -> a c d -> a b d
first :: a b c -> a (b,d) (c,d)
```

The operator `arr` *lifts* an ordinary function to an arrow. Any arrow that can be constructed in that way is called *pure*. The operator `>>>` composes arrows in series, similar to ordinary (albeit reversed) function composition. The operator `first` is a "widening" operation that converts an arrow from type `b` to type `c` to one operating on pairs, processing the first component of the pair through the arrow, but leaving the second component of the pair unchanged. This combinator is crucial when combining arrows to work on more than one input, as it allows part of the input to be "routed past" an arrow for individual processing later.

Ordinary functions is the canonical example of an arrow. In that case, `arr` is just the identity function, `>>>` is reversed function composition, and `first` applies a function to the first component of a pair.

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or *feedback* [22]. Since not all arrow instances support such an operation, it is a method of a separate class:

```
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

An intuitive, and in the context of this paper, fully adequate, way to think about arrows, is as "boxes" with an input and an output, encapsulating a mechanism for computing the output from the input, possibly making use of some form of state in the process. Figure 1 uses that idea to illustrate the arrow combinators graphically.

Other arrow combinators can be defined in terms of these primitives. Commonly used derived combinators are `second`, the mirror image of `first`, and `***` and `&&&`, two forms of parallel arrow composition:

```
second :: Arrow a => a b c -> a (d,b) (d,c)
(***) :: Arrow a => a b c -> a d e
                    -> a (b,d) (c,e)
(&&&) :: Arrow a => a b c -> a b d -> a b (c,d)
```

It is a remarkable fact that using only the three basic arrow operators and `loop`, it is possible to express any conceivable network of interconnected arrows.

As to the algebraic laws, Hughes lists 9 laws for the three basic operators in his paper, and Paterson lists a further 6 laws for `loop`. The following are the laws that are directly exploited for optimization purposes later in this paper:

```
(f >>> g) >>> h  =  f >>> (g >>> h)      (1)
  arr (f >>> g)  =  arr f >>> arr g      (2)
   arr id >>> f  =  f                    (3)
               f  =  f >>> arr id        (4)
```

That is, composition is associative, is preserved by `arr`, and `arr id` is an identity for composition. Note that the fact that functions are arrows is exploited in the formulation of law 2.

2.2 Yampa

Functional Reactive Programming (FRP) is about describing reactive systems as functions mapping *signals* (time-varying values) to signals. The nature of these signals depends on the application domain, but they could represent input from sensors, video streams, or control signals for motors and other actuators. FRP has been applied to a number of domains, for example robotics (Frob) [23, 24], visual tracking (FVision) [25], and graphical user interfaces (Fruit) [8].

FRP grew out of Conal Elliott's and Paul Hudak's work on Functional Reactive Animation [12]. Since then, the basic FRP framework has been implemented in a number of different ways. However, *synchrony* and support for both *continuous* and *discrete* time are common to most of them. There are thus close connections to, on the one hand, synchronous dataflow languages like like Esterel [2], Lustre [5, 13], and Lucid Synchrone [6, 27], and to, on the other hand, hybrid automata [14] and languages for hybrid modeling and simulation, like Simulink [20].

2.2.1 Basics

Yampa is an arrow-based implementation of FRP [21, 16]. Yampa takes the idea of describing systems as functions on signals quite literally and provides *Signal Functions* as the central abstraction. The type of a signal function mapping a signal of type α onto a signal of type β is written `SF` $\alpha\,\beta$. Intuitively:

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

where

$$\texttt{Signal}\,\alpha \approx \texttt{Time} \rightarrow \alpha$$

for some suitable type `Time` representing continuous time. However, only signal functions are first-class entities in Yampa: signals only exist indirectly, through signal functions. This distinguishes Yampa from earlier FRP implementations.

To ensure that signal functions are executable, they are required to be *causal*: the output of a signal function at time t must be uniquely determined by the input signal on the interval $[0, t]$. Note that time here is *local* time, measured from the time at which a signal function is applied to its input signal, or *switched in* using Yampa terminology. If the output at time t is determined solely by the input at time t, the signal function is said to be *stateless*, otherwise it is said to be *stateful*. A simple example of a stateful signal function is `integral`:

```
integral :: SF Double Double
```

defined by

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

where $x(t)$ is the input signal and $y(t)$ is the output signal.

If more than one input or output signal are needed, tuples are used for α or β since a signal of tuples is isomorphic[2] to a tuple of signals. A Yampa system consists of a number of interconnected signal functions, operating on the system input and producing the system output. The signal functions operate in parallel, sensing a common *rate* of time flow. This is why Yampa is a *synchronous* language.

[2] More or less: the fact that the "undefined value" ($\perp$) belongs to every type in Haskell complicates the picture slightly.

Of course, when it comes to implementation, Yampa can only approximate the conceptually continuous signal model since the signals necessarily are evaluated for only a discrete set of sample points. In Yampa, these points need not be equidistant, but it is the *same* set for all continuous-time signals in a system. Another way to think of a signal function is thus as a simple synchronous process, possibly encapsulating some internal state, that at each tick of a global clock reads a value from the input, processes it, updates any internal state, and outputs the result. This is indeed close to how Yampa (and other FRP-based systems) are implemented.

Signal functions are arrows, and Yampa makes the signal function type `SF` an instance of the classes `Arrow` and `ArrowLoop`. The Yampa instance of the combinator `arr` lifts an ordinary function to a signal function by applying the function pointwise to the input signal. The result is a stateless signal function since the instantaneous value of the output signal at any point in time only depends on the instantaneous input value at that same time. Any stateless signal function can be constructed in this way, and stateless signal functions are thus *pure* arrows. The Yampa instance of composition, `>>>`, is what one would expect: at all points in time, the input to the composed signal function is fed into the first subordinate signal function, the resulting output is fed into the second subordinate signal function, and the resulting output from the second signal function is the overall output of the composed signal function. The other basic arrow combinators are defined in a similar manner.

