



אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

הפקולטה למדעי ההנדסה
המחלקה להנדסת חשמל ומחשבים
Faculty of Engineering Science
Dept. of Electrical and Computer Engineering

פרויקט ההנדסי שנה ד'
Fourth Year Engineering Project

דו"ח מכין
Preliminary Report

כלי תכנות פונציונליים תגובתיים (FRP) עבור יישומים רובוטים

Functional Reactive Programming (FRP) development tools for Robotics applications

Project number:	s-2010-096 / p-2010-040	מספר הפרויקט:
Students (name & ID):	נעם לואיס 014740351	סטודנטים (שם ו ת.ד.):
Supervisors:	הוגו גוטרמן	מנחים:
Submitting date:	22.11.2009	תאריך הגשה:

Contents

1	Introduction	2
1.1	Motivation	2
1.2	What is Functional Programming?	3
1.3	Denotational Semantics	4
1.4	Denotational Design and functional programming	4
1.5	Functional reactive programming	5
2	Research proposal	7
2.1	Project name	7
2.2	Research Goals	7
2.3	Methodology	7
2.4	Expectations	8
3	Research Review	9
3.1	Functional Programming, and Haskell	9
3.2	Reactive Systems Programming	15
3.3	FRP - Functional Reactive Programming	18
4	Specification of the robotic system	24
4.1	Introduction	24
4.2	The controlling program	24
4.3	The Segway RMP 200	25
5	Evaluating FRP - testing the robotic system	26
6	Estimated Budget	27
7	Work plan	28
7.1	Stage 1 - Implementation using FRP	28
7.2	Stage 2 - Implementation using another reactive language	28
7.3	Stage 3 - Comparison	29
7.4	Stage 4 - Conclusion	29
8	Timetable	30
A	Lambda Calculus at a glance	32
A.1	Expressions	32
A.2	Reduction rules	32
A.3	Examples of Lambda Expressions	33
	Bibliography	35

List of Figures

3.1	Signal function as a graph	22
3.2	Alternative graphic model for composite signal function	22
4.1	General structure of the robot	24
4.2	Top-level design of the controlling program	25
8.1	Project time plan as Gantt chart	31

List of Tables

3.1	Comparison of languages for reactive programming	16
6.1	Cost estimate	27
8.2	Timetable	30

Abstract

Reactive systems are required to react continually to time-varying input. Construction of such systems using common programming tools is a complex task. It requires the programmer to deal with a host of implementation details while obscuring the high-level meaning of the design. Reactive programming aims to simplify this process and abstract away many of the implementation details. Specifically designed reactive programming languages often speak in terms of time-varying values, whether discrete or continuous. Non-reactive programming, in contrast, forces the programmer to deal with temporal value updates manually. Orthogonal to reactive programming is *functional programming* - a general approach that leads to higher modularity and conciseness, and has been successfully used for many applications. Furthermore, the paradigm of *denotational design* which emphasizes the semantic precision and clarity of a design, is most naturally implemented using functional programming. Unfortunately, most reactive programming languages do not take full advantage of functional programming and disregard semantic simplicity. FRP (functional reactive programming) is a research area aiming to unify functional and reactive programming while encouraging denotational design, and supplying an expressive, high-level semantic model for designing reactive systems. This project will evaluate FRP for the implementation of a robotic system. We will concentrate on the advantages of the semantic model of FRP and evaluate it in terms of expressiveness, modularity, and tractability. For the implementation we will use one of the several available FRP frameworks, and compare it to an equivalent implementation in either Simulink, Microsoft RDS or another reactive environment.

