Tatu Tolonen

# Functional Reactive Programming in robotics

School of Electrical Engineering

Espoo February 19, 2016

**Thesis supervisor:**

Prof. Ville Kyrki

Aalto University
School of Electrical
Engineering

# Contents

# Symbols, abbreviations and terminology

## Symbols

| | |
|---|---|
| $:$ | "is type of" operator for function or value |
| $\rightarrow$ | Function type theory operator. Denotes function to be a mapping from domain left of the arrow to a codomain right of the arrow. |
| $\circ$ | Function composition operator. |
| $Behavior_\alpha$ | Polymorphic datatype named "Behavior " with a type variable $\alpha$. |

## Terminology

| | |
|---|---|
| **Domain** | A mathematical term, the input set of a function. |
| **Codomain** | Codomain is also known as range of function. Codomain is the output set of a function. |
| **Denotational semantics** | An approach for formalizing the meaning of computer programs by connecting programming expressions to a mathematical constructs. |
| **Domain Specific Language** | Programming language that is for general purpose, but designed for a specific use case. In this paper, Fran is a DSL for describing animations and interactive animation. |
| **Embedded Domain Specific Language** | EDSL is a domain specific language which is implemented as a library in host programming language. |
| **Tuple** | A finite ordered list of elements in mathematics and programming. |
| **Polymorphic data type** | A data type capable of being more than one type. |
| **Monad** | Type class for polymorphic data types. For type to implement a monad, it should have function *return* and bind operator, which uphold the monad laws.[1] |
| **Type class** | Description of a set of functions which are applicable for a type which implements the type class. For example Num type class states that the type should have operators plus, minus, multiply, negation, abs, signum and fromInteger. Int, Float, Double and many other types implement this type class. |

# 1 Introduction

The most common and arguably the most practical method for implementing a robot behavior is through software development. A software source code written in a programming language describes the software behavior accurately.

Programming paradigms are a general design approaches which guide the software design process. Two of the most common programming paradigms today are imperative programming and object oriented programming[2]. In imperative programming paradigm, software is thought as a set of instructions, which affect the state of the computer and the external world around the computer. This paradigm is very crucial as all microprocessors and micro controllers operate with a small set of imperative instructions. Historically this has therefore been the default way of thinking how to compose software. However, while these paradigms are most common in software development of today, there are other paradigms which can facilitate the development of complicated robot software system.

In this seminar paper I will concentrate on Functional Reactive Programming[3], which is a programming paradigm combining functional programming methods with reactive programming paradigm. There exists two common approaches on how to come to the motivation of coming up with the idea of functional reactive programming. First approach is with functional programming, and wanting to extend the good properties of functional programming to the realm of reactive systems. Second approach is from the reactive systems. Many existing reactive programming languages are practical and usable for developing reactive systems, but these languages are constrained. By introducing functional programming features to existing reactive languages, one can find more generalized programming interface for all the reactive systems needs.

Therefore, both of the functional programming and reactive programming paradigms are discussed before examining functional reactive programming. The topics will be discussed in sections 2 and 3 respectively. Functional programming discussion is spoken in the context of Haskell programming language as many of the FRP implementation have been made as an embedded domain specific languages into Haskell, as a programming library. As there exists plenty of iterations of different FRPs, this paper only reviews those that are relevant in robotics context in section 4.

While in this paper most crucial aspects of Haskell are explained for functional reactive programming purpose, this seminar is not a tutorial on Haskell programming. Refer to learning materials such as [4] for further information.

Robotics aspect is discussed in section 5. There we look into how functional reactive programming has been used in robotics settings.

## 2 Functional programming

In this section we review functional programming, especially the features seen in the Haskell programming language which are required for understanding functional reactive programming. Haskell[5] is one of the most commonly used pure, lazily evaluated programming languages. The definitions of these terms and their implications are explained in this section.

