Undergraduate Research Opportunity Program

(UROP) Project Report

Investigation of Design and Implementation of Domain-Specific Languages

By

Xu Jun

Department of Computer Science

School of Computing

National University of Singapore

AY2018/2019

Undergraduate Research Opportunity Program

(UROP) Project Report

Investigation of Design and Implementation of Domain-Specific Languages

By

Xu Jun

Department of Computer Science

School of Computing

National University of Singapore

AY2018/2019

Project No: U018130

Advisor: A/P Chin Wei Ngan

Deliverable:

Report: 1 Volume

Abstract

An exploratory investigation is carried out in the field of domain-specific languages (DSLs from here onwards). DSLs are languages tailored for application domains and they often require special, idiomatic abstractions for particular problems. In pure functional programming, Arrow is a construct that structures code in a modular and disciplined manner, and by that virtue could potentially be used to build application-specific abstractions. This project aims to uncover how Arrow can be applied in the domains of game programming and hardware design. Firstly, it evaluates from a programmer's perspective whether Arrow provides a suitable abstraction for the problem. Secondly, some domain-specific issues surrounding Arrow are discussed from an implementer's perspective.

Subject Descriptors:

D.1.1 Applicative (Functional) Programming

D.3.3 Language Constructs and Features

Keywords: Arrow, abstraction, space leak, games, hardware description language (HDL)

Implementation Software:

Haskell Platform, Macintosh OS, Ubuntu, Emacs, Vim

Table of Contents

Chapter 1 INTRODUCTION

This chapter introduces the necessary background. In particular, it first motivates Arrow for functional programming. Subsequently, it discusses pattern matching, which is needed for chapter 2. Lastly, it discusses hardware design, which is necessary for chapter 3.

The rest of the report is organized as follows. Chapter 2 discusses the case of Arrow in game programming. Chapter 3 discusses the case of Arrow in hardware design. Chapter 4 concludes.

1.1 Motivating Arrow

Arrow is mathematically rooted in category theory. In functional programming, it was first employed by Hughes (Hughes, Generalising monads to arrows, 2000) as a generalization to the Wadler's Monad interface (Wadler, 1995).

Similar to Monad, Arrow is a *combinator*. A *combinator* is a construct that enables clean and elegant composition of computation. It specifies a set of combination operations, which act like an interface during program construction. A computation could be a pure function, or it could model some effects. Therefore, Monad and Arrow as combinators enable the handling of effects in a structured and disciplined manner, which is of great benefit to the design of software libraries in pure functional languages.

Let us take the example of parsing. A parsing computation could be modeled as a function that takes an input string and outputs a pair comprising the parsed result and the remaining input. A simple definition appears below, where the keyword *newtype* basically means "defining the following function as a data type", which in this case is a type called Parser and takes as argument the type of parsed token *a*. The keyword *Maybe* is a data type that encapsulates error:

$$data\ Maybe\ a = Nothing\ |\ Just\ a$$

$$newtype\ Parser\ a = String \rightarrow Maybe\ (a, String)$$

It is simple to combine two parsers in the following way:

$$combine\ p1\ p2 = \ \lambda input \rightarrow case\ p1\ input\ \text{of}$$

$$Nothing \rightarrow Nothing$$

$$Just\ (a, rest) \rightarrow p2\ a$$

But this way of combining parsers is rather ad-hoc and arbitrary. Also, for larger grammars different combinations of parsing are going to be required, such as choice and iterate. This presents a challenge to keeping code clean and modular. To solve this problem, Monad gives a unified interface:

$$class\ Monad\ m\ where$$

$$return :: a \rightarrow m\ a$$
$$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

```
$instance\ Monad\ Parser\ where$
$return\ v = \lambda s \rightarrow Just\ (v, s)$
$p1 \gg= f = \lambda s \rightarrow case\ p1\ s\ of$
            $Nothing \rightarrow Nothing$
            $Just\ (v, s') \rightarrow\ f\ v\ s'$
```

We could implement our parser as an instance of Monad and subsequently implement a host of parser combinators in terms of ($\gg=$) and *return*. Using those combinators recursive descent parser can be written in a very intuitive manner. Interested readers can find out more about Monadic parsers in Wadler's paper (Wadler, 1995).

The particular combinator that led to problem was alternation, which selects from a pair of parsers. It runs the first parser, and returns the result if successful; otherwise, it runs the second parser on the original input.

```
$alternate\ p1\ p2 = \lambda s \rightarrow case\ p1\ s\ of$
            $Just\ (v, s') \rightarrow\ Just\ (v, s')$
            $Nothing \rightarrow p2\ s$
```

We are faced with a space leak arising from backtracking. String *s* cannot be garbage collected as *p1* consumes it and a pointer must be kept at the starting position in case *p1* fails and *p2* needs to be tried. The well-known solution to this problem is LL(1) parsing, which yields a new parser type incorporating first sets:

$$newtype\ Parser\ a = P\ [Token]\ (String \rightarrow Maybe\ (a, String))$$

However, LL(1) parser does not fit into the Monad interface. In particular, ($\gg=$) cannot be implemented because its type signature does not allow computation of first set of the composed parser. We illustrate this problem by attempting to implement ($\gg=$) below.

```
$(P\ fst1\ p1) \gg= f = P\ (\ldots)$                    ... (0)
        $(\lambda s \rightarrow case\ fst1\ s\ of.$          ... (1)
            $Nothing \rightarrow Nothing$ ... (2)
            $Just\ (v, s') \rightarrow f\ v\ s')$ ... (3)
```

We cannot determine the first set at line 0 because it depends on the first sets of both *p1* and the second parser returned by function *f*, which is not available until we run *p1* (in line 1) and supply the result to *f* (in line 3) first.

If we do not want to abandon the Monad interface, the only way forward is to extend it. This motivated Hughes to generalize Monad to Arrow:

$$\begin{aligned}&class\ Arrow\ a\ where\\&arr :: (b \to c) \to a\ b\ c\\&(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d\end{aligned}$$

The Arrow composition $(\ggg)$ then allows us to combine LL(1) parser as follows. The first set of the overall parser is just the union of two first sets, if p1 reduces to epsilon; otherwise, it is just the first set of p1.

```
$(P\ fst1\ p1) \ggg (P\ fst2\ p2) =$
  $let\ fst_{overall} = if\ fst1.contains(\varepsilon)\ then\ fst1\ \cup fst2\ else\ fst1$
  $in\ P\ fst_{overall}$
    $(\lambda s \to case\ p1\ s\ of$
        $Nothing \to Nothing$
        $Just\ (v, s') \to p2\ s'\ )$ ... (6)
```

Everything is well and good except we lose the ability to pass the result of the first parse to the second (in line 6). In Monad, this was easily achievable because a lambda expression was prescribed to bind the result to scope. We would like to preserve this functionality because the overall parse result needs to be returned. This should be done by each constituent parser accumulating on the overall result, thus a way to pass information from one Arrow to the next is needed. A natural solution is to extend the parser type with an extra parameter, where *a* contains the result passed from the previous parse:

$$newtype\ Parser\ a\ b = P\ [Token]\ \big((a, String) \to Maybe\ (b, String)\big)$$

$$(line\ 6\ in\ the\ definition\ of\ \ggg) \ldots Just\ (v, s') \to p2\ (v, s')$$

We now have an almost complete type signature for Arrow. Let us add the last utility method, *first*, to complete the picture. *first* alters the information passing between Arrows, in that it creates an additional channel by extending the input and output types to tuples. What it does is convert an existing Arrow into one which acts upon the first component of its input and leaves the second unchanged. This is useful for obvious reasons, such as passing constants across a computation, but it also enables additional functionalities such as dynamic choice. This will be elaborated in the next section.

$$\begin{aligned}&class\ Arrow\ a\ where\\&arr :: (b \to c) \to a\ b\ c\\&(\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d\end{aligned}$$

$$first :: a\ b\ c \rightarrow a\ (b, d)\ (c, d)$$

1.2 Arrow Operations

There are a variety of ways Arrow can be composed, some of which can be defined in terms of *first*, while others require knowledge of the underlying arrow instance. It is advisable to view these operations abstractly for the moment, as they will be given a more concrete meaning in later chapters.

1.2.1 Working with Pairs

second is the dual of *first*, converting an Arrow to work on the second component of a tuple.

$$second\ f = arr\ \big(\lambda(a,b) \rightarrow (b,a)\big) \ggg first\ f \ggg arr\ \big(\lambda(a,b) \rightarrow (b,a)\big)$$

We can also combine two arrows to work on both components of a tuple.

$$(***) :: Arrow\ a => a\ b\ c \rightarrow a\ d\ e \rightarrow a\ (b,d)\ (c,e)$$

$$f *** g = first\ f \ggg second\ g$$

A single input can also be duplicated to two channels and processed separately.

$$(\&\&\&) :: Arrow\ a => a\ b\ c \rightarrow a\ b\ d \rightarrow a\ b\ (c,d)$$

$$f\ \&\&\&\ g = arr\ \big(\lambda b \rightarrow (b,b)\big) \ggg (f *** g)$$

1.2.2 Dynamic Choice

We sometimes need to run arrows conditionally. For example, an input needs to go to one arrow when it satisfies some condition, and another arrow otherwise. This can be achieved with Haskell's sum type, which basically means "either this or that".

$$data\ Either\ a\ b = Left\ a\ |\ Right\ b$$

Dynamic choice could then be implemented to treat Left and Right values differently.

$$(|||) ::\ a\ b\ d \rightarrow a\ c\ d \rightarrow a\ (Either\ b\ c)\ d$$

$$f\ |||\ g = arr\ (\lambda a \rightarrow case\ a\ of$$

$$Left\ l \rightarrow \cdots$$

$$Right\ r \rightarrow \cdots)$$

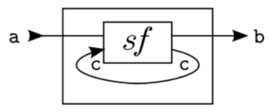
The exact implementation is dependent on particular arrow instances.