תקציר

מערכות תגובתיות הינן מערכות הנדרשות לתגובה מתמדת לקלט משתנה בזמן. תכנות של מערכות כאלו הינו תהליך מורכב הדורש מהמתכנת לטפל בפרטי מימוש רבים תוך כדי טשטוש התמונה הכוללת. תכנות תגובתי מנסה לפשט תהליך זה ולהסתיר פרטי מימוש לא רלוונטיים. שפות תכנות מיוחדות למערכות תגובתיות מאפשרות למתכנת לדבר בשפה של ערכים משתנים בזמן, רציף או בדיד. לעומתם, שפות לא-תגובתיות מכריחות את המשתמש לטפל באופן ידני בעדכון ערכים עם הזמן. בנפרד ובאופן בלתי-תלוי, תכנות פונקציונלי הינה גישה כללית לתכנות המובילה למודולריות ולתמציתיות רבה יותר במימוש אשר מיושמת בהצלחה במגוון אפליקציות. יתר על כן, הגישה של תכנון דנוטציוני (Denotational Design) המדגישה דיוק ופשטות סמנטית של תכנון המערכת, מיושמת באופן הטבעי ביותר בשפות תכנות פונקציונליות. אולם, רוב המודלים לתכנות תגובתי אינם מנצלים את היתרונות של תכנות פונקציונלי עד תום ואינם מתייחסים לפשטות סמנטית. תכנות פונקציונלי תגובתי (*Functional Reactive Programming, FRP*) הנמצא במחקר מנסה לשלב בין הגישה התגובתית לפונקציונלית תוך שימת דגש על תכנון דנוטציוני, ולספק מודל סמנטי עשיר ומופשט לתכנות מערכות תגובתיות. בפרויקט זה נבחן את היתרונות של FRP עבור בניית מערכת רובוטית. נתמקד ביתרונות של המודל הסמנטי של FRP ונעריך אותו במושגי הבעתיות (expressiveness), מודלריות, ופשטות הניתוח של התוכנית. עבור המימוש נשתמש באחד מהספריות הקיימות ל - FRP ונערוך השוואה למימוש שקול באחת מהסביבות Simulink, Microsoft RDS או בסביבה תגובתית אחרת.

Chapter 1

Introduction

1.1 Motivation

Reactive systems must continually react to time-varying input. This wide class includes systems such as robots ([7], [13]) and graphical user interfaces ([8]). Unfortunately, the construction of reactive systems using common programming tools is a disproportionately complex task. As programmers, we must deal with a host of low-level details while obscuring the high-level meaning of the design. A negative side effect is that we grow accustomed to inappropriate design methods. This practice is further encouraged by the limited set of abstractions offered by our programming environments. For reactive systems that deal exclusively with numerical-valued functions we can take advantage of *control theory*, which gives precise definitions and analysis. However, if we wish to use other abstract representations we fall back to the imprecise practices prevalent in software design ([2]). We gladly accept abstractions such as objects and message passing, processes and synchronization, control flow structures, etc. These abstractions *seem* well suited for designing systems, but they lack the important property of a *precise, tractable mathematical meaning* - the very feature of control theory ¹ that makes it so useful. Once we allow any of these abstractions into our design, we lose the ability to reason about it.

It is not only reactive software systems that suffer this problem - most software designs today are plagued by intractable models. This has been pointed out repeatedly by luminaries such as Dijkstra (for example, [5]) and John Backus ² (in his Turing award speech, see [2]). Most software designs can not be described precisely (mathematically) in a reasonable fashion, often even for very small parts of the design. The problem with the mathematical complexity of such models is twofold. First, as already stated, it makes them intractable (hard to analyze and understand fully). A second, possibly worse problem is that the added complexity also means loss of generality, which reduces modularity. We end up doing similar work again and again. Re-usability is harmed and so is the ability of others to easily understand what we are doing. Of course we can make an effort to think in terms of tractable abstractions, but the programming languages available encourage us *not* to do so (as an example, the obsession with “object oriented” design leaves little space for alternatives which are often semantically simpler). We would need to constantly translate our design and thinking into the irrelevant abstractions imposed by our programming languages. We therefore need better design methods, and also a way to implement such designs easily.

Let us return to the case at hand - reactive systems. What we want is an appropriate model for designing such systems, and a way to implement the designs without losing our grip on precision. In the rest of this introduction we shall first give a very short presentation of functional programming. Then we shall discuss denotational design, an approach that gives a precise meaning to every abstraction we use. Following that we will explain the notion of *functional reactive programming* (FRP, [8], [19]) which is the primary focus of the project. After this introductory section we shall expand with more detail these

¹ Control theory is just one example - we might as well mention communication theory, circuit theory, and of course physics, all of which use precise mathematical concepts to model systems.

² Backus was a programming languages pioneer, the inventor of Fortran and Backus-Naur Form for which he won the Turing award in 1978.

subjects. We shall also recall the existing tools for reactive systems programming and compare them to FRP.

The goal of this project is to evaluate FRP in comparison to other paradigms, and use it to design and implement a robot based on the Segway Robot Mobility Platform.

1.1.1 Design vs. Implementation

To make the discussion less vague, let us define *design* and *implementation* in the context of this project:

- The **design** of a system is a model, that describes the system in any abstract terms we may find convenient.
- The **implementation** of a system is the code that performs the desired operation.