2.2.2 Events

In FRP, the domain of a signal can, conceptually, be either be continuous or discrete. In the former case, the signal is defined at every point in time. In the latter case, the signal is a partial function, only defined at discrete points in time. Such a point of definition is called an *event*. In Yampa, this distinction has been deliberately blurred to make it easier to mix and match continuous-time and discrete-time signals. The notion of discrete-time signals is captured by lifting the *range* of continuous-time signals using an option type called `Event`, similar to Haskell's `Maybe` type.[3] This type has two data constructors: `NoEvent`, representing the absence of a value; and `Event`, representing the presence of a value, also called an event *occurrence*. In Haskell notation:

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

A discrete-time signal carrying elements of type α can thus be thought of as a function of type `Signal` (`Event` α).

Yampa provides a rich set of functions for operating pointwise on events. In fact, the type `Event` is abstract in the current Yampa implementation, so events cannot be manipulated except through these operations. Some examples are:

```
tag     :: Event a -> b -> Event b
lMerge  :: Event a -> Event a -> Event a
rMerge  :: Event a -> Event a -> Event a
filterE :: (a -> Bool) -> Event a -> Event a
```

Note that these all are ordinary functions. They have to be lifted (using `arr`) in order to process discrete-time *signals*. The function `tag` tags an event with a new value, replacing the old one. `lMerge` and `rMerge` allow two discrete signals to be merged pointwise. In case of simultaneous event occurrences, `lMerge` favors the left event (first argument), whereas `rMerge` favors the right event (second argument). `filterE`, finally, suppresses events which do not satisfy the Boolean predicate supplied as the first argument. Additionally, `Event` is an instance of *Functor*, allowing `fmap` to be used on events:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Yampa also provides *stateful* signal functions for generating and processing events. Four important examples are:

```
edge :: SF Bool (Event ())
hold :: a -> SF (Event a) a
accumBy :: (b -> a -> b) -> b
           -> SF (Event a) (Event b)
accum :: a -> SF (Event (a -> a)) (Event a)
```

The signal function `edge` generates an event whenever the input changes from `False` to `True`. A signal function whose output signal is of type `Event` α for some type α is called an *event source*. The signal function `hold` converts a discrete-time signal to a continuous-time one by "holding on" to the value carried by the last input event. The signal function `accumBy` processes events by applying a function to incoming events and an internal state. An output event is generated in response to each input event. It is tagged with the result of the function application, which also becomes the new internal state. The signal function `accum` is similar.

2.2.3 Dynamic System Structure

The structure of a Yampa system may evolve over time. These structural changes are known as *mode switches*. This is accomplished through a family of *switching* primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is `switch`:

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

`switch` switches from one subordinate signal function into another when a switching event occurs. The first argument of `switch` is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. Once the switching event occurs, `switch` applies its second argument to the value with which the event is tagged and switches into the resulting signal function.

Note that the switching constructs are what apply a signal function to a signal argument, or, if one prefers, spawns a process, with the exception of the initial top-level signal function that gets started by other means. A signal function in itself is just inert code.[4]

Yampa also includes *parallel* switching constructs that maintain dynamic collections of signal functions connected in parallel. Signal functions can be added to or removed from such a collection at runtime in response to events; see figure 2. The first class status of signal functions in combination with switching over dynamic collections of signal functions makes Yampa an unusually flexible language for describing hybrid systems and sets it apart from typical synchronous dataflow and simulation languages [21].

2.3 Generalized Algebraic Data Types

Generalized Algebraic Data Types (GADTs) have recently been added to the list of Haskell extensions supported by the Glasgow Haskell Compiler (GHC) [26]. The idea of GADTs is not new. A number of variations have been proposed under a number of different names, such as guarded recursive data types and equality qualified types. In particular, GADTs are closely related to inductive families that have been studied in the dependent types community for a very long time. However, this is the first time these ideas have

[3] Note that `Event` is used as the name for both the *type* constructor and a *data* constructor, which is not uncommon in Haskell code. A clearer name for the type constructor might have been `MaybeEvent`, but that, among other problems, is a bit verbose.

[4] This is in contrast to some earlier FRP implementations that didn't make a sharp distinction between signals and signal functions.

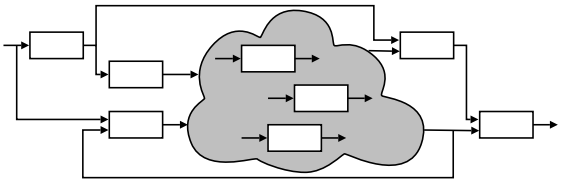


Figure 2. System of interconnected signal functions with varying structure

been made available to a wider audience in the context of a very mature implementation of a reasonably well-established language.

The key idea of GADTs is to allow the type of each data type constructor to be stated separately, and to take that extra type information into account in individual case branches when taking data structures apart, during pattern matching. Let us return to the example from the introduction. GADTs allow us to define a constructor representing `arr id` that has the expected, more specific type `SF a a` as opposed to `SF a b`:

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

But apart from this, there is nothing special with `SFId`, and its type is a perfectly normal Haskell type.

The fun starts when we do case analysis on signal functions. The following fragment of the definition of `>>>`, that reflects arrow law 3 (section 2.1), is exactly as given in the introduction:

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

The difference is that the type checker now is going to accept the definition since it knows that the type of the first argument in case it is constructed using `SFId` must be `SF a a`. That is, the types `a` and `b` *must* in fact be equal, and thus the second argument `sf` in fact has type `SF a c`, which is exactly the return type. The equation is therefore well-typed.