The key concept in functional programming is the notion of function being similar to function in mathematical context. In other words, function is a mapping of a value from a domain to a codomain. If a programming language is restricted just to this interpretation, it is called pure functional language. A concept of domains, codomains and function composition are shown in figure 1.

If the language permits an extensions to the interpretation of function that inside the funtion there can be a notion of mutable value, language is categorized as impure functional language.

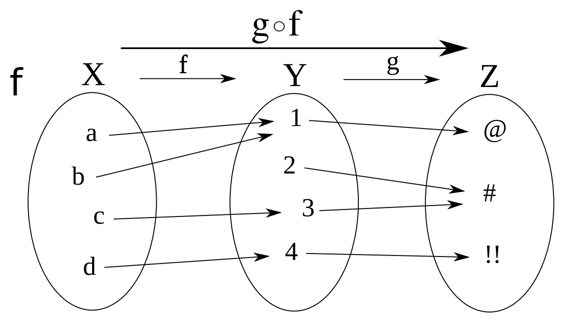

Figure 1: Functions $f$ and $g$ and their composition $g \circ f$.

Implications of purity restriction are wide. For example, loops commonly seen in imperative languages require to use a counter variable, or a condition variable, which is modified for each iteration of the cycle. In Haskell, these types of execution loops are implemented using recursion, where the loop is a function with parameters stating the counter or condition variable. Performance wise type of program is optimized by the compiler in case the recursive function is the last call made inside the function. This optimization is called tail call optimization (TCO).

Another implication of immutability is the implementation of many of the common data structures. In pure functional language, an operation on a data results in a new data of the same type. In trivial cases, such as shown in equation 1, the creation of an extra Integer does not cause much extra computation when compared to variable mutation. Example of a type definition of a function which takes an integer as an input and returns an integer. Notation is from Haskell programming language.

$$Increment :: Int \to Int \tag{1}$$

However, when taking a case of appending an large array by a single variable, making an extra copy of the whole array just for a small change is a quite large hit on the performance. Strategy for solving this is to change the data structure to something that has similar performance characteristics and is persistent. A persistent data structure stores its previous versions of itself. [6]

Compared to imperative programming language, where interaction with the external systems is mainly done by side effects, it is not trivial for purely functional language to, for example, write data into a file or to read it. In fact, a pedantic interpretation of pure functinal program would limit the computation not having any external inference. For practical use Haskell has decided to integrate the impure effects to the pure Haskell by the use of monads [1]. In similar fashion, efficient data structures can also be implemented with similar constructs, e.g. using ST monad[7].

A great charcteristic of functional programming is the usage of first class and higher-order functions. A language that support first class functions can operate functions the same way values are operated. Higher-order functions are functions which can have functions as their argument and can return a function as a result. These constructs help the programmer to define operations on higher level, reducing the code size. For example, a function

$$foldl :: Foldable\ t \Rightarrow (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow t\ a \rightarrow b \tag{2}$$

provides a functionality to sum number with $foldl\ (+)\ 0$, calculate product with $foldl\ (*)\ 1$, find maximum with $foldl\ max\ minBound$ and many others.

# 3 Reactivity and Reactive Programming

## 3.1 Reactivity in imperative programming

In imperative languages reactivity can be programmed with callback structure. Callback is a piece of program executable, which is executed each time an event occurs. These could be executed with either synchronously or asynchronously. In object oriented programming, observer pattern matches most closely to the structure of callbacks.

In fully synchronous setting, events are initiated with the help of an event loop, shown in the figure below. In the loop, the events are polled from the soureces, and these are ordered into an event queue which is then processed, calling each appropriate callback. In asynchronous setting, the event queue can also be filled asynchronously.

Here we will begin to see some difficulties which this approach does not answer. Consider a case where two callbacks are connected to a same event. As each of the callbacks change the global state of the system, and the operation being dependent on the global state, the order in which the callbacks are executed affect the outcome.