1.2.3 Loop

Another operation is recursive arrows. It has the following type signature:

$$loop :: Arrow\ a => a\ (b, d)\ (c, d) \to a\ b\ c$$

Again, its implementation is instance-dependent. In the case of game programming, it has two important benefits. First, it is used when two Arrows' results are mutually dependent and are fed back as each other's inputs. Second, it is used to compute recursive signals. Both will be elaborated in chapter 2. As a whole, the composed arrow becomes one whose output is fed back to input, as shown in the following diagram:



loop sf

1.3 background on pattern matching and semantics

Pattern matching is a very useful construct in functional programming. Very often, we want to check if the argument passed into a function is of a certain type, or of a certain value. For example, we do the following when we want to know if a list argument is empty:

```
f list = if (list.isEmpty()) then … else …
```

This is very cumbersome and ad-hoc, because we need to manually keep track of argument's type and constructor and provide the necessary methods for every type. Pattern matching is an elegant technique that completely solves this mess. Consider the again the list example. Now we use pattern match to 1) check if the argument is indeed a list and 2) extract the values from that list.

```
f [] = …
f (x:xs) = …
```

This is much cleaner and more elegant.

In general, pattern matching works as follows.

```
f p11 p12 …p1n = …
f p21 p22 … p2n = …
f pm1 pm2 … pmn = …
f v1 v2 … vn =
```

n is an arbitrary number of arguments a function could have. m is the maximum number of data constructors of all the arguments' types. We have m branches and we match every argument in a function call against each corresponding pattern, iterating through all the branches sequentially. A match can yield three results: success, failure and diverge. Success and failure are easy to comprehend. Divergence occurs when the value evaluates to semantic bottom, which operationally could mean exception or infinite loop.

A pattern match success happens when all the values matches all the patterns in one branch. Evaluation will proceed with this branch. If any of the patterns fail to match the corresponding value, move on to the next branch. If any of the values diverges, the whole function diverges. Divergence is formally denoted by ⊥, which reads "bottom".

A pattern is strict if it requires any evaluation of the value to be matched. This is expressed formally as follows. If the overall result is ⊥ when one of the arguments is ⊥, then the pattern is strict on that argument. If a pattern is non-strict, it means the corresponding value needs not be evaluated to be matched against it. This basically means anything matches. The two typical cases of non-strict pattern are variable, which introduces a binder to the value, and wildcard "_" which typically denotes "all others" in the last branch, which is akin to "default" in case expressions of other programming languages.

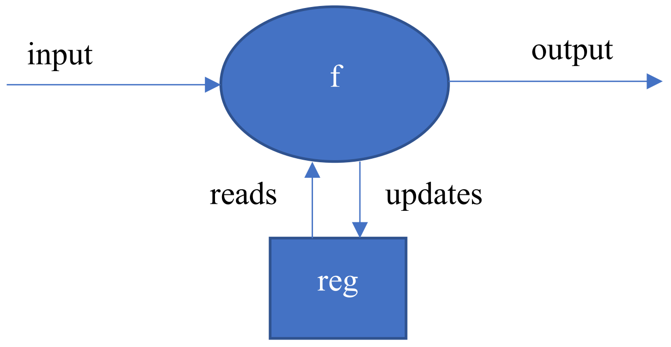
$$f\ p_1\ p_2 \ldots p_n = \cdots$$
$$f\ is\ strict\ on\ p_i\ if\ at\ a\ function\ call\ f\ v_1 \ldots v_i \ldots\ v_n,$$
$$f\ v_1 \ldots v_i|_{v_i=\bot}\ v_n = \bot$$

There are different degrees of strictness. Total strictness means the argument must be evaluated to a value until it is matched against a pattern. A value simply means normal form, or something that cannot be evaluated further. Examples of values are primitive numbers and lambda abstractions. Head-strict means the argument must be evaluated to Weak Head Normal Form (WHNF). All values are in WHNF, plus all data constructors. The extra degree of tolerance comes from data constructors, as WHNF does not require expressions inside data constructors to be evaluated. A typical example would be `[1+2,3]`, whereby arithmetic expression inside a list is left unevaluated.

Haskell is typically head-strict because of its recursive data types. Programs tend to use a constructor pattern to force evaluation of some expression to constructor, and then use variables to match data within the constructor; variable patterns are non-strict. A common example is list processing, where we use `(x:xs)` to match a non-empty list whose head is bound to `x` and tail is bound to `xs`.

1.4 Background on Hardware Design

Any piece of hardware comprises clock-controlled storage units (registers) and combinational logic (which can be visualized as pure function in logic gates). Registers are updated at each clock edge. The basic architecture appears in the following diagram for a mealy machine:



Registers represent a state that is read in every iteration and updated in the next iteration. Verilog and VHDL are two predominant Hardware Description Languages (HDLs) used in industry. Here we take Verilog as an example. Verilog supports combinational logic with asynchronous procedural block and continuous assignments. An example of an adder is as follows:

```
Module Adder where
      Input [31:0] A;
      Input [31:0] B;
      Output [31:0] out;
      Assign out = A + B;
      /* alternatively:
      always@(*) begin
            Out = A + B;
      End
      */
Endmodule
```

Continuous assignments and asynchronous procedural blocks are essentially the same and will not be elaborated further.

As for synchronous logic, Verilog has synchronous procedural blocks. An accumulator of a running sum is written as follows:

```
Module Accum where
      Input Clk;
```

```
        Input reset;
        Input [31:0] A;
        Output reg [31:0] sum;

        always@(posedge Clk) begin
              if(reset)
                    sum <= 0;
              else
                    sum <= sum + A;
        End
Endmodule
```

The `<=` operator is for concurrent assignment. All such assignments are scheduled for concurrent update at the clock edge rather than updated sequentially.

Verilog is a primitive language. The author has identified some of its disadvantages:

1) It is not uncommon for beginner Verilog programmers to experience error arising from a mixture of `=` and `<=`.
2) Verilog's type system is primitive. The only types are `wire` and `reg` along with their dimensions, and `reg` does not always imply a physical register.

Research has been done to tackle these issues through functional HDLs. Functional HDLs have a long history. Starting from Lava, there have been Kansas Lava, BlueSpec, Clash and Chisel, to name a few. The aim of functional HDL is High Level Synthesis (HLS), where hardware can be described at a high level of abstraction that could be even algorithmic. These languages often possess some facilities of functional programming that enables HLS – strong type system, polymorphism, higher-order functions.

However, the most important problem is 1), which requires new or modified abstraction for synchronous logic. Different functional HDLs achieve this to different degrees. Lava (both original and Kasas) syntax (Bjesse, Classen, Sheeran, & Singh, 1998) is far from intuitive because its model for hardware is a function between two streams. This is more geared towards simulation than ease of programming. Chisel (Bachrach, et al., 2012) and BlueSpec (Bluespec, Inc, 2010) structure sequential circuits around concurrent, conditional assignment operators. This is more intuitive and closer to Verilog, but both of these languages have evolved to be object-oriented and thus deviate from the scope of this project. There is another proposal to build hardware out of Cartesian Closed Categories (Elliott, 2017) which is compiled from lambda calculus written in Haskell. Even though this sounds very interesting, it is not clear from the presentation how synchronous logic can be modelled and expressed.

Clash is a functional HDL based on Haskell. It has the most natural model for synchronous logic as the figure in 1.4.1 is directly expressed as a Haskell transfer function of type:

```
f:: state -> input -> (state, output)
```

This essentially expresses synchronous hardware as a function from old state to new. It has a few advantages: 1) it is easy to write, 2) it separates concurrent update from sequential assignments using types, 3) it preserves functional nature of the language without becoming object-oriented. Therefore, Clash is the language of study. Arrow has been considered to better support hardware description (Gerards, 2011), but it does not take off eventually because the existing abstraction is already good. The case is elaborated in chapter 3.

Chapter 2 ARROW IN GAME

Functional Reactive Programming (FRP) models the game domain as a collection of interacting components maintaining and updating individual, internal state at continuous time steps, while taking in user inputs and reacting to discrete events. This model could be used to simulate a running game. FRP enables, in pure functional languages, a host of applications that fit the model of interacting, time-varying signals. Some examples are elevator control (Thompson, 2000), robotics (Paul Hudak, 2002) and games (Antony Courtney, 2003).

Arrow gives FRP a clean interface, as well as a subtle performance enhancement. The idea of using Arrow for game construction is not new. This chapter describes the author's implementation of a simple tic-tac-toe game using Arrow and explains its workings. The performance enhancement is elucidated subsequently.

2.1 Building a Game with Arrow

Similar to traditional games, the overall architecture of games in Arrow revolves around a game loop. In each iteration, three things are done: 1) capture user input 2) update game state 3) render. There are normally several interacting components in a game, each modelled by a state-carrying Signal Function.

2.1.1 Signal Function

A signal function accepts an input signal and outputs a signal, while mimicking the update of internal state in each iteration. It has the following type signature. Noticeably, at each iteration, it accepts input a, and outputs b together with a replacement of itself, that is, the "updated" version of itself for the next iteration.

$$newtype\ SF\ a\ b = \{unSF :: a \to (SF\ a\ b, b)\}$$

What about the supply of input, or the game loop as a whole? We will rely on a driver function, `runSF`. This function handles control jobs in the following sequence:

1) capture user input,

2) run input through signal function `sf`, and capture its output,

3) call low-level library functions `output` for rendering,

4) feed updated signal function `sf'` for the next iteration. If the current output asserts `quitF`, then exit without entering the next iteration.