3. Basic Yampa Implementation

This section outlines the implementation of the current version of Yampa. This has been described before [21], and is only included here to make the present paper self-contained. The current Yampa version performs some basic dynamic optimizations, for example, it exploits arrow law 2 (section 2.1), replacing arrow composition by simple function composition, but as noted in the introduction, there are a number of obvious optimizations that were not implemented because the Haskell 98 type system got in the way.

However, before looking at the current version, a very simple implementation that does not do any dynamic optimization at all will be described. This is done in part for explanatory purposes, and in part to provide a baseline for establishing the effectiveness of the basic optimizations, as that up until know had been taken on faith. For example, while replacing arrow composition by simple function composition indeed does seem like an obvious win in a system like Yampa, where arrow composition is an expensive operation compared to function composition, the merit of simplicity should never be underestimated. Simplicity does translate into smaller function definitions, less case analysis, and possibly a bit more compact representation of signal functions, all of which could have a positive impact on the performance. This section is thus concluded by a simple comparison of the performance of the current implementation and the totally unoptimized one.

3.1 The Unoptimized Implementation

The Yampa implementation uses a continuation-based signal function representation, originally inspired by the implementation of the Fudgets graphical user interface toolkit [4]. There are also similarities to the "residual behaviors" implementation of Fran [10].

We start by considering the unoptimized version. Each signal function is essentially represented by a *transition function*. This takes as arguments the amount of time passed since the previous time step and the current instantaneous value of the input signal. It returns a *transition*: a pair of a (possibly) updated representation of the signal function, the *continuation*, along with the current value of the output signal:

```
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. They are assumed to be strictly greater than 0. We will return to the question what the initial time delta should be below.

The function `reactimate` is responsible for animating a signal function. It runs in an infinite loop. At each point in time, `reactimate` reads an input sample and the time from the external environment (typically via an I/O action), feeds this sample value and the amount of time that passed since the previous sampling to the signal function's transition function, and then writes the resulting output sample to the environment (also typically via an I/O action). The loop then repeats, but uses the continuation returned from the transition function on the next iteration, thus ensuring that any internal state is maintained properly.

As a first example of a signal function implementation, let us consider the combinator `arr`:

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

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.

Now let us consider serial composition. Since each of the signal functions being composed could be stateful, we have to make sure to combine their continuations into an 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 the definition corresponds to the intuitive semantics given in section 2.2.1. Also note how the *same* time delta is fed to both subordinate signal functions, thus ensuring synchrony.

When a transition function is invoked for the very first time, there is no meaningful time delta to feed in since there is no prior invocation of that transition function. For that reason, signal func-

tions are actually represented by a slightly simpler transition function that only expects a single argument: the initial input value. Once an initial input sample has been fed in, the signal functions makes a transition to a "running" state, where the transition function has the more general form described above. Thus, the real type definitions are as follows:

```
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)
```

The type `SF'` is an internal type, hidden from the user. The real definitions of `arr` and `>>>` are also slightly more complicated, as the definitions above are internal and really called something else, and what the user sees are essentially wrappers around these internal definitions.

3.2 The Simply Optimized Implementation

We now turn our attention to the current, simply optimized version of Yampa. The central idea is to represent stateless signal functions, i.e., signal functions constructed by `arr`, using a special constructor. This makes it possible to recognize such signal functions for example when composing signal functions, allowing arrow law 2 to be exploited:

```
arr (f >>> g)  =  arr f >>> arr g
```

Additionally, we introduce a special constructor for the signal function `constant`, conceptually defined through

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

This is done because the following laws hold, suggesting what ought to be worthwhile optimizations:

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

A number of useful identities involving `constant` also hold for other arrow combinators, like `first` and `&&&`, and for some Yampa-specific ones like `switch`. The utility of handling `constant` specially thus goes beyond serial arrow composition.

Here is the new definition of `SF'`:

```
data SF' a b
  = SFConst {
      sfTF'  :: DTime -> a -> Transition a b,
      sfCVal :: b
    }
  | SFArr {
      sfTF'  :: DTime -> a -> Transition a b,
      sfAFun :: a -> b
    }
  | SF' {sfTF' :: DTime -> a -> Transition a b}
```

Note that all constructors has a `sfTF'` component. This enables signal functions to be handled uniformly in cases where one do not wish to differentiate between the different kinds.

The internal versions of `constant` and `arr` are then defined as follows:

```
sfConst :: b -> SF' a b
sfConst b = sf
  where
    sf = SFConst {
           sfTF'  = \_ _ -> (sf, b),
           sfCVal = b
         }

sfArr :: (a -> b) -> SF' a b
sfArr f = sf
  where
    sf = SFArr {
           sfTF'  = \_ a -> (sf, f a),
           sfAFun = f
         }
```

Implementing the internal version of `>>>` is straightforward, if a bit tedious:

```
cpAux _ sf2@(SFConst {})   = sfConst (sfCVal sf2)
cpAux sf1@(SFConst {}) sf2 = cpAuxC1 (sfCVal sf1)
                                     sf2
cpAux sf1@(SFArr {}) sf2   = cpAuxA1 (sfAFun sf1)
                                     sf2
cpAux sf1 sf2@(SFArr {})   = cpAuxA2 sf1
                                     (sfAFun sf2)
cpAux sf1 sf2              = SF' {sfTF' = tf}
  where
    tf dt a = (cpAux sf1' sf2', c)
      where
        (sf1', b) = (sfTF' sf1) dt a
        (sf2', c) = (sfTF' sf2) dt b
```

The functions `cpAuxC1`, `cpAuxA1` and `cpAuxA2` are specialized versions of `cpAux` that exploit that the form of the first or second of the arrows being composed is known, and that it thus would be wasteful to pattern match on those over and over again. We give `cpAuxA1` as an example. Note the second equation that implements the optimization suggested by arrow law 2:

```
cpAuxA1 _  (SFConst{sfCVal=c}) = sfConst c
cpAuxA1 f1 (SFArr {sfAFun=f2}) = sfArr (f2 . f1)
cpAuxA1 f1 (SF' {sfTF'=tf2})   = SF' {sfTF'=tf}
  where
    tf dt a = (cpAuxA1 f1 sf2', c)
      where
        (sf2', c) = tf2 dt (f1 a)
```

Recall that Yampa, through the use of switching, allows system with dynamic structure to be described (see section 2.2.3). This means that a signal function at some point could "become" a totally different signal function, perhaps something very simple, like `arr f`. For example:

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

If the above code fragment was used in, say, an arrow composition, it might, at the time of the switch, become possible to eliminate that arrow composition completely, e.g.:

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

Yampa does implement this, as can be seen from the code for composition above (`cpAuxA1`), but it means that the various arrow combinators constantly have to monitor the subordinate signal functions to see if they have "changed shape" in such a way that an optimization has become possible. The dynamic system structure is why optimizations have to be performed dynamically in Yampa, while the system is running. A single optimization phase before the signal processing proper starts would not be good enough. See section 6 for some further discussion.

Figure 3. Screenshot of Space Invaders

3.3 Effectiveness of the Simple Optimizations

From comparing the code of the unoptimized and the simply optimized implementations of Yampa, it should be clear that the simply optimized version is quite a bit bigger, and that it potentially spends more time on case analysis than the unoptimized version since there simply are more cases to consider. All of this could have a negative performance impact, and that price would have to be paid regardless of whether any actual optimization opportunities turn up or not. Thus, as always, the question is if the gains really outweigh the costs? In particular, what is the bottom line in the context of fairly large, realistic applications? Are there sufficiently many optimization opportunities, and are any gains large enough to justify the extra implementation complexity?

To begin answering such questions, a simple, initial experiment was carried out. The performance of two reasonably large and demanding application were measured using the two Yampa implementations to see if any significant performance differences emerged. Obviously, since only two applications are involved, any far-reaching conclusions based on the outcome must be avoided. But a positive outcome should still be reassuring.

The Yampa Space Invaders game [9] was chosen as one of the benchmark application since it was considered representative of typical applications. The screenshot in figure 3 should give an indication as to its complexity. Since the objective was to measure the impact of the optimizations on the execution of signal functions, the game was modified by disabling the graphical rendering as that accounted for a significant part of the overall run time. Further modifications were made to run the game for a fixed number of cycles over a fixed length of simulated time on fixed input.

The other benchmark is a high-level model of a MIDI[5] Event Processor (MEP), patterned after hardware devices like the Yamaha MEP4[6]. Such a device allows a stream of MIDI events (like note on and off) to be transformed to achieve effects like splitting, layering, and arpeggios (by adding delayed messages). Yampa's synchronous nature and event processing capabilities make it very well suited for this type of application. The Yampa MEP was programmed to carry out typical split and layering duties, including adding arpeggiated notes if the velocity attribute of a note-on event indicates that the keyboard key has been played hard.

[5] Musical Instrument Data Interface

[6] `http://www.yamaha.co.jp/manual/english/`

Benchmark	T_{U} [s]	T_{S} [s]	$T_{\mathrm{S}}/T_{\mathrm{U}}$
Space Invaders	0.95	0.86	0.91
MEP	19.39	10.31	0.53

Table 1. Benchmark performance. Averages over five runs.

The measurements were carried out on a 1.6 GHz Pentium laptop running Linux. The speedstep facility was turned off to ensure that the CPU speed was not changed during the measurements. Two versions of the each benchmark were compiled, one using an unoptimized version of Yampa like the one presented in section 3.1, the other using the simply optimized version. The benchmarks were then run with a known initial heap size (8 MByte). The size was chosen large enough so that most time would be spent executing actual Yampa code as opposed to doing garbage collection, but also small enough so that really bad space behavior of either implementation would have an impact. The average execution time was measured over five "good" runs using the Linux `time` command by adding the reported "user" and "system" times. A run was considered good if 90 % or more of the CPU time was spent on executing the benchmarks, indicating that not much else that could skew the results was going on. The GHC runtime system was instructed to print out the time spent on executing code and doing garbage collection. About 5 % of the overall run time was spent on garbage collection in each case. Thus there were no significant differences in this respect.

The results are given in table 1. T stands for total execution time. The subscript U refers to the unoptimized implementation, whereas S refers to the simply optimized one. The results do indeed suggest that the simple optimizations have a substantial positive impact at the system level, in particular the results for MEP where the optimized version runs almost twice as fast. The Space Invaders benchmark involves a lot of numerical floating point and vector computations, and a closer inspection reveals that a large part of the overall time is spent there. Considering this, a 10 % shorter execution time is a fairly good performance improvement.

4. Optimizing Yampa Using GADTs

This section explores how GADTs can be employed to implement dynamic optimizations in a Yampa that go beyond what is possible in plain Haskell 98. General arrow optimizations are considered first, then Yampa-specific ones, particularly improving on event processing.

4.1 Optimizing Pure Arrows

Let us return to the problem of optimizing arrow composition with the identity arrow. One approach was outlined in section 2.3, and integrating that into the current Yampa implementation as described in section 3.2 is mostly a matter of just adding the constructor `SFId` to the type `SF'`. However, when the emerging optimization opportunities are implemented for the various arrow combinators, there is a lot of code duplication related to composition of pure arrows, of which there now are three varieties: `constant`, `identity`, and all others. Additionally, for Yampa-specific optimizations, it is necessary to consider further cases, as will be discussed later.

A better approach is to factor out the description of the functions to be lifted, write a single function for dealing with composition of such descriptions, and finally add a single signal function con-

structor SFArr for pure arrows, that incorporates this description (instead of just a function about which nothing is known).

Using GADTs, a data type describing the three kinds of functions can be defined as follows:

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

FDI represents the identity function id, FDC b represents the function const b, and FDG f represents a general function f. Note how the types of the function descriptions match the types of the functions being described.

Next, a way to recover the described function from a function description is needed:

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

Note how GADTs come into play in the FDI case. The equation is well-typed only because FDI has type FunDesc a a, meaning that types a and b are equal in that case, which in turns makes it legitimate to return id as the result.

Composition of function descriptions can now be defined. Again, note how GADTs come into play wherever FDI is used.