Callbacks can and in many points have to initiate additional events to the system. This creates a data flow graph, which has large implications on system behaviour. However, only way to read the graph is to analyse each callback function source code if it sends events and in which conditions.

Asynchronous callbacks are initiated when the task is known to be time consuming, for example by doing heavy computation or by interacting with some external entity, such as making a query to a server.

## 3.2 Reactive programming paradigm

Reactive programming is a programming paradigm intended for dealing explicitly with the reactivity. Intuitive way to understand it by examples. Consider a simple case where a variable $a$ is defined as $a = b * c$. This can be seen as $a$ is depending on $b$ and $c$. If $b$ changes, the change should propagate to $a$. These types of definitions of variables and their dependencies form a graph.

In reactive programming, programmer models the software as a graph, where nodes are are computations and the directed links are the data flows from the output of a process to an input of another process. Thus, the programmer is not programming the memory management of the graph and its values. Many of the implementations of the paradigm also provide a range of operations for computations, such as switches, merges and delays.

Spreadsheet such as Excel is a good example of a programming interface, where each sell of a spreadsheet is a variable. Variables can be constants, written by the user, or they can be expressions such as calculations depending on other cells. Changing a value in a cell automatically changes the results of the other cells that depend on it.

It depends on the implementation what types of values can be in a variable. Basic types, and how they are handled. Most of the languages support integers, floating point numbers, characters and groups of theses. Implementations of reactive programming languages might have limitations which are not present in generic programming languages

In automation, MATLAB Simulink[8] is commonly used graphical programming environment for simulation of dynamic systems. Control system code can also be generated from the drawn model. In figure 2 below an example Simulink controller shown.

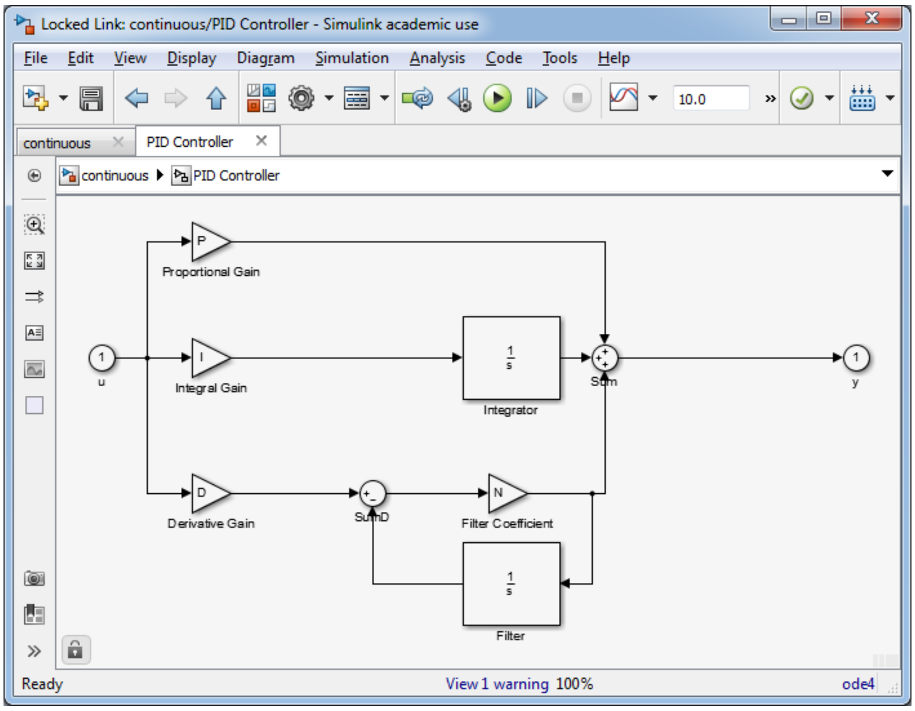

Figure 2: PID control system drawm in MATLAB Simulink

Memory usage and computational efficiency are always present in reactive systems. In control systems which have hard real-time requirements, predictability of a resulting control system is preferred. However methods for increasing predictability, such as minimization of dynamic memory allocation during execution, can affect the memory usage negatively. In Simulink for example, the generated C code operates by reserving memory for every computational block, and recomputes every value of the graph, even in cases where a switch block would allow a branch of flow to be discarded, and some values to be skipped during computational cycle.