Its full definition along with type signature appears below.

```
runSF :: IO a     -- input channel
```

```
  -> (b -> IO ()) -- output channel, should be sdl renderer
  -> SF a b       -- signal function
  -> (b -> Bool)  -- to query the current result whether game has ended
  -> IO ()
runSF input output sf quitF =
  do
     val <- input
     let (sf', res) = unSF sf val
     output res
     unless (quitF res) (runSF input output sf' quitF)
```

To facilitate the very essential state update, we use `accum`, shorthand for accumulate. This is the most important primitive signal function in the whole collection. The parameters `acc` and `acc’` keep track of states from one iteration to the next.

```
accum :: acc -> (a -> acc -> (acc, b)) -> SF a b
accum acc f = SF $ \inp ->
  let (acc',out) = f inp acc
  in (accum acc' f, out)
```

2.1.2 Game Architecture

The SF type fits into the Arrow interface as such:

```
instance Arrow SF where
    arr f = SF $ \a -> (arr f, f a)
    first (SF f) = SF $ \(b,d) ->
            let (cir, c) = f b
            in  (first cir, (c,d))
```

Each arrow represents a component in a game. The logic part of the game has the following components:

```
makeMove :: SF (Int, Int, Player, Bool) (Board, Maybe (Int,Int))
changePlayer :: SF (Maybe (Int, Int)) Player
delayedEcho :: a -> SF a a
checkStatus :: SF Board Bool
giveMessage :: SF (Board, Player, Bool) [String]
```

`makeMove`, `changePlayer` and `delayedEcho` are `accum` based state-carrying components, and the other two do not have state. Arrow operations as introduced in section 1.2 wire user inputs from an aggregate data type to different components. The complexity of such wiring increases substantially as the structure of components gets complicated. To solve this problem, Patterson's arrow syntax (Paterson, 2001) is employed. It provides intuitive syntactic sugar to the wirings, and is automatically translated to operations in section 1.2. For a full description of the workings of this syntax, section 3 of Hughes' tutorial paper (Hughes, Programming with arrows, 2004) provides a good reference.

Let us use `makeMove` to illustrate the workings of Signal Function. First, it is a signal function that takes in `(Int, Int, Player, Bool)`. The first two integers are coordinates of a tic-tac-toe piece, Player is an indicator of which player is moving, and Bool is a signal coming from the game status checker `checkStatus`, indicating if the game has ended – if true, the move will not be made.

As mentioned, `makeMove` is a state-carrying signal function. Thus, it is implemented with `accum`. The state is the tic-tac-toe board, modelled as a 2-D list and updated every iteration. Below is the top-level definition of `makeMove`, where `updateBoard` updates the state according to user input and exit conditions:

```
emptyBoard = [[]]
makeMove = accum emptyBoard updateBoard
      where updateBoard (x,y,player,exit) board = …
```

This function is then combined, using the arrow syntax, with others to give the major game logic:

```
updateBoardAndPlayer :: SF (Int, Int) GameState
updateBoardAndPlayer = proc (x, y) -> do
  rec result <- makeMove -< (x, y, player, exitGame)
      hasEnded <- checkStatus -< fst result
      exitGame <- delayedEcho False -< hasEnded
      player <- changePlayer -< snd result
      msg    <- giveMessage -< (fst result, player, hasEnded)
  returnA -< Game (fst result) player exitGame msgw
```

This main logic is composed with functions that capture and wire keyboard inputs to form the top-level game Arrow.

```
game :: SF (Maybe KeyInput) (GameState, Focus, Bool)
game = (updateFocus >>> first (stepDown initialState
updateBoardAndPlayer))
       &&& arr (\x -> x == Just Quit)
       >>> unwrap
```

```
        where unwrap = arr (\((a,b),c) -> (a,b,c))
```

This game arrow is then invoked by `runSF` to give the game loop.

```
gameLoop :: Renderer -> IO ()
gameLoop r = do
  runSF processKeyboard (G.renderGame r font) game testQuit
  where testQuit (_,_,quit) = quit
```

`gameLoop` is implemented in `Main.hs`. The main game logic is in `Logic.hs`. The Arrow implementation is in `Lib.hs`. These three files are provided in Appendix A.

2.2 Implementation issue I: A time and space leak

Upon contemplation, it seems that Arrow is redundant. The Signal Function type, as realized in FRP, is all that is required to build a simple game. The combinators can be customized and there is no need to resort to Arrow for support. This is not true because Arrow does not only provide a standardised interface. Although not directly encountered in the author's game implementation, a generic class of time-and-space leak is solved by the Arrow `loop` combinator. The first FRP library that uses Arrow is Yampa. However, researchers who use Yampa in their applications seem to blissfully accept the aforementioned performance benefit without worrying about why it is.

The paper which originally describes the leak is by Liu (Liu & Hudak, 2007), but the explanation provided in the paper is rather brief and abstract. This section seeks to elucidate the problem further with mathematical formulation and concrete profiling.

2.2.1 A Problematic Signal

FRP did not originally use Signal Function as first-order construct. Instead, it used first-order Signals, which is actually a more intuitive definition. A Signal is defined as a function from time to value, as follows:

$$newtype\ Signal\ a = Time \rightarrow a$$

This would have been adequate in describing the continuous nature of games, where objects change state according to time. However, the above mathematical definition has to be approximated by discrete time steps in some way, because that is how computers work. In the definition that follows, a signal consists of the current value and its continuation function:

```
type Time = Double
Signal a = S (a, Time -> Signal a)
```

Here is a way to unfold a signal through “time”. The lengths of time steps are supplied as a list in the second argument:

```
runC (S (a,f)) (dt:dts) = a : runC (f dt) dts
runC (S (a,f)) [] = []
```

The problem occurs when we try to define recursive signals on top of an already recursive signal type. It is not uncommon for game’s physics engine to do integration on signals, such as the following:

```
integral init (S p)
= S (init, \dt -> integral (init + (fst p)*dt) (snd p dt))
```

It is important to notice that the integrated signal is delayed from the original by one time step because of arbitrary constant appearing in `init`. Let us integrate a constant signal initialized to 10.

```
const1 = S (1, \dt -> const1)
dts = repeat 1

*Main> take 10 (runC (integral 10 const1) dts)
[10,11,12,13,14,15,16,17,18,19]
*Main> take 10 (runC const1 dts)
[1,1,1,1,1,1,1,1,1,1]
```

It is normally expected that the input signal, const1, are synchronous with the integrated signal, but in this implementation the integrated signal is late by one time step. At each time step, the integral function takes the current input and compounds it only at the next step. This is a very property that leads to the creation of the exponential signal, where we integrate a signal by itself:

```
e = integral 1 e
```

At first glance, there is an error because `e` is referenced before it is defined. However, because the integrated signal is late by one time step and is initialised to 1, we are able to take the input at the current time step, which is itself, compound it for the next step, and continue with this procedure.

```
*Main> take 10 (runC e dts)
[1,2,4,8,16,32,64,128,256,512]
```

However, computing recursive signals such as `e` has a time and space leak. Intuitively, `e` should take O(n) time and constant space to execute, because to compute the nth value, we repeat the compounding values about n times. There is only one unit of memory to store simultaneously. This could be mathematically inferenced as follows:

$e = integral\ 1\ e$

$let\ e = S_0S_1S_2 \dots, where\ S_i\ is\ the\ value\ of\ e\ at\ time\ i.$

$let\ dt_0dt_1dt_2 \dots denote\ the\ lengths\ of\ time\ steps.$

$Then\ S_0 = 1\ by\ the\ definition\ of\ integral.$

$To\ calculate\ S_1, we\ compound\ the\ input\ signal\ at\ the\ current\ step\ to\ the\ result,$

$which\ is\ S_1 = S_0 + S_0dt_0$

$At\ step\ i, we\ compute\ S_{i+1}, for\ which\ we\ compound\ the\ current\ input\ to\ the\ current\ result.$

$We\ get\ S_{i+1} = S_i + S_idt_i\ .$

It is easy to see that operationally we only need the current result for the next result. We can also go by the definition of integration, which is the naïve way, as in (3):

$let\ e = S_0S_1S_2 \dots, where\ S_i\ is\ the\ value\ of\ e\ at\ time\ i.$

$By\ the\ mathematical\ definition\ of\ integration,$

$$S_i = \int_0^i S\,dt$$

$$\approx \begin{cases} C & when\ i = 0 \qquad (1) \\ C + \sum_{k=0}^{i-1} S_kdt_k & otherwise. \qquad (2) \end{cases}$$

$From\ (2),$

$$S_{i+1} = C + \sum_{k=0}^{i} S_kdt_k \qquad (3)$$

$$= C + \sum_{k=0}^{i-1} S_kdt_k + S_idt_i$$

$$= \left(C + \sum_{k=0}^{i-1} S_kdt_k \right) + S_idt_i$$

$$= S_i + S_idt_i \qquad (4)$$

We soon realise that (3) reduces to (4), which means our old intuition about e is correct. Also, (3) is way less efficient. It takes O(n^2) time to reach from the first to the nth value because every ith step requires recalculation from the bottom. It takes O(n) space because for the nth value, it has to create space for all of the previous values in order to sum them up.

While the mathematical derivations are intuitive and helpful in explaining the issue, empirical evidence is sought to confirm the above reasoning. Time and space profiling are done using the naïve recursive signal implementation, with input sizes ranging from 1000 to 5000 at an interval of 1000. The result of time profiling confirms that execution time is indeed quadratic, as tabulated below:

N	Time (s)	Time, normalized
1000	0.04	1
2000	0.15	3.75
3000	0.38	9.5
4000	0.67	16.75
5000	1.10	27.5

Space profiling has also been done using input size of 2000. This size is arbitrary and does not matter much because all that needs to be shown is space building up linearly as program runs. The result confirms O(n) space usage:

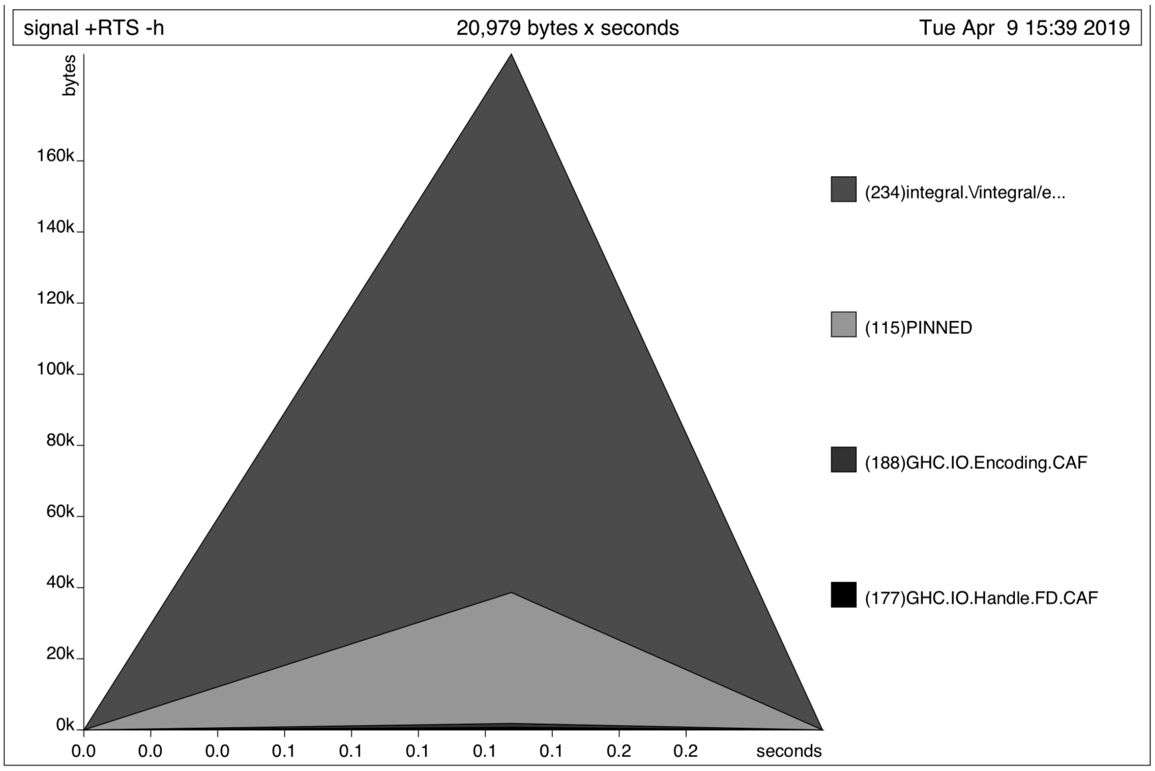


Fig 2.2.2 space profile of naïve signal implementation

Next, an a-priori lexical unfolding of the execution of `integral` is carried out to explain that it indeed implements equation (3) instead of (4). Throughout the demonstration, pointers to function arguments are shared across the function call according to standard call-by-need. Introduction to a pointer is denoted by superscript such as ^{1}, and pointer reference is denoted by square brackets [1]. Also, as arguments are evaluated, the content of pointer may change. This will be reflected in the accompanying memory map.

```
--runC will 1)put the first component in the output list, 2)strip out the
second component of signal, at each iteration
  and apply dt to it
dts = repeat 1
$^1$e = integral 1 e
--unfold e assuming dt_i = 1 for all i
= $^1$(1, \dt->integral (1+(fst [1])*dt)) (snd [1] dt)

--strip out second component and apply dt (@)
integral (1+fst [1]*1) $^2$(snd [1] 1)
--evaluate (1+fst[1]*1).
      [1] = (1,…)
      (1+fst[1]*1) = 2
we get (2,\dt->integral (2+fst [2]*dt) (snd [2] dt)) (@@)

--strip out second component and apply dt
integral (2+fst [2]*1) (snd [2] 1)
--evaluate (2+fst [2]*1).
      [2] = (snd [1] 1)
          = integral $^3$(1+(fst [1])*1) $^4$(snd [1] 1)
          = ([3], \dt-> integral ([3]+fst [4]*dt) (snd [4] dt))
      (2+fst [2]*1)
          = (2+[3]*1)
          = (2+ ( (1+fst[1])*1 ) * 1) –take note of this value here
```

Memory:
[1] (1, \dt->…)
[2] (snd [1] 1)
[3] (1+(fst [1]) * 1)
[4]

When evaluating `(2+fst [2]*1)` as the third value of the sequence, we take the first of [2]. This is same as taking first of (second of [1] applied to 1), which is repeated work because the exact same thing has been done at (@). The culprit is that second of [1] is a lambda abstraction, so we have no choice but to copy and re-evaluate. Imagine second of [1] being a first-order construct, then (@) would have rewritten it with (@@), which could be shared when evaluating first of [2].

Consequently, the computation of every next value will have to traverse the entire sequence and eventually refer back to the initial value at `[1]`, which means storage for all previous values cannot be freed, resulting in build-up of space linear to the amount of values as calculation proceeds. For one particular iteration, all previous values need to be calculated again, which results in quadratic time.

The way out of this entanglement is to go back to the original intention of equation (4) in the mathematical definition of integral. In each iteration, we need a way to feed the output value back in order to create the next value. For example, in calculating `fst [2]` at (@@), note that `[2]` is the continuation of `[1]`, which is just `[2]`.

2.2.2 Arrow to the Rescue

The loop construct in section 1.2.3 is perfect for solving the problem outlined above. It is able to implement equation (4) of section 2.2.2 straightforwardly, bypassing the need of tinkering on the original Signal type.

The implementation which Liu provides appears as follows (slightly modified). The general idea is to create a Signal Function that accepts part of its output as input, as in 1.2.3:

```
instance Arrow SF where
  arr f = SF (\x -> (f x, \dt -> arr f))
  first (SF f) = SF (\(x, z) -> let (y, f') = f x
                                in ((y, z), first . f'))
instance Cat.Category SF where
  id = SF (\x -> (x, const Cat.id))
  (.) = dot
    where
      SF g `dot` SF f = SF (\x -> let (y, f') = f x
                                      (z, g') = g y
                                  in (z, \dt -> g' dt `dot` f' dt))
instance ArrowLoop SF where
  loop (SF f) = SF (\x -> let ((y, z), f') = f (x, z)
                          in (y, \dt -> loop (f' dt)))
integralSF:: Double -> SF Double Double
integralSF i = SF (\x -> (i, \dt -> integralSF (i + x * dt)))
```

```
runSF:: SF Double Double -> [Double] -> [Double] -> [Double]
runSF (SF f) (dt: dts) (x:xs) = let (v, f') = f x
                                 in v : runSF (f' dt) dts xs
runSF _ [] _ = []
runSF _ _ [] = []
```

Noticeably, `loop` constructs a feedback loop from output z back to input. This enables the current signal value to be used directly for computing the next value and uses only a constant structure throughout. Therefore, time complexity should be O(n) and space complexity should be O(1) a-priori.

Liu also mentions there is another way to visualize the signal type as lists:

```
Signal a = S ([DTime] -> [a])
integralS i (S f) = S (\dts -> scanl (+) i (zipWith (*) dts (f dts)))

scanl f acc [] = []
scanl f acc (x:xs) = acc : scanl f (f acc x) xs
```

O(n^2) time and O(n) space also applies here, but the corresponding Arrow implementation is not presented in the paper. We provide our complementary implementation in Appendix B.

Our hypothesis is that both Liu's implementation and ours (as in Appendix B) run in linear time and constant space. Liu's paper does not present any evaluation result to confirm the hypothesis. The Glasgow Haskell Compiler (GHC) provides a suite of standard time and space profiling tools. We implement the solutions in Haskell and use GHC for time and space analyses.

Before the actual experiment, some test runs are performed on our implementation using some arbitrary input size, and the results look rather strange:

N	Time (s)	Time, normalized
1000	0.009	1
2000	0.007	0.8
3000	0.011	1.2
4000	0.013	1.4
5000	0.013	1.4
10000	0.019	2.1
100000	0.17	18.9

The time complexity appears constant over 1000-5000 and explodes at 100000, although it is still within linear range. It is suspected that overhead is predominant for small input sizes, so there is little variation as input increases by one thousand. Therefore, input size has to be reasonably large, and in subsequent experiments are set to start from 10000. Also, there is sizeable variation in time among several runs of the same program, supposedly due to runtime environment variation. This deviation has to be averaged out. Lastly, GHC profiler records time more accurately in CPU ticks rather than seconds, so time will be in ticks subsequently. Each tick is 1000us.

Both Liu's and our implementations are run on the Haskell Platform with both time and space profiling. Inputs range from 10000 to 60000, with step of 10000. For each input size, the code is run 3 times and the average result is taken as the time usage. Lastly, time is normalized with respect to time taken for input size of 10000.

The assumption is that time taken for the signal to unfold is the predominant time consumer of the programs. This is fairly realistic because the tests contain only the unfolding algorithm.

The results of time profiling appear below. Complexity appears linear for both implementations.

Author's implementation

N	time1/tick	tiem2/tick	time3/tick	time_avg/tick	time_normal
10000	29	17	23	23.00	1.00
20000	39	41	49	43.00	1.87
30000	46	55	58	53.00	2.30
40000	73	79	68	73.33	3.19
50000	75	86	81	80.67	3.51
60000	95	105	112	104.00	4.52

Liu's implementation

N	time1/tick	tiem2/tick	time3/tick	time_avg/tick	time_normal
10000	14	19	21	18.00	1.00
20000	35	38	37	36.67	2.04
30000	59	49	65	57.67	3.20
40000	79	87	79	81.67	4.54
50000	98	93	95	95.33	5.30
60000	114	96	104	104.67	5.81

As for space profiling, we have run both implementations on input size of 60000. The results show that both implementations is indeed O(1) in space. In the screenshots that follow, we can see space usage being constant throughout the program runs.