```
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)
```

It is also worth to note how pleasingly natural and direct the definition is.

4.2 Optimizing Arrow Composition

Arrow composition is associative. Thus it would be desirable if optimization of composition was performed as well as possible regardless of the bracketing. This is particularly important when arrow notation [22] is being used since the user then does not always have control over the bracketing. However, the current Yampa implementation is not that well-behaved. For example, if sf is some general signal function, then

```
(arr f >>> arr g) >>> sf
```

would be optimized to

```
arr (g . f) >>> sf
```

eliminating one costly arrow composition, but

```
arr f >>> (arr g >>> sf)
```

would not be optimized, except that the implementation of >>> is slightly more efficient for composition with a pure arrow (see section 3.2).

To address this problem, a constructor representing arrow compositions of the form

```
arr f >>> sf >>> arr g
```

is introduced. The idea is due to Hughes [19]. The point is that composition becomes observable, allowing re-bracketing if necessary, and that pure arrows to either side always can be "absorbed".

Here is the new definition of SF'. SFArr is the single constructor for pure arrows that was discussed in the previous section. SFCpAXA represents pre and post composition with pure arrows:

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

Note that a signal function constructed using SFCpAXA includes the representations of the three subordinate signal functions. This is what makes it possible to change the bracketing structure. However, they play no direct role when making a transition: that is the sole responsibility of the transition function, as before. It will construct an updated version of the composed signal function, where the two subordinate pure arrows of course are the same as before, but where the signal function "in the middle" possibly has been updated.

Let us now look at the implementation of composition. Like in the simply optimized implementation, unnecessary construction and destruction of signal function representations is avoided by having specialized functions dealing with cases where one or more of the subordinate signal functions is something simple, like a pure arrow, constant, or even identity. However, there are now more cases to consider, and we will thus only look at some fragments of the definition to convey the general ideas.

We begin with composition of two arbitrary signal functions. A naming convention is used where X stands for arbitrary signal function (could be anything) and A stands for an arbitrary pure arrow. Thus the function dealing with composition of arbitrary signal functions is called cpXX, and the first few lines of its definition are as follows:

```
cpXX :: SF' a b -> SF' b c -> SF' a c
cpXX (SFArr _ fd1) sf2           = cpAX fd1 sf2
cpXX sf1           (SFArr _ fd2) = cpXA sf1 fd2
```

As can be seen, if it turns out that either the first or the second argument is a pure arrow, one of two dedicated functions that are optimized for those cases are invoked to carry out further analysis.

Here is the definition of cpAX that handles the case where the first argument is a pure arrow.

```
cpAX :: FunDesc a b -> SF' b c -> SF' a c
cpAX FDI      sf2 = sf2
cpAX (FDC b)  sf2 = cpCX b sf2
cpAX (FDG f1) sf2 = cpGX f1 sf2
```

If the first argument is the identity function, then the entire composition can be immediately optimized to just sf2. GADTs are needed to make this case well-typed. Otherwise, further processing is delegated to functions that analyze the second argument to spot further optimization possibilities and, if none is found, are designed to implement composition as efficiently as possible given the knowledge they have about the first argument (constant or a pure function).

If two signal functions constructed by SFCpAXA are composed, then a re-bracketing step is performed with the aim of fusing the

two pure arrows in the middle. Algebraically, we have:

```
(arr f >>> sf1 >>> arr g)
>>> (arr f' >>> sf2 >>> arr g')
=
arr f
>>> ((sf1 >>> arr (f' . g)) >>> sf2)
>>> arr g'
```

The code is as follows:

```
cpXX (SFCpAXA _ fd11 sf12 fd13)
     (SFCpAXA _ fd21 sf22 fd23) =
  cpAXA fd11
        (cpXX (cpXA sf12 (fdComp fd13 fd21))
              sf22)
        fd23
```

Finally, if no optimizable case has been detected, the basic composition is performed. Note that the updated representation of the composed signal function is constructed recursively using `cpXX`, ensuring that `sf1'` and `sf2'` will be analyzed during the next time step to detect any emerging optimization possibilities.

```
cpXX sf1 sf2 = SF' tf
  where
    tf dt a = (cpXX sf1' sf2', c)
      where
        (sf1', b) = (sfTF' sf1) dt a
        (sf2', c) = (sfTF' sf2) dt b
```

Other arrow combinators are optimized in similar ways. For example, `first identity` is optimized to `identity`. More advanced optimizations involving e.g. `first` *and* `>>>` have not yet been attempted. For instance, Hughes identify optimizing `first f >>> first g` to `first (f >>> g)` as troublesome due to the type system *really* getting in the way. GADTs should solve this. The main difficulty is structuring the code in a way that keeps the number of cases that have to be considered manageable.

4.3 Optimizing Event Processing

As was discussed in section 2.2.2, Yampa deliberately blurs the distinction between continuous-time signal and discrete-time signals by layering the latter on top of the former through the `Event` option type. This is quite convenient from a programming perspective, but does mean that it is hard to implement pure event-processing as efficiently as one could hope for. The involved signal functions gets invoked at every single time step, even though everything remain constant between events. Thus, if events are relatively sparse, the overhead is considerable. In some earlier FRP implementations, where discrete and continuous signal functions were separate concepts, there was greater scope for optimization. In this section, we will see how GADTs allow some optimizations of event-processing to be reintroduced.

Consider the following composition of pure arrows:

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

arr f >>> arr g
```

Normally events occur relatively sparsely. Thus, for the most part, the output from `f` is going to be `NoEvent`, and the result of `g` applied to `NoEvent` is going to be computed over and over again. That seems a bit wasteful. If the assumption that events occur sparsely holds, it would be better to compute `g NoEvent` only once, and reuse that value whenever the output from `f` is `NoEvent`. The function `g` would then only have to be invoked on event occurrences.