## 4 Functional Reactive Programming

Functional reactive programming (FRP) is a programming paradigm which was originally created for declarative representation of animations, and was introduced alongside Fran (Functional Reactive animation), an implementation for the paradigm. It can be said that FRP today has several definitions as the original author of Fran defines FRP as denotationally[9] approached field for continuous time programming[10] while many other practical implementations which state to be FRP do not follow the denotational approach[11][12].

As original author of Fran used it on animation purposes, and was later formalized to a more general purpose reactive programming use[13][14]. As that first implementation of FRP was named FRP, but it also shares the name with the paradigm, I will use Fran to mean the first implementation and its semantics, and FRP to mean the programming paradigm.

Fran bases its idea to a concept of time varying values which are called behaviors, and events which are a values that happen at single point of time. Time in Fran was considered as a set of real numbers , $\mathbb{R}$. A behavior was semantically stated as a function from a time to a value, as shown in equation 3, where $\alpha$ denotes the type of of the value. Events were denoted as a list of time and value pairs, shown in equation 4. For events the time is exteded with $-\infty$ and $\infty$.

$$Behavior_{\alpha} : T \rightarrow \alpha \tag{3}$$

$$Event_{\alpha} : [\hat{T}, \alpha] \tag{4}$$

$$\hat{T} : T \cup \{\infty, -\infty\} \tag{5}$$

A ball affected by an external force can work as a small example use of Fran. In Fran, this behavior could be written as shown in figure 3

```
s,v :: Behavior Real
s = s0 + integral v
v = v0 + integral f
```

Figure 3: Code snippet for simple Fran code depicting a ball affected y an external force. s0 and v0 are initial position and velocity, and f is a behavior described elsewhere. Code is written in Haskell[15]

Fran code captures declaratively the behavior of the differential equations below matching the scenario.

$$s(t) = s_0 + \int_0^t v(t)$$

$$v(t) = v_0 + \int_0^t f(t)$$

## 4.1 Deficiencies of Fran

It was later discovered that the semantics of FRP described originally were computationally unpractical [15] [16]. The key problem the users faced was the memory efficiency. One of the key aspect of FRP is to provide programmer an interface where whole behaviors are programmed, and the handling of historical data is left for the inner workings of the FRP library and the runtime system (Fran was written for Hugs Haskell interpreter, which had garbage collector).

Ploeg and Claessen categorize [17] memory leak problems in three ways:

1. Leak is caused by the application software using the FRP library.
2. Implementation of the FRP library causes the the leak.
3. FRP is inherently leaky on the programming interface.

In the resulting system all types of memory leaks are seen similarily: memory allocations done by the software grows during execution until the computer runs out of memory, or an event launches a computation which results in computation after which the allocated memory can be released by the garbage collector. This behavior of the FRP puts emphasis on verifying the correct behavior of the underlying FRP implementation and its semantics.

Function shown in equation 6 describes an event which is produced at the same time the event given in argument is produced. The content of the resulting event is a sample of the behavior given as argument at that time. Practical use for this could be for example in graphical user interface where user's mouse position should be acquired when mouse is clicked. This has been proved to leak memory, making Fran inherently leaky.[17]

$$Snapshot : Behavior_\alpha \rightarrow Event_{()} \rightarrow Event_\alpha \tag{6}$$

## 4.2 Yampa

Yampa is a functional reactive language implemented in Haskell as an EDSL. Yampa differs from Fran by not having inherent time and space leaks. This is achieved by constraining the computational model such that the developer is not handling behaviors as first class values. Instead, developer can only use and program signal functions, similar to how in Simulink user does not describe the signal behaviors, but the transformations inside blocks.