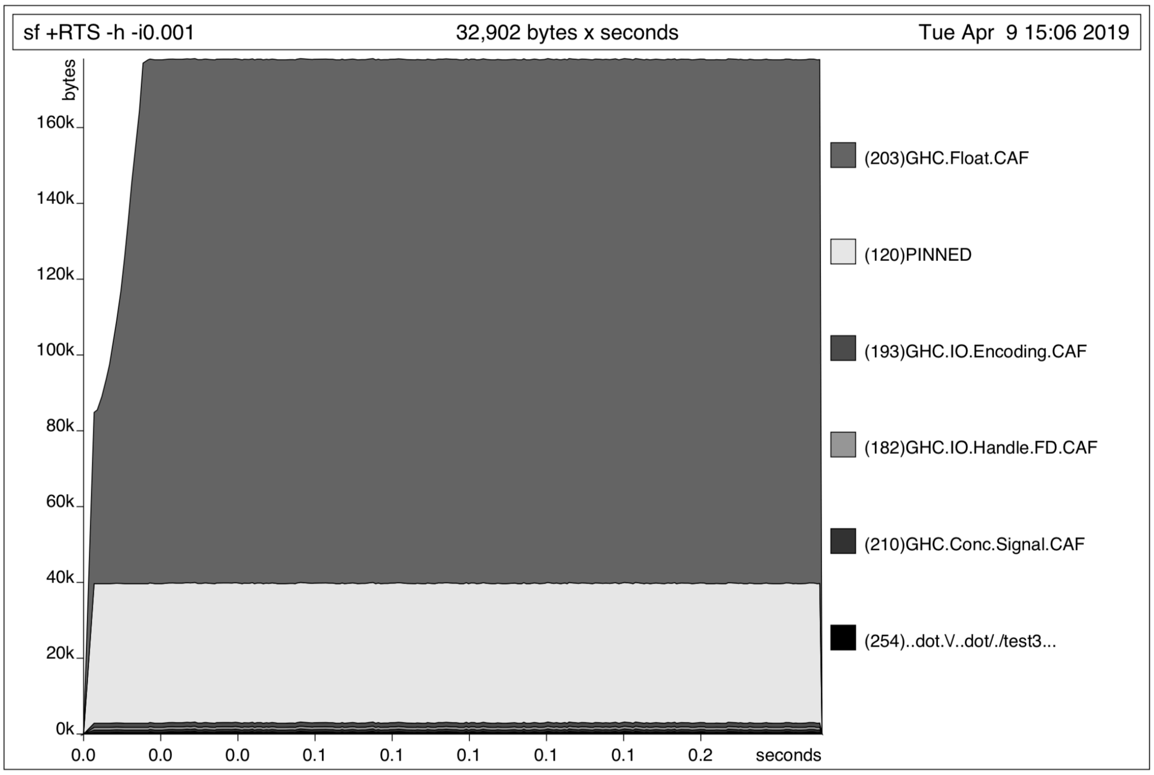


Fig 2.2.3a space profile of Liu's implementation

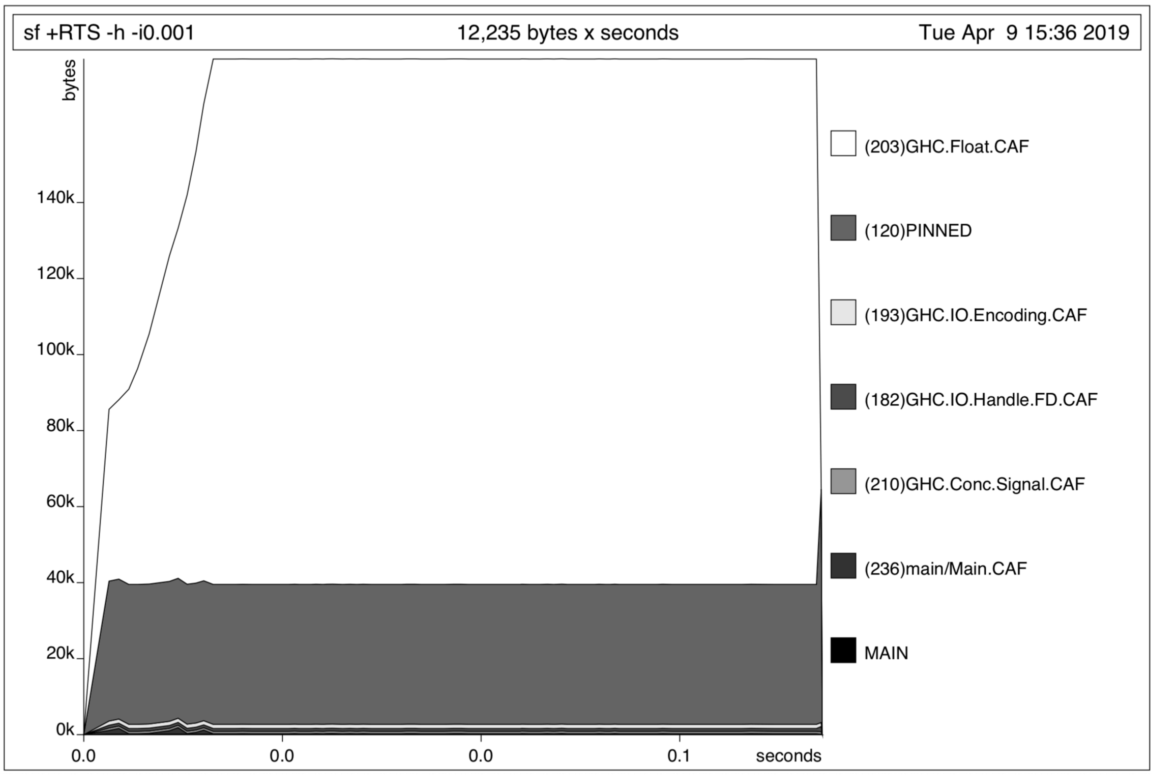


Fig 2.2.3b space profile of author's implementation

We have thus demonstrated, with empirical evidence, that Arrow has immense performance benefits for evaluating recursive signals.

2.3 Implementation issue II: Lazier functional programming

During implementation of both the naïve Signal type and the Arrow instance in Appendix B, problems arise with regards to infinite loop. `zipLazy` is used instead of the normal `zip`. When the normal zip is used to create the input stream of tuples to run through argument Signal Function `f`, the feedback term z during the first iteration is undefined. This is because the initial signal value is arbitrary (refer to equation (1) of 2.2.2) and does not depend on any previous values. That is why we should make an exception about reading in `zs` at iteration 0. After the initial iteration, `zs` will be defined and further reading of `zs` will return the current signal value.

To achieve this effect, we need to "not evaluate" `zs` at first. This is not possible with `zip` because the zip is head-strict on both of its arguments. A `~` sign changes this, making the pattern match irrefutable by delaying evaluation until when the argument is really needed. Overall, the Arrow runs in the following steps:

1) At iteration 0, inputs and `zs` is zipped up and passed to `test2`. Because of irrefutable pattern matching, `zs` is not evaluated and `test2`, which is essentially `scanl` with an initial value, outputs the initial value without querying `zs`.
2) The above output is fed back by loop to define the first value of `zs`.
3) At iteration 1, the list produced by zipping `zs` and inputs in step 1) is again passed to `test2`. `Scanl`, having produced the first output, will now evaluate its input list and take the first value to output the next value. However, the input list is exactly `zs`, and its first value is defined in 1). Thus, the evaluation proceeds successfully, with `scanl` producing and using the latest value in `zs` every iteration.

The technique of irrefutable pattern matching could be made more general. The above is just one very special use case. In general, there are a few cases where irrefutable pattern makes sense:

1) All parameters to the function have types with single data constructor. In other words, no parameter has sum type.
2) When one of the parameters has sum type, there are preconditions on the actual arguments.

The first of the above is easier to visualize. When no parameter has sum type, the pattern matching will only have one branch. Adding ~ has no effect other than delaying evaluation and, intuitively, any failure in pattern matching will be caught later when the argument is needed, or never if the function does not need the actual argument.

The second case is complicated because irrefutable pattern will alter program behavior in unexpected ways. Let us consider the old `zip` function.

```
zipNormal (x:xs) (y:ys) = (x,y) : zipNormal xs ys
zipNormal _ _ = []
```

If we want to delay evaluating the second argument, we put a ~ in front like so:

```
zipLazySnd (x:xs) ~(y:ys) = (x,y) : zipNormal xs ys
zipLazySnd _ _ = []
```

The behavior of zipNormal is that if one list is longer than the other, the result length follows the shorter list. This is operationally realized in the second branch. If the first branch fails, it must mean one of the lists is empty, and the second branch will succeed anyway and return [].

zipLazySnd, on the other hand, requires the second argument to be longer than the first. Consider the opposite case where the second list runs out before the first. We have a non-empty first argument and an empty second argument. The first branch of pattern match still succeeds because the second argument is irrefutably matched. However, error will appear as evaluation proceeds to zip the actual arguments. In the Arrow case, this presents no problem because both arguments are infinite, but this is not generalizable.

It seems that irrefutable pattern can only be used sparingly with very specific purpose in mind. In cases where it does gets used, however, we enjoy extended program semantics. Consider an example where a function returns a constant regardless of its argument:

```
f ~(x:xs) = 3
```

Had the ~ not been there, an argument that is $\perp$ will result in divergence. With ~, the function becomes non-strict and always terminates.

In this project, the technique of automatically making patterns lazy has been considered to reap the semantic benefits. However, it did not take off because of lack of motivation. First, use cases are few; second, real-world programmers seem cautious about this kind of change. Appendix C is an excerpt from Stackoverflow, where the author asks about automatically making pattern matching lazier to expand the program's semantics. The response is rather negative. The author thinks it is reasonable this way because irrefutable pattern walks away from the spirit of pattern matching in the first place, which is to put up a fence that bars data that does not fit the pattern. In case where it should fail, it is better to fail early.

2.4 Discussion

In this project we have shown that Arrow could be used as an abstraction for games. What we have not shown is how good an abstraction it is. Mainstream games have a similar game loop, but entities and interactions are built with C++ and are object-oriented. This means the question somehow defaults to a comparison between functional programming (specifically FRP) and OO, and that has too broad a scope to be addressed here. From the author's experience, however, FRP is way less error-prone than imperative paradigms. Once the code passes static type-checking, the game just works. One downside of Arrow may be that performance-critical components are not easily built. That is because Arrow sits at a higher level of abstraction and is mostly about modelling behaviours rather than performance fine-tuning. The author admits being not well-versed in traditional game programming and asks for a more in-depth comparison between the two paradigms.

One comment about time and space leaks. Although Arrow could solve the specific problem outlined in section 2.2 with good effect, it is only possible because each signal value depends solely on the previous one. In general, much storage would still have to be allocated if a signal inherently depends on everything since the beginning. Thus, it is questionable how the Arrow technique is applicable to wider classes of space leaks.

One demystifying (or rather advertising) statement is that functional programming is NOT difficult. Admittedly, using FRP in building games happens mostly in research and personal projects. It is immature and does not have the kind of tooling and environment support that C++ has. However, we have a lot to lose if we are to dismiss FRP as too hard for common programmers. From the author's point of view, Arrow and OO are not that different after all. We are perfectly free to visualize each object in OO as a component Arrow in FRP, and that is actually how some more complicated games are built with Yampa the Arrow library [the Yampa arcade].