This can be achieved by introducing a function description for event-processing functions, allowing the composition of such functions to be handled specially. However, instead of representing functions of type `Event a->Event b` as the example above would suggest, we chose to represent functions of type `Event a->b` to cover more functions. Nothing is lost by doing this: if two such functions are composed, the result type of the first function must in fact be `Event`, and GADTs allow that fact to be exploited.

The type `FunDesc` from section 2.2.2 is thus extended as follows:

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

The first argument to `FDE` is the event-processing function in question. The second argument is the result of applying that function to `NoEvent`. The function `fdComp` is then extended to handle compositions involving event-processing functions:

```
fdComp :: FunDesc a b -> FunDesc b c
          -> FunDesc a c
...
fdComp (FDE f1 f1ne) fd2 = FDE (f2 . f1)
                                (f2 f1ne)
  where
    f2 = fdFun fd2
fdComp (FDG f1) (FDE f2 f2ne) = FDG f
  where
    f a = case f1 a of
            NoEvent -> f2ne
            f1a     -> f2 f1a
```

The second equation reflects the discussion above. Note how `f2` only gets invoked on events. Also note the crucial role played by GADTs for making the code well-typed. In particular, the type refinement due to GADTs is what makes the case analysis of `f1 a` possible.

The only question now is how to get `FDE`s into play. The other two special function descriptions, `FDI` and `FDC`, are introduced by providing the special signal functions `identity` and `constant` in the Yampa API, and asking the user to use those in preference to writing `arr id` and `arr (const x)`. Similarly we could introduce a special version of `arr`, say `arrE :: Event a -> b`, and ask the user to use that wherever possible. However, while not ideal, `identity` and `constant` work because they are fairly natural signal functions to have in the API anyway, and because it is only those two. Asking the user to remember to use `arrE` where possible is a bit too much. Moreover, when the arrow notation is used, many `arr`s get introduced by the *translation* into plain Haskell, outside the control of the user.

One way to resolve this dilemma is to employ a GHC-specific trick. A *rewriting rule* is specified that instructs the *compiler* to rewrite `arr f` to `arrE f` whenever the type of `f` is `Event a -> b`[7]. The rule is remarkably simple, exploiting that the GHC rewriting rules only apply when the types match:

```
{-# RULES "arrPrim/arrEPrim"
          arrPrim = arrEPrim #-}
```

This solution is good because it is totally transparent to the user. However, the price paid is that the Yampa implementation gets tied even harder to GHC. An alternative might be to use the class system, but the only solution in that direction explored by this author necessitated changing the `Arrow` class slightly and enabling both

[7] Note that GHC rewrite rules could not be used to implement the optimizations described in this paper in general since the optimizations have to be carried out dynamically as explained in section 3.2.

overlapping and incoherent instances. Altogether not very appealing. In the worst case, should rewriting rules not be available, all is not lost since optimization of *stateful* event processing, described in the following, only relies on GADTs.

Event-based optimization in the spirit above can also be implemented for *stateful* event-processing signal functions like `accumBy` and `hold`. Both of these, and many other Yampa event processors, can be seen as instances of a general stateful event-processing signal function that we can call `ep`:

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

The argument of type `c` is the initial sate, the argument of type `b` is the initial quiescent output (when the input is NoEvent), and the function argument is the function that gets invoked on the value of any incoming event and the current state to compute the updated state, the output at the point of the event, and the new quiescent output. For example, `hold` can be defined as follows:

```
hold :: a -> SF (Event a) a
hold a_init = ep f () a_init
  where
    f _ a = ((), a, a)
```

The point of this is that composition of `ep` with `ep` or with pure arrows, both event-processing ones and others, results in a signal function that can be expressed in terms of a single `ep`, in a manner similar to the composition of pure, event-processing arrows, thus allowing an arrow composition to be optimized away. The result is that "pipelines" of event-processing signal functions execute very efficiently since the latter signal functions in the pipeline only get invoked on events. All that is needed to achieve this is to introduce a constructor that represents `ep`, and extend the implementation of arrow composition accordingly:

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

Note that GADTs are again needed here. The code for composition is in many ways very similar to the stateless case discussed above. The details are omitted.

4.4 Optimizing Simple Stateful Signal Processing

Many simple, stateful *continuous* signal functions, like `edge` or `pre` (an "infinitesimal" delay), can be expressed in terms of a common underlying primitive signal function in much the same way as the stateful event processors discussed above. If that underlying signal function is designed in such a way that it composes nicely with pure arrows, stateful event processors, and itself, this opens up many opportunities for optimization. There is a quite large design space here. The current design employs a function of type

```
c -> a -> Maybe (c, b)
```

to compute the new internal state and the output given the present internal state an input. If the result is `Nothing`, this indicates that both the state and output should stay as they were. The new constructor is called `SFSScan`:

```
data SF' a b where
  ...
  SFSScan ::
    (DTime -> a -> Transition a b)
    -> (c -> a -> Maybe (c, b)) -> c -> b
    -> SF' a b
```

Benchmark	T_{S} [s]	T_{G} [s]
1	0.41	0.00
2	0.74	0.22
3	0.45	0.22
4	1.29	0.07
5	1.95	0.08
6	1.48	0.69
7	2.85	0.72

Table 3. Micro benchmark performance. Averages over five runs.