These signal functions, noted as $SF\ a\ b$ in Haskell, can be combined with a set of combinator functions, shown in figure 4. The most simple of these SF combinators is $arr$, which allows the developer lift a function $f : A \rightarrow B$ to a data type $SF\ A\ B$. Other common signal function is $integrate : SF\ A\ B; A, B \in Vectorspace$, which integrates the input value with a rectangle method approximation. Other approximation commonly used in FRP implementations is Euler method. *Vectorspace* is an Yampa type class for scalar and vector types which implement basic arithmetic operations, such as sum and divide.

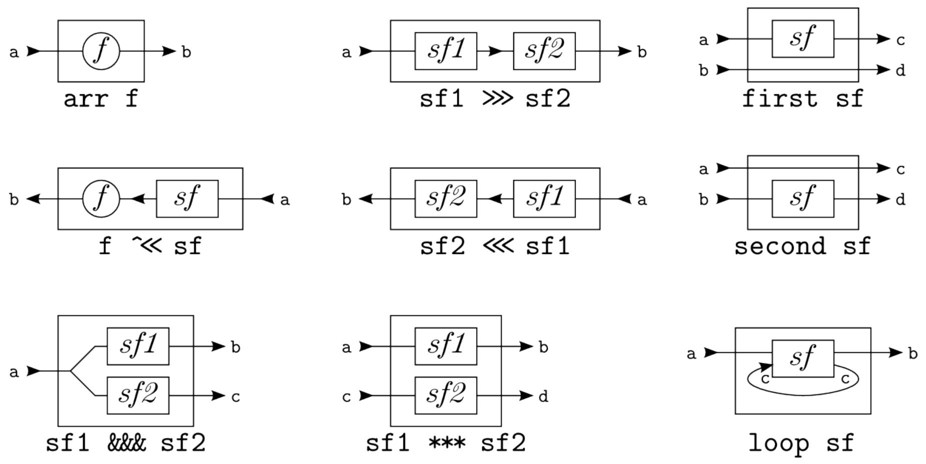

Figure 4: Signal function combinators of Yampa.[18]

Behaviors and Events in Fran semantics are changed in Yampa. In Yampa, there exists only Signal, which would carry both continuous time varying values, and Events of which exist in a single point in time. For that reason Events are similar to a polymorphic data type *Maybe* $\alpha$, where there either exists a value $\alpha$ or otherwise *Nothing*. Thus, signal functions in Yampa are operate on event streams, and Yampa provides many different types of switches for using event streams. Switches can be used as name suggests, changing the outputting signal based on received events.

### 4.2.1 Deficiencies of Yampa

After a signal graph has been contructed, it can be executed. Yampa provides a *reactimate* function which attaches a signal function to an IO operations providing input and output to the signal function. This cycle of 1. execute input IO, 2. evaluate signal function 3. exeucte output IO, is repeated as long as the 3rd step returns false. This reminds of the stream based IO used in haskell before the monadic IO, and shares the same problems:

- Extending the behavior of the system requires changing the wrapper IO functions.
- Connection between signal function and the IO is in cases too separated. Example case of a SF requesting to open a file handle a special use of unique identifiers is required.
- Routing all inputs of SF to the IO input and all outputs of SF to IO output has to be done always manually.

Yampa and arrowized functional reactive programming do share their own criticism. In the author mentioned that using the generalised arrow interface it will be problematic with data differentation. Also, evaluation model of Yampa is demand based, which ties event latency with the sampling rate.

# 5 Functional Reactive Programming in Robotics

## 5.1 Robot paradigms

Robot control has three behavioral paradigms on how to approach the requirement of making robot both responsive to its environment and become capable of doing its task or mission.