To conclude, FRP certainly could be used in building games. It is a functional modelling of the traditionally OO game domain and has all the benefits functional programming has to offer. It has good potential and should be tried out more at scale.

Chapter 3 ARROW IN HARDWARE

It is mentioned in the introduction that Arrow did not eventually take off. In fact, Clash has had Arrow before, but it was removed in recent years. As for why it is gone, here is Clash's maker's reply to the author (Baaij, 2019):

So as to why we/I stopped using arrows:

1. I prefer the Applicative interface over the Arrow interface

2. Desu[ga]ring of Arrow syntax is very sequential (i.e. parallel compositions are decomposed to `first f . second g` as opposed to `f *** g`)

3. Arrow syntax would be another syntax on top of Haskell that non-Haskellers would have to learn when using Clash

Personal preference aside, it seems Arrow does not add much benefit to Clash's abstraction as point 3 suggests. We do not re-implement Arrow but seek to discuss briefly why Arrow is limited through a small example in Clash.

3.1 Clash Example

Our example is a mini encoding scheme using a lookup table (LUT). The idea is extracted from the author's current Verilog project. The encoding takes the extremely simple steps as follows:

```
$given\ input\ x$
$repeat\ 2\ times$:
  $x \coloneqq lut(x)$
$output\ x$
```

For the purpose of illustration, we implement this algorithm as synchronous logic. State diagram and functional block diagrams appear below:

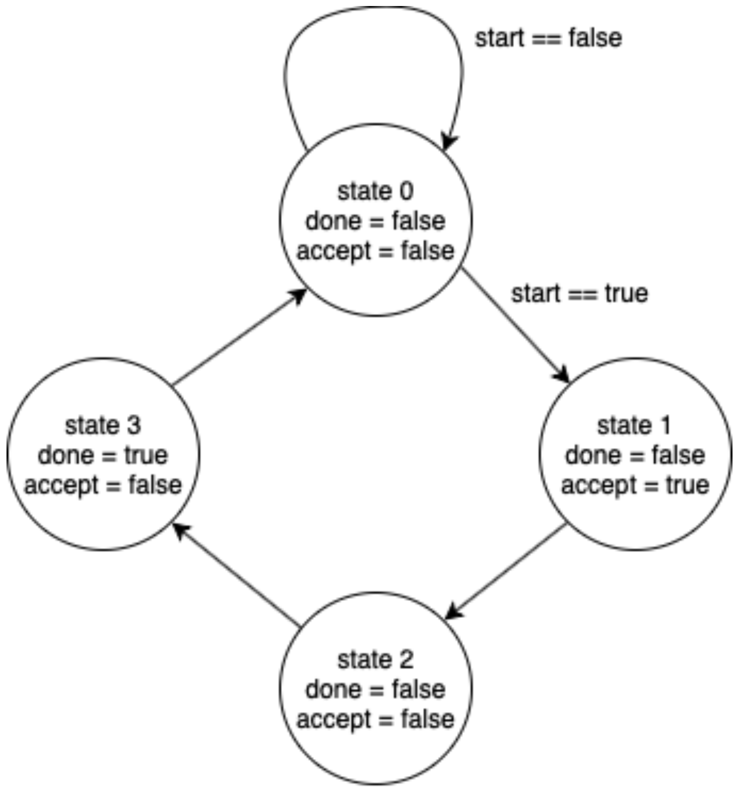


Fig 3.1a finite state machine diagram

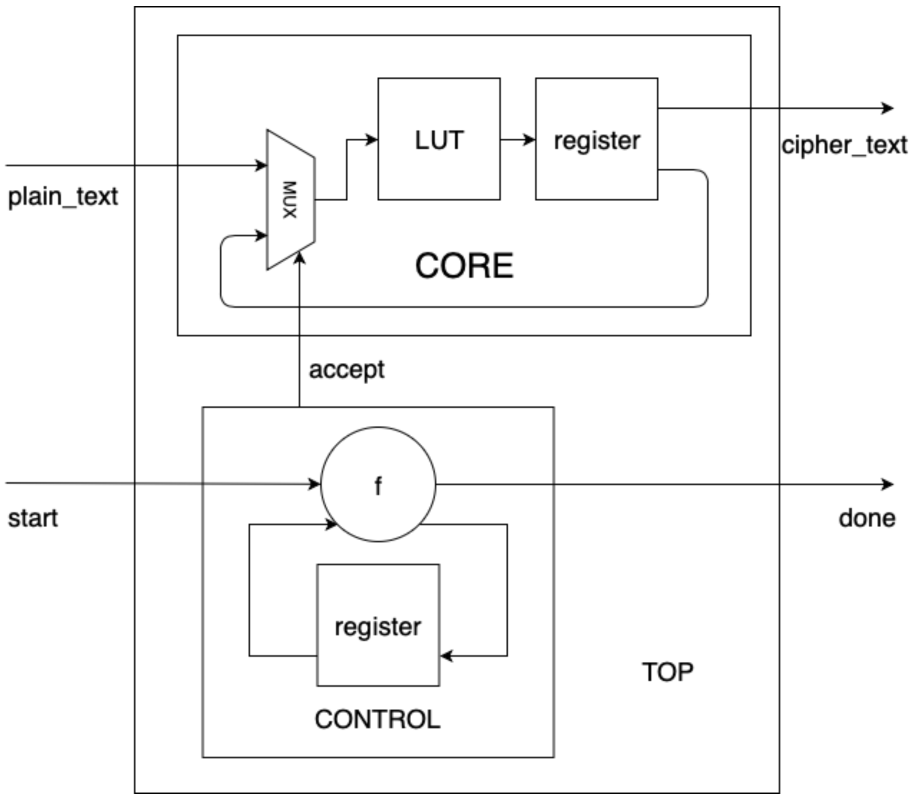


Fig 3.1b functional block diagram

The full Clash code is in Appendix D. We pay particular attention to the top-level function, aes_top, as well as the type signatures of other transfer functions:

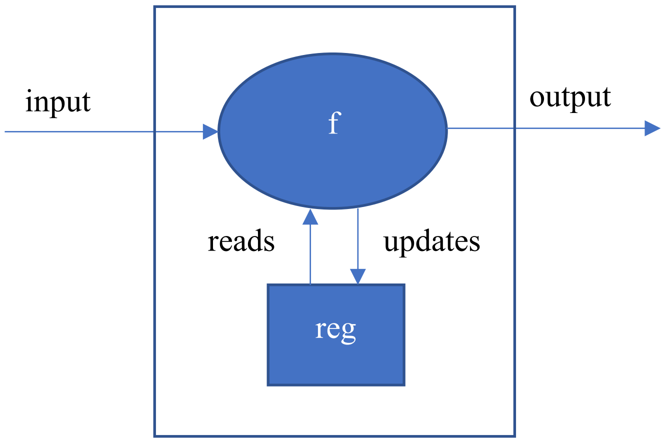
```
aes_cntx:: (Signed 32) -> Bool -> (Signed 32, (Bool,Bool))
aes_core :: (Signed 32) -> (Bool, Signed 32) -> (Signed 32, Signed 32)
aes_top :: Signal (Bool, Signed 32) -> Signal (Signed 32, Bool)
aes_top input =
     let (start, plain_text) = unbundle input
         (accept, done) = unbundle (mealy aes_cntx 0 start)
         cipher_text = mealy aes_core 0 (bundle (accept, plain_text))
     in
         bundle (cipher_text, done)
```

This function implements top-level logic TOP of Fig 3.1b.

Notice the use of `mealy`. It is a critical function that converts a transfer function to function between `Signal`s. It has the following signature:

```
      mealy:: (s -> i -> (s, o)) -> s -> Signal i -> Signal o
```

Within aes_top, the control and core units written as transfer functions cannot be directly applied without `mealy`. This is because of a type-signature mismatch as illustrated below:



When we write transfer functions, we take the perspective of f inside the box. Thus, we have a function type from both state AND input to (state, output). However, when we instantiate a piece of synchronous logic, we are taking the perspective outside the box. We do not have the arguments we had for transfer functions, only input and output. This is why even though transfer functions can fully describe synchronous logic, it cannot be directly applied for instantiation. Instead, some mechanism is needed to wrap them in a “box”, supply initial register values and make them available for use. This is exactly what `mealy` does. From an outside view, states in transfer functions are “hidden”.

3.1 Arrow’s Downfall

Arrow gives another method to “hide” state and wrap a transfer function. The Arrow instance used here is exactly the same as that in chapter 2, that is, a simplified version of Liu’s implementation. We reproduce the important parts below. The full definition is in Appendix D.

In `liftF`, we create a state-carrying Arrow out of a transfer function.

```
liftF :: (s -> i -> (s,o)) -> s -> SF i o
liftF f acc = SF $ \inp ->
     let (acc', out) = f acc inp
     in (liftF f acc', out)

aes_core_sf = liftF aes_core 0
aes_cntx_sf = liftF aes_cntx 0
```

then, we could compose Arrows using the usual combinators introduced in chapter 1. This is easy as states are kept within individual Arrows and only input/output types are exposed. We just need to take care of wiring the signals correctly.

```
aes_top_sf :: SF (Bool, Signed 32) (Signed 32, Bool)
aes_top_sf = first aes_cntx_sf
    >>> arr (\((accept,done), plain) -> (done, (accept,plain)))
    >>> second aes_core_sf
>>> arr (\(a,b) -> (b,a))
```

Subsequently, we can convert the SF into a function between Signals. Signal has the signature `Signal a = a :- Signal a`, which is a simplification of the `Signal` definition of 2.2.1. A signal comprises its current value and a continuation. Instead of using tuple, an analogous custom notation `:-` is used by Clash, so we follow as needed.

```
liftSF:: SF a b -> ((Signal a) -> (Signal b))
liftSF (SF f) = \(x :- xs) ->
    let (sf', out) = f x
in (out :- (liftSF sf' xs))
aes_top2 = liftSF aes_top_sf
```