Note that GADTs are not absolutely essential for expressing `SFSScan` in itself: existential quantification could have been used. However, GADTs are needed when coding composing with e.g `SFEP`. The details of coding the various compositions are omitted, but again similar to what has been discussed before.

5. Evaluation of the GADT-based Optimizations

In the previous section we saw how GADTs made possible a number optimizations, both general and Yampa-specific ones. However, we also saw that the size and complexity of the implementation grew. That could offset any gains from the optimizations. This section presents some benchmark results in an attempt to determine whether the optimizations are worthwhile.

Extensive testing was conducted on a number of "micro benchmarks" designed to evaluate whether optimizations based on the arrow laws, event processing, and composition of simple stateful signal functions worked as intended. A few representative ones are considered in the following: see table 2. The procedure and testing conditions were as described in section 3.3, except that the benchmarked implementations now are the simply optimized one (Yampa$_{\mathrm{S}}$) and the one with GADT-based optimizations (Yampa$_{\mathrm{G}}$). The benchmarks were executed for a fixed simulated time period. Appropriate input (continuous or events) was generated using suitable signal functions. However, the time for generating the input was excluded from the benchmark times in order to highlight how much *longer* the execution times became when the input was processed through the benchmarks of table 2.

The average execution times for the micro benchmarks are given in table 3. T_{S} and T_{G} are the execution times for Yampa$_{\mathrm{S}}$ and Yampa$_{\mathrm{G}}$ respectively. The amount of time spent on garbage collection was monitored, but it was small in all cases. The measured times thus represent time spent on executing Yampa code.

As can be seen from the times for benchmark 1, Yampa$_{\mathrm{G}}$, as expected, succeeds in eliminating the overhead of composition with identity completely. Studying benchmarks 2 and 3, we see that Yampa$_{\mathrm{G}}$ is insensitive to parenthesization order, whereas Yampa$_{\mathrm{S}}$ is not (`sf` is the input generator and is not included in the measured times). Moreover, Yampa$_{\mathrm{G}}$ appears to be a little faster.

In the case of benchmarks 4 to 7, one should keep in mind that the implementations of the involved stateful signal functions are very different in Yampa$_{\mathrm{S}}$ and Yampa$_{\mathrm{G}}$. While it certainly is good that Yampa$_{\mathrm{G}}$ is a *lot* faster here, the most important is to study how the execution times change as further signal functions are added to a pipeline. Comparing the times for benchmarks 4 and 5 shows that the overhead of adding an additional stateful event processor (here `accumBy`) to an event processing pipeline in Yampa$_{\mathrm{G}}$ is almost negligible. Similarly, the results for benchmarks 6 and 7 show that the overhead of adding stateless and stateful event processing signal functions to some stateful continuous-time signal processing is also very small. GADTs have thus enabled efficient fusing of a variety of stateful and stateles signal functions. However, it should

Benchmark	Code	Description
1	`identity >>> identity`	Composition of identity
2	`(sf >>> arr (*1)) >>> arr (*2)`	Left-biased parenthesization
3	`sf >>> (arr (*1) >>> arr (*2))`	Right-biased parenthesization
4	`accumBy count >>> hold 0`	Stateful event processing
5	`accumBy count >>> accumBy (+) 0 >>> hold 0`	Stateful event processing
6	`arr (>) >>> edge`	Stateful and stateless signal processing
7	`arr (>) >>> edge >>> arr (‘tag‘ (+1)) >>> accum 0`	Stateful and stateless signal and event processing

Table 2. Micro benchmarks.

Benchmark	T_{S} [s]	T_{G} [s]	$T_{\mathrm{G}}/T_{\mathrm{S}}$
Space Invaders	0.86	0.88	1.02
MEP	10.31	9.36	0.91

Table 4. Application benchmark performance. Averages over five runs.

be pointed out that the event density was very low in these experiments. The results may thus be a bit too optimistic.

Anyway, these results are important. The present Yampa API contains a number of "pre-composed" signal functions, such as `accumHoldBy` and `edgeTag`, implemented as primitives for reasons of efficiency. However, benchmark results 4 to 7 indicate that there is no need for such primitives in Yampa$_{\mathrm{G}}$. The GADT-based optimizations make it is possible to simplify both the Yampa implementation, and, more importantly, the Yampa API, in this respect.

The above results are good, but perhaps not wholly unexpected: the benchmarks are small and set up in a way that let the optimizations have maximal impact. What about the effect on larger application like Space Invaders and MEP (see section 3.3)? Table 4 shows the results. As can be seen, the execution times for Space Invaders are almost exactly the same. This application does quite a bit of floating-point and vector computations, and a closer inspection reveals that that is where most time is spent. After the basic optimizations as implemented in Yampa$_{\mathrm{S}}$ have been performed, there just is not so many more (easy) optimization opportunities. This is a bit disappointing, but at least it suggests that the size and complexity of Yampa$_{\mathrm{G}}$ does not have any major negative impact.

The MEP fares better, showing a worthwhile speedup. Of course, one would hope that, given that this is an event-centered application and the effort put into optimizing event processing. However, it should be emphasized that the MEP is non-trivial, involving many stateful signal functions that cannot be expressed in terms of the composable primitives. It is *not* the case that the MEP trivially reduces to just a single or a few signal functions.

6. Related Work

Dynamic optimization of domain-specific embedded languages it not new. This was briefly discussed in the introduction, one example being Swierstra's and Duponcheel's self-analyzing parser combinators [29]. Optimization in FRP or FRP like systems has also been attempted a number of times, e.g. various implementations of Fran [10]; Pan, a *compiled* embedded language implementation [11]; the highly optimized FranTk implementation, employing `unsafePerformIO` behind the scenes [28]; and, of course, the current Yampa implementation [21].