First and the oldest one is deliberative paradigm, also called hierarchical or Sense-Plan-Act paradigm. In it robot control should operate on three major phases sequentially, namely sense, plan and act phases. This paradigm turned out to have severe problems when the time taken by the planning phase affected the robots operations.

Second large paradigm, called the reactive paradigm, approaches the issue of long planning phases by only considering the next appropriate action to be made, removing the planning of the overall task. In a sense this removes the plan phase altogether as it becomes a function mapping from sense phase to action. In this manner, for a robot to succeed in its task, the behavior should be carefully designed such that reaching the task objective is inevitable. Designing it such is considered very hard, as plans can and will become complicated.

Third paradigm paradigm is called hybrid paradigm, trying to combine the best of the two previous paradigms. In it, there exists the reactive part of mapping sensed environment to a immediate next action, and it is monitored and controlled by a deliberative planning component. Control is done by modifying the parameters of the reactive parts, commmonly named as schemas (name comes from behavioral research), and by enabling and disabling the parts and/or rules.

## 5.2 FROB

Functional Robotics (FROB) was a framework for developing robot control software, and it was based to original FRP semantics as Fran language. In the following code snippet 5 FROB was used to implement a simple wall following control, which matches with the differential equations 7 and 8 describing the intended control.

$$v = \sigma(v_{max}, f - d*) \tag{7}$$

$$\omega = \sigma(sin(\theta_{max}) * v_{curr}, s - d*) - \dot{s} \tag{8}$$

In FROB, the programmer was tasked with describing Tasks, which are tuples consisting of a $Behavior_{robotwheelcontrol}$ and Event describing the ending condition of the task. These tasks could be combined with other tasks, for example making tasks executed in sequence. For ease of use, tasks can also be merged together to be run in parallel, ending with the first event condition is active first.[19]

```
wallFollow v_curr s f d_star =
 (v, omega)
  where
   v          = limit v_max (f - d_star)
   omega      = rerror - derivative s
   rerror     = limit (v_curr * sin theta_max)
                      (d_star - s)

limit high x = max (-high) (min x high)
thetamax = 10 # degrees
vmax     = 50 # cm_per_sec
```

Figure 5: Code snippet describing wall follower controller

FROB was capable if implementing commonly known reactive robotics architecture, such as subsumption architecture and motor schemas. This flexibility combined with the mathematical nature of FRP and Haskell provide a development interface which is easier to reason about and yet be expressive.[20]

FROB had weaknesses compared to other robot programming languages, which Pembeci et. al noted in [20]. FROB is based on Haskell, thus it uses garbage collection. This causes problems in cases where there real-time requirements as unexpected garbage collection activation can cause unexpected computation delays. FROB also does have only weak concurrency model providing concurrency to only tasks that are executed in parallel, which is fairly limited. FROB code might become a bit more complicated because of its exception handling mechanic of task ending events. Also FROB does not have a shared global database for different tasks of data, which also makes code more complicated.

Pembeci et. al. [21] upgraded FROB with arrowized notation. They demonstrated robot being able to do object tracking and following tasks. Camera was integrated with C++ XVision 2 library, which handled all the heavy computation associated with machine vision. XVision 2 is tightly coupled library with FVision, which was a Haskell library for machine vision. Authors used HVision to describe the machine vision algorithms, which allowed them to create a well performing platform for robot prototyping.

## 5.3 Yampa in robotics

Other than the research done by Pembeci et. al. mentioned earlier, arrowized FRP has been used in simulation purposes [13]. This setup is close to FROB description, with tasks describing macro level decision planning, which the developer is able to combine using monad interface. While in original FROB implementation developer described a Behavior which was inside each task, in Yampa the key robot behavior was described as a single signal function *SF RobotInput RobotOutput*. This setup shares the same advantages and disadvantages of FROB system.

# 6 Conclusions

In this seminar paper we've examined Functional reactive programming paradigm, and how it has been used in the robotic control context.