These are all the definitions we need to utilize Arrow for hardware simulation. Both the native `mealy` implementation (aes_top) and the new Arrow implementation (aes_top2) are tested with a stream of data, and they give the same encoding for the input numbers. Behaviourally, the two implementations are equivalent. The console output is presented below.

```
_take :: Int -> [a] -> [a]
x = (True,2) : _repeat (False,2)
*AES> _take 10 (simulate aes_top x)
[(0,False),(0,False),(4,False),(9,True),(0,False),(0,False),(0,False),(0,F
alse),(0,False),(0,False)]
*AES> _take 10 (simulate aes_top2 x)
[(0,False),(0,False),(4,False),(9,True),(0,False),(0,False),(0,False),(0,F
alse),(0,False),(0,False)]
```
From this result, we could see the viability of Arrow as an abstraction for circuit simulation. However, how much value does it add to the original implementation? There is no real benefit considering the following:

1) There is no performance boost here because we are not dealing with recursive signals containing second-order values as in section 2.2.
2) It does not make programming hardware more intuitive. Moreover, there is additional structural overhead because input and output signals need to be wired correctly in Arrow notation. Of course, Arrow Syntax is there to help; but that is still not as convenient as plain `mealy`.
3) It has limited use. In hardware design, the only abstraction where we need to tie together many pieces of synchronous logic is probably the top level as in our example. No other places will need Arrow, and although Arrow could theoretically be nested, it does not really make sense for registers and would only complicate the picture and chances of error.
4) It has extra cost of de-sugaring and synthesis. This is partly point 2 as Clash's maker mentioned in the beginning of the chapter, but extra compiler support is needed to compile Arrow to Verilog. Our implementation in Appendix D is a simple one and is only good for simulating the external behaviours of circuits. Hardware synthesis, on the other hand, needs to face the complicated Arrow type signature head-on. The anticipated work is not trivial.
5) It is more complicated for programmers to learn.

Considering all of the above, Arrow in Clash is by and large a lost cause. Of course, there may be places other than front-end, or languages other than Clash where Arrow could help, but that is left for future investigations.

Chapter 4 CONCLUSION

We have investigated the application of Arrow in domains of game programming and hardware description. The discovery is that while Arrow successfully provides abstraction for both domains, it has better utility in games because of the unifying interface and the performance benefits.

In hardware description, however, Arrow does not really value-add to the current, already intuitive abstraction. Moreover, it incurs extra learning curve and requires extra compiler support. These problems are largely absent in games, because 1) the learning curve for Arrow is the learning curve for any kind of functional game programming, so having Arrow is not "extra" and 2) extra compiler support is not needed in game because game is essentially a simulation and does not require separate compilation.

The answer to the question of whether Arrow provides an effective abstraction for application domains, therefore, is a qualified yes. We have seen libraries that have really benefitted from Arrow, and ones that consign it to oblivion. We hope that Arrow can find its way to more application domains in the future.

Chapter 5 Bibliography

Wadler, P. (1995). Monads for Functional Programming. *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text* (pp. 24-52). Berlin: Springer-Verlag Berlin, Heidelberg.

Hughes, J. (2000). Generalising monads to arrows. *Science of Computer Programming - Special issue on mathematics of program construction*, 67-111.

Bjesse, P., Classen, K., Sheeran, M., & Singh, S. (1998). Lava: hardware design in Haskell. *ICFP '98 Proceedings of the third ACM SIGPLAN international conference on Functional programming*, (pp. 174-184).

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., . . . Asanović, K. (2012). Chisel: constructing hardware in a Scala embedded language. *DAC '12 Proceedings of the 49th Annual Design Automation Conference* (pp. 1216-1225). San Francisco: ACM.

Bluespec, Inc. (2010). *Bluespec SystemVerilog Reference Guide.* Bluespec, Inc.

Elliott, C. (2017). Compiling to categories. *Proceedings of the ACM on Programming Languages*.

Gerards, M. E. (2011). Higher-Order Abstraction in Hardware Descriptions with C?aSH. *Conference: Digital System Design (DSD).* Oulu, Finland.

Hughes, J. (2004). Programming with arrows. *AFP'04 Proceedings of the 5th international conference on Advanced Functional Programming* (pp. 73-129). Tartu, Estonia: Springer-Verlag Berlin, Heidelberg.

Paterson, R. (2001). A new notation for arrows. *ICFP '01 Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (pp. 229-240). Florence, Italy: ACM.

Liu, H., & Hudak, P. (2007). Plugging a Space Leak with an Arrow. *Electronic Notes in Theoretical Computer Science*, 29-45.

Baaij, C. (2019). Email Correspondence.

Thompson, S. (2000). A functional reactive animation of a lift using Fran. *Journal of Functional Programming*, 245-268.

Paul Hudak, A. C. (2002). Arrows, Robots, and Functional Reactive Programming. In *Lecture Notes in Computer Science* (pp. 159-187). Springer.

Antony Courtney, H. N. (2003). The Yampa arcade. *Haskell '03 Proceedings of the 2003 ACM SIGPLAN workshop on Haskell* (pp. 7-18). Uppsala, Sweden: ACM.

Appendix A Game implementation

Lib.hs

```haskell
module Lib
    ( accum, accum', accumEvent, accumulate, constant,
      embed, hold, runSF, SF, stepDown, tag,
      sfInit, runOnce,
      SFState(..)
    ) where

import           Control.Arrow
import qualified Control.Category as Cat
import           Control.Monad
import           Data.IORef
import           Data.Maybe

someFunc :: IO ()
someFunc = putStrLn "someFunc"

{-Arrow instance and primitives-}
newtype SF a b = SF {
    unSF :: a -> (SF a b, b)
}

instance Cat.Category SF where
    id = SF $ \a -> (Cat.id, a)
    (.) = dot
      where
        (SF cir2) `dot` (SF cir1) = SF $ \a ->
            let (cir1', b) = cir1 a
                (cir2', c) = cir2 b
            in  (cir2' `dot` cir1', c)

instance Arrow SF where
    arr f = SF $ \a -> (arr f, f a)
    first (SF f) = SF $ \(b,d) ->
            let (cir, c) = f b
            in  (first cir, (c,d))

instance ArrowChoice SF where
  left c@(SF f) = SF $ \input ->
    case input of
      Left l  -> let (newCir, out) = f l in (left newCir, Left out)
      Right r -> (left c, Right r)

instance ArrowLoop SF where
  loop c@(SF f) = SF $ \input ->
    let (sf', (c, d)) = f (input, d)
    in (loop sf', c)

--primitive
accum :: acc -> (a -> acc -> (acc, b)) -> SF a b
accum acc f = SF $ \inp ->
  let (acc',out) = f inp acc
  in (accum acc' f, out)

accum' :: acc -> (a -> acc -> acc) -> SF a acc
```

```haskell
accum' acc f = accum acc (\inp acc -> let out = f inp acc in (out, out))

--primitive
tag :: b -> SF (Maybe a) (Maybe b)
tag val =
  arr $ \input -> case input of
                    Nothing      -> Nothing
                    Just someVal -> Just val

--primitive
hold :: a -> SF (Maybe a) a
hold init =
  let f inp acc = case inp of
        Nothing  -> (acc, acc)
        Just out -> (out, out)
  in accum init f

constant val = SF $ \_ -> (constant val, val)

--primitive to make a SF incapable of event processing behave as if it can
stepDown :: b -> SF a b -> SF (Maybe a) b
stepDown init sf = arr (\inp -> case inp of
                                  Nothing -> Right Nothing
                                  Just v  -> Left v)
              >>> ( (sf >>> arr Just) ||| arr id )
              >>> hold init

--primitive
accumulate :: a -> SF (Maybe (a -> a)) (Maybe a)
accumulate init =
  let f fIn acc = case fIn of
        Nothing -> (acc, Nothing)
        Just g -> let out = g acc
                  in (out, Just out)
  in accum init f

--primitive
accumEvent :: a -> (a -> a -> a) -> SF (Maybe a) a
accumEvent init g =
  let f inp acc = case inp of
        Nothing -> (acc, acc)
        Just curr -> let out = g curr acc
                     in (out, out)
  in accum init f

embed :: SF a b -> [a] -> [b]
embed cir ls = case ls of
  [] -> []
  (x:xs) -> let (cir', c) = unSF cir x
            in  c: embed cir' xs

runSF :: IO a      -- input channel
  -> (b -> IO ()) -- output channel, should be sdl renderer
  -> SF a b       -- signal function
  -> (b -> Bool)
  -> IO ()
runSF input output sf quitF =
  do
     val <- input
```

```
    let (sf', res) = unSF sf val
    output res
    unless (quitF res) (runSF input output sf' quitF)
```