However, most closely related to the optimization approach presented here is Hughes' work on self-optimizing arrow-based combinator libraries [19]. There are two parts to Hughes' approach. First, a generalized optimization framework for arrows based on arrow transformers is introduced. This allows any arrow instance to be transformed into a self-optimizing version utilizing the arrow laws. Moreover, the run-time overhead is small, since the optimizations are carried out once and for all up front: the transformed arrow is run to produce an optimized untransformed arrow. Second, Hughes makes clever use of (commonly implemented) extensions to the Haskell 98 type system to allow optimizing e.g. composition with the identity arrow. Now that GADTs are available, they could have been used instead, allowing more optimizations, but at the cost of being tied to GHC for the time being. This is no surprise: both Hughes and Nilsson et al. [21, p. 62] note that some form of dependent types is what really is wanted in this context.

The main question then is why the idea of an optimizing arrow transformer was not used for Yampa? That would arguably have been more elegant than optimizing the the individual arrow combinator instances individually. There are two reasons for this choice. First, optimizations of event processing is very important but rather Yampa-specific. There does not seem to be much point in inventing a Yampa-specific arrow transformer for transforming the Yampa arrow. A layered approach might have been possible, where the basic Yampa arrow first was transformed using Hughes' general optimizing arrow transformer, and then further transformed by an arrow transformer addressing only event processing. That does seem rather complicated, however. The second reason is that Yampa needs to consider optimization at *every time step* since Yampa allows systems with dynamic structure to be described. This was discussed in section 3.2, where an example showing how a signal function eventually might "become" something simple, like a pure arrow, was given, at which point one would like any optimizations that this enables to be carried out. The overhead of an arrow transformer approach can thus not be reduced away once and for all, and the approach simply seemed too costly.

7. Conclusions

This paper showed how GADTs make possible a number of useful dynamic optimizations for Yampa, an arrow-based, domain-specific language for Functional Reactive Programming. Some of these optimizations are applicable for arrow-based libraries in general, whereas others are specific to Yampa. A set of small benchmarks showed that these optimizations can boost the performance significantly in some cases. At the level of of complete applications, the gains ranged from modest but worthwhile to none.

However, the optimization possibilities have not yet been exhausted. In particular, optimization of combinations of serial (`>>>`) and parallel (`first`, `***`, ...) arrow composition has not yet been attempted. GADTs are expressive enough to cope with such extensions: the difficult part is to manage the combinatorial growth of cases for optimization that need to be considered.

All in all, despite the lack of major system-wide performance gains, and without taking the possible impact of further optimizations into account, the GADT-based Yampa implementation is still an improvement on the old one. The gains at the code fragment

level are important as they allow a more concise and principled API, without "pre-composed" signal functions that are there only for performance reasons. The GADT-based implementation is also able to exploit that arrow composition is associative, making it insensitive to how arrow composition is parenthesized. The measurements thus far suggest that these benefits were obtained without an overall negative performance impact that could have been the result of the general increase of the size and complexity of the core of the Yampa implementation.

On a general note, GADTs were found to be intuitive to use and pleasingly expressive. Among other applications, they should be quite useful for dynamic optimizations in a wide range of embedded domain-specific languages. At present, they do, however, have a bit of a "bolted on" feel. For instance they cannot be used together with Haskell's labeled-field notation (compare the stylistic difference between the definitions of `SF'` in sections 3.2 and 4.2 for an example).

Acknowledgments

The author would like to thank the anonymous reviewers for their thorough and constructive comments.

References

[1] Arthur I. Baars and S. Doaitse Swierstra. Type-safe, self inspecting code. In *Proceedings of the 2004 ACM SIGPLAN Haskell Workshop (Haskell'04)*, pages 69–79. ACM Press, 2004.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):217–248, 1992.

[3] Magnus Carlsson and Thomas Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 321–330, Copenhagen, Denmark, June 1993. ACM Press.

[4] Magnus Carlsson and Thomas Hallgren. *Fudgets – Purely Functional Processes with Applications to Graphical User Interfaces*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 1998.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[6] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Submitted for publication, 2000.

[7] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL 2000), Aachen, Germany, September 2000*, volume 2011 of *Lecture Notes in Computer Science*, pages 176–193. Springer-Verlag, 2001.

[8] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.

[9] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[10] Conal Elliott. Functional implementations of continuous modelled animation. In *Proceedings of PLILP/ALP '98*. Springer-Verlag, 1998.

[11] Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation (SAIG 2000)*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–27, Montreal, Canada, September 2000. Springer-Verlag.

[12] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous datafbw programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[14] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logics in Computer Science (LICS 1996)*, pages 278–292, 1996.

[15] Paul Hudak. Modular domain specifi c languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[16] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[17] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[18] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[19] John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. To be published by Springer Verlag in their LNCS series.

[20] The MathWorks, Inc. *Using Simulink Version 4*, June 2001. `http://www.mathworks.com`

[21] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[22] Ross Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, September 2001.

[23] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conference on Robotics and Automation*, May 1999.

[24] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of PADL'99: 1st International Conference on Practical Aspects of Declarative Languages*, pages 91–105, January 1999.

[25] John Peterson, Paul Hudak, Alastair Reid, and Greg Hager. FVision: A declarative language for visual tracking. In *Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages*, pages 304–321, January 2001.

[26] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalized algebraic data types. Submitted to POPL'05, July 2004.

[27] Marc Pouzet, Paul Caspi, Pascal Couq, and Grégoire Hamon. Lucid Synchrone v2.0 – tutorial and reference manual. `http://www-spi.lip6.fr/lucid-synchrone/lucid_synchrone_2.0_manual.ps`, April 2001.

[28] Meurig Sage. FranTk: A declarative GUI system for Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, September 2000.

[29] S. D. Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.

[30] Philip Wadler. How to replace failure with a list of successes. In *Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1985.