Functional reactive programming (FRP) paradigm appeared as a programming paradigm which provides a structured and general way of describing reactive systems. Some of the FRP implementations have their basis in denotational semantics, which has been used to both prove usability of the implementations and their weaknessess. Historically the main weakness in many of the FRP systems has been a leaking memory problem, where the program unexpectedly stores historical data read by the program in hazardous amounts. Improvements have been made over the years, some which reduce the programming interface, and some which tweak the semantics.

Functional Reactive Programming has been utilized in laboratory setting with a differential drive robots running on floor. A robot programming language called FROB has been created which has been used to describe robot behavior in a declarative manner. FROB has had its basis in semantics used in Fran language, and later has been upgraded to use arrowized notation similar to Yampa language. FROB turned out to be expressive, where describing reactive robot architectures was relatively straight forward, and creation of finite state automata strategies mapped well with the task monad structure.

FROB has received criticism due to its weak concurrency model, unpredictability of computation time. While there have been aims on making FRP systems with hard real-time capabilities, those were not covered in this seminar paper.

## 6.1 In future

I will be doing my master's thesis on the topic of functional reactive programming usage in robotics. During this seminar we have seen that there are better implementations solving memory leak problems, such as FRPNow! implementation. Also there were valid critiques to FROB structure which I try to search a better alternative solution, or at least my version of the FROB.

## References

[1] P. Wadler, "Monads for functional programming," in *Advanced Functional Programming*, pp. 24–52, Springer, 1995.

[2] T. Group and others, *TIOBE Index for ranking the popularity of Programming languages.* 2013.

[3] C. Elliott and P. Hudak, "Functional reactive animation," in *ACM SIGPLAN Notices*, vol. 32, pp. 263–273, ACM, 1997.

[4] C. Allen, "Chris Allen's open source guide for learning haskell, available at URL: https://github.com/bitemyapp/learnhaskell," Feb. 2016.

[5] "Haskell Language," Feb. 2016.

[6] C. Okasaki, *Purely functional data structures.* Cambridge University Press, 1999.

[7] "Monad/ST - HaskellWiki."

[8] J. B. Dabney and T. L. Harman, *Mastering simulink.* Pearson/Prentice Hall, 2004.

[9] D. Schmidt, *Denotational Semantics: A Methodology for Language Development.* 1979.

[10] Conal Elliott, "Conal Elliott on FRP and Denotational Design," Nov. 2014.

[11] R. Trinkle, "reflex: Higher-order Functional Reactive Programming," 2015.

[12] Ertugrul Söyleme, "netwire: Functional reactive programming library | Hackage," 2014.

[13] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, Robots, and Functional Reactive Programming," vol. 2638 of *Lecture Notes in Computer Science*, (Oxford University), pp. 159–187, Springer-Verlag, 2003.

[14] H. Nilsson, A. Courtney, and J. Peterson, "Functional reactive programming, continued," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 51–64, ACM, 2002.

[15] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *ACM SIGPLAN Notices*, vol. 35, pp. 242–252, ACM, 2000.

[16] W. Jeltsch, "Signals, Not Generators!," *Trends in Functional Programming*, vol. 10, pp. 145–160, 2009.

[17] A. v. d. Ploeg and K. Claessen, "Practical principled FRP: forget the past, change the future, FRPNow!," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pp. 302–314, ACM, 2015.

[18] "Yampa - HaskellWiki."

[19] G. D. Hager and J. Peterson, “Frob: A transformational approach to the design of robot software,” in *Robotics Research - international symbosium*, vol. 9, pp. 257–264, 2000.

[20] I. Pembeci and G. Hager, “A comparative review of robot programming languages,” tech. rep., Aug. 2001.

[21] I. Pembeci, H. Nilsson, and G. Hager, “Functional reactive robotics: An exercise in principled integration of domain-specific languages,” in *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pp. 168–179, ACM, 2002.