Logic.hs

```
--Game logic for arrowized tic tac toe
{-# LANGUAGE Arrows #-}
module Logic
  ( game, dummy,
    initialState,
    updateBoardAndPlayer,
    GameState(..), Board, Focus,
    delayedEcho,
    parseText,
    Player(..), Symbol(..)
  ) where

import           Control.Arrow
import           Data.List
import           Lib
import           SDLInput

data Symbol = X | O | N deriving (Show,Eq)

data Player = PX | PO deriving Show

notP PX = PO
notP PO = PX

getSymbol:: Player -> Symbol
getSymbol PX = X
getSymbol PO = O

type Board = [[Symbol]]
type Focus = (Int, Int)
type InputXY = (Int, Int)

data GameState = Game
  {getBoard  :: Board,
   getPlayer :: Player,
   hasWon    :: Bool,
   msg       :: [String]
  }\

alternate :: SF () Bool
alternate = accum True (\_ acc -> (not acc, acc))

delayedEcho :: a -> SF a a
delayedEcho init = accum init $ \inp acc -> (inp, acc)

updateFocus :: SF (Maybe KeyInput) (Maybe InputXY, Focus)
updateFocus = accum (0,0) f
  where
    validate (x,y) mov = case mov of
      Up   -> if x==0 then (x,y) else (x-1,y)
      Down -> if x==2 then (x,y) else (x+1,y)
      Lft  -> if y==0 then (x,y) else (x,y-1)
      Rt   -> if y==2 then (x,y) else (x,y+1)
    f op currFocus = case op of
      Nothing ->  (currFocus, (Nothing, currFocus))
      Just Enter -> (currFocus, (Just currFocus, currFocus))
      Just Quit -> (currFocus, (Nothing, currFocus))
      Just mov -> let n = validate currFocus mov
                  in (n, (Nothing, n))

emptyBoard :: Board
emptyBoard = map (const [N, N, N]) [(), (), ()]

updateBoardAndPlayer :: SF (Int, Int) GameState
updateBoardAndPlayer = proc (x, y) -> do
  rec result <- makeMove -< (x, y, player, exitGame)
      hasEnded <- checkStatus -< fst result
      exitGame <- delayedEcho False -< hasEnded
      player <- changePlayer -< snd result
      msg    <- giveMessage -< (fst result, player, hasEnded)
  returnA -< Game (fst result) player exitGame msgw
        where
          makeMove :: SF (Int, Int, Player, Bool) (Board, Maybe (Int,Int))
          makeMove = accum emptyBoard updateBoard
            where updateBoard (x, y, p, exitGame) b =
                    let modify b p x y =
                          let (rows1, row: rows2) = splitAt x b
                              (cols1, e: cols2) = splitAt y row
                          in rows1 ++ ( (cols1 ++ getSymbol p: cols2): rows2 )
                    in if exitGame then (b, (b, Nothing))
                       else case b !! x !! y of
                              N -> let b' = modify b p x y
                                in (b', (b', Just (x,y)) )
                              _ -> (b, (b,Nothing) )

          checkStatus :: SF Board Bool
          checkStatus = arr checkR &&& arr checkC &&& arr checkD >>> arr summarise
            where summarise (br, (bc, bd)) = br || bc || bd
                  checkR = foldl (\acc input -> acc || input) False . map
allTheSame
                    where allTheSame :: [Symbol] -> Bool
                          allTheSame [] = True
                          allTheSame [x] = True
                          allTheSame (x:y:xs) = x /= N && x == y && allTheSame
(y:xs)
                  checkC = checkR . transpose
                  checkD [[a,_,d], [_,b,_], [f,_,c]] = checkR [[a,b,c],[d,b,f]]

          changePlayer :: SF (Maybe (Int, Int)) Player
          changePlayer = tag notP >>> accumulate PX >>> hold PX

          giveMessage :: SF (Board, Player, Bool) [String]
          giveMessage = proc (b, p, hasEnded) -> do
            let
                m = if hasEnded then show (notP p) ++ " has won" else "continue"
            returnA -< [m]

game :: SF (Maybe KeyInput) (GameState, Focus, Bool)
game = (updateFocus >>> first (stepDown initialState updateBoardAndPlayer))
       &&& arr (\x -> x == Just Quit)
       >>> unwrap
       where unwrap = arr (\((a,b),c) -> (a,b,c))

initialState = Game emptyBoard PX False ["Welcome to Arrow Tic Tac Toe"]

dummy = constant (initialState, (0,0), False)

--a text UI
toString :: Board -> String
toString b = unlines $ map toString b
  where toString row = intercalate " | " $ map show row

parseText :: String -> (Int,Int)
parseText = convert . take 2 . fst . head . (reads :: ReadS [Int])
  where convert [a,b] = (a,b)
```

Main.hs

```
{-# LANGUAGE Arrows            #-}
{-# LANGUAGE OverloadedStrings #-}
module Main where

import           Control.Arrow
import qualified Control.Category as Cat
import           Control.Monad
import           Data.Maybe

import qualified Graphics         as G
import           Lib
import           Logic
import           SDLInput

import           Foreign.C.Types
import qualified Graphics.UI.GLUT as GL
import           SDL
import qualified SDL.Font

main :: IO ()
main = do
  initializeAll
  SDL.Font.initialize
  window <- createWindow "My SDL Application" defaultWindow
  renderer <- createRenderer window (-1) defaultRenderer
  gameLoop renderer

gameLoop r = do
  font <- SDL.Font.load "/Library/Fonts/AmericanTypewriter.ttc" 40
  runSF processKeyboard (G.renderGame r font) game testQuit
  where testQuit (_,_,quit) = quit
```

Appendix B   Arrow Implementation

```
newtype SF2 a b = SF2 (([DTime] -> [a]) -> ([DTime] -> [b]))

--list style
instance Cat.Category SF2 where
  id = SF2 (\x -> x)
  (.) = dot
    where
      SF2 g `dot` SF2 f = SF2 (\inp -> g (f inp))

instance Arrow SF2 where
  arr f = SF2 (\inp -> map f . inp)

  first (SF2 f) =
    let
      stripFst :: ([DTime] -> [(a,c)]) -> ([DTime] -> [a])
      stripFst inp = \dts -> map fst (inp dts)
      stripSnd inp = \dts -> map snd (inp dts)
      pairF one two = \dts -> zip (one dts) (two dts)
    in SF2 (\inp -> pairF (f (stripFst inp)) (stripSnd inp))

instance ArrowLoop SF2 where
  loop (SF2 f) =
    let zipF one two = \dts -> zipLazy (one dts) two
        --stream ~[] = []
        stream ~(x:xs) = x: stream xs
        stream3 ~(x:xs) = x: xs
        stream2 xs = let y:ys = xs in y:(stream2 ys)
    in SF2 (\inp dts -> let (bs, zs) = unzip $ f (zipF inp (zs)) dts
                        in bs
           )

--zipLazy [] _ = []
--zipLazy _ [] = []
zipLazy (x:xs) ~(y:ys) = (x,y) : zipLazy xs ys

runSF2:: SF2 () Double -> [DTime] -> [Double]
runSF2 _ [] = []
runSF2 (SF2 f) dts = f (const (repeat ())) dts

integralS:: Double -> SF2 Double Double
integralS i = SF2 (\inp dts -> let inputs = zipWith (*) dts (inp dts)
                               in scanl (+) i inputs)

--top level definition
test2 :: SF2 () Double
test2 = loop $ second (integralS 1) >>> arr (\(a,b) -> (b,b))
```

Appendix C Stackoverflow Question and Answer

Here is my question:

Automatically inserting laziness in Haskell

Haskell pattern matching is often head strict, for example,f (x:xs) = ... requires input list to be evaluated to (*thunk* : *thunk*). But sometimes such evaluation is not needed and function can afford to be non-strict on some arguments, for example f (x:xs) = 3.
Ideally, in such situations we could avoid evaluating arguments to get the behaviour of const 3, which could be done with irrefutable pattern: f ~(x:xs) = 3. This gives us performance benefits and greater error tolerance.
My question is: Does GHC already implement such transformations via some kind of strictness analysis? Appreciate it if you could also point me to some readings on it.

Here is an answer:
As far as I know, GHC will never make something *more lazy* than specified by the programmer, even if it can prove that doesn't change the semantics of the term. I don't think there is any fundamental reason to avoid changing the laziness of a term when we can prove the semantics don't change; I suspect it's more of an empirical observation that we don't know of any situations where that would be a really great idea.
(And if a transformation *would* change the semantics, I would consider it a bug for GHC to make that change.)
There is only one possible exception that comes to mind, the so-called "full laziness" transformation, described well on the wiki. In short, GHC will translate

```
\a b -> let c = {- something that doesn't mention b -} in d
```

to

```
\a -> let c = {- same thing as before -} in \b -> d
```

to avoid recomputing `c` each time the argument is applied to a new `b`. But it seems to me that this transformation is more about memoization than about laziness: the two terms above appear to me to have the same (denotational) semantics wrt laziness/strictness, and are only operationally different.

Appendix D Transfer Function Implementation

```
module AES where
import Control.Arrow
import qualified Control.Category as Cat
import Control.Monad
import CLaSH.Prelude
import CLaSH.Signal.Internal

newtype SF a b = SF {
     unSF :: a -> (SF a b, b)
}
instance Cat.Category SF where
     id = SF $ \a -> (Cat.id, a)
     (.) = dot
         where
             (SF sf2) `dot` (SF sf1) = SF $ \a ->
                 let (sf1', b) = sf1 a
                     (sf2', c) = sf2 b
                 in (sf2' `dot` sf1', c)
instance Arrow SF where
     arr f = SF $ \a -> (arr f, f a)
     first (SF f) = SF $ \(b,d) ->
         let (sf, c) = f b
         in (first sf, (c,d))

liftSF:: SF a b -> ((Signal a) -> (Signal b))
liftSF (SF f) = \(x :- xs) ->
    let (sf', out) = f x
    in (out :- (liftSF sf' xs))

liftF :: (s -> i -> (s,o)) -> s -> SF i o
liftF f acc = SF $ \inp ->
     let (acc', out) = f acc inp
     in (liftF f acc', out)

aes_core_sf = liftF aes_core 0
aes_cntx_sf = liftF aes_cntx 0

aes_top_sf :: SF (Bool, Signed 32) (Signed 32, Bool)
aes_top_sf = first aes_cntx_sf
    >>> arr (\((accept,done), plain) -> (done, (accept,plain)))
    >>> second aes_core_sf
    >>> arr (\(a,b) -> (b,a))

aes_top2 = liftSF aes_top_sf

aes_top :: Signal (Bool, Signed 32) -> Signal (Signed 32, Bool)
aes_top input =
    let (start, plain_text) = unbundle input
        (accept, done) = unbundle (mealy aes_cntx 0 start)
        cipher_text = mealy aes_core 0 (bundle (accept, plain_text))
    in
        bundle (cipher_text, done)

lut :: (Signed 32) -> (Signed 32)
lut input = case input of
    1 -> 2
    2 -> 4
    3 -> 3
    4 -> 9
    5 -> 10
    6 -> 33
    7 -> 1
    8 -> 4
    _ -> 0

aes_core :: (Signed 32) -> (Bool, Signed 32) -> (Signed 32, Signed 32)
aes_core res (accept,plain_text) = (res', cipher_text)
    where
        res' = case accept of
            True -> lut plain_text
            False -> lut res

        cipher_text = res

aes_cntx:: (Signed 32) -> Bool -> (Signed 32, (Bool,Bool))
aes_cntx state start = (state',o)
    where
        state' = case state of
            0 -> if start then 1 else 0
            3 -> 0
            x -> x + 1

        o = case state of
            -- (accept, done)
            1 -> (True, False)
            3 -> (False, True)
            _ -> (False, False)

x = [(True,0),(False,2),(False,2),(False,2),(False,2)]

--topEntity is top level, must be
--monomorphic
--first-order
topEntity = aes_top2
```