Depending on the abstractions we use in the design, and on the ones that the programming language makes available to us, we may be able to write code very similar to our design. For example, if we design the system as a group of communicating objects that operate in a global sequential environment, it will be straightforward to implement it in an object oriented language. Otherwise, the code can be structured in an entirely different way that is not related to the model we designed. For example, designing a system as a data flow graph with concurrent paths and then implementing it in C. The language will lack the required primitives and we will end up writing a lot of unrelated code, to implement our design.

1.2 What is Functional Programming?

“functional programming” usually means a mix of several closely related notions:

- Programming that centers on pure, mathematical functions from input to output. For every input a function always produces the same output, regardless of global state.
- Programming without (or very little) mutability. Mutability means allowing variables to change their value, allowing any variable to embody *state*. The opposite is *immutability* which disallows stateful variables, and encourages recursion on substructures as an alternative.
- Programming languages that are based on the *lambda calculus*, a branch of mathematics that deals with the formal definition and evaluation of functions, using syntactic expression transformations. Appendix A provides a quick introduction to lambda calculus, and the uninitiated reader is encouraged to try it.

One of the great advantages of functional programming languages is the ability to do *equational reasoning* (see [3]). We can take *any* part of our program, and using a simple set of rules transform it into an equivalent expression, possibly simpler or more general. It is impossible (or very, very difficult) to do this on an arbitrary part of a program in non-functional languages such as Java and C++.

1.2.1 Practical functional programming languages

Functional programming languages extend the idea of lambda calculus to make it practically useful. They add a type system, recursion (without explicit usage of the Y-combinator), pattern matching definitions for functions, and much more. This project makes use of the **Haskell** functional programming language which has been used successfully for many applications ([16]). A simple functional definition of the factorial function in Haskell is:

```
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

We shall give an extended explanation with more examples in the section following the introduction. For the rest of this introduction we shall turn to the subject of denotational design principles, their connection with functional programming, and how this is used to build FRP (functional reactive programming).

1.3 Denotational Semantics

Design of software *languages* is a specific problem for which computer scientists (specifically, Scott and Strachey in [1]) have devised an appropriate mathematical tool: *denotational semantics*. This involves the definition of a *semantic function*, a mapping from syntactic expressions of a language to precise meanings in a mathematical model chosen by the language designer. The semantic function is marked $[[\cdot]]$, and its type is: $\text{Syntax} \rightarrow \text{Meaning}$. It is *compositional*, so that a syntactic expression is mapped into a meaning by decomposing it to map each of its sub-expressions (using the semantic function again), and finally using some mathematical definition to combine the sub-meanings. Thus, denotational semantics concentrates on *the precise meaning* of a syntactic expression in an ideal mathematical model, rather than describing how it operates.

As an example let's invent a simple (and useless) language for expressing numerical addition. The most basic syntax is that for numbers, which are written in decimal format using the digits 0–9 and the decimal point .:

$[[E]]$ = The numerical value of E when interpreted as a decimal number.

Now we can define syntax for expressing numerical addition. Let's say we'd like to express addition by placing the symbol `!+!` between two expressions. For example, `1243.3!+!432`. Here's where the semantic function begins to take action:

$[[E1!+!E2]]$ = $[[E1]]$ + $[[E2]]$

The expression inside the $[[\cdot]]$ on the left is in our language's syntax, whereas the expression on the right is in mathematical notation. The semantic function maps the syntax `E1!+!E2` to the value of a numerical addition: *the semantic meaning of E1 plus meaning of E2*. Of course, we could use `E1+E2` (instead of `!+!`) in our syntax, but that may make the example confusing.

For a more interesting example, consider the definition of a structure type in C (`struct`) according to the ISO standard ([25]):

“A structure type describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.”

This describes how a structure is represented (note the words “sequentially allocated...”) in the computer. We can invent the following hypothetical *denotational* definition of a structure type using a semantic function:

$[[\text{struct } T \{ T1 \text{ name1}; T2 \text{ name2}; \}]]$ = (“T”, $[[T1]]$, $[[T2]]$)

This definition maps `struct` types to mathematical n-tuples (in this case, a triplet) whose first element is the name of the type, and each following element is the *meaning* of a field type mentioned in the `struct`. There is a fundamental difference between what the ISO standard defines and the semantic definition that we invented. The semantic definition gives a precise mathematical *meaning* to the syntax. If we were designing a language similar to C, we could make the choice of whether to exclude the first element (the type name, “T”) in the tuple, or to include it. This is a decision on the *meaning* of our `struct`. The decision affects whether structure types that have the same field types but different names are equivalent, or not. Denotational semantics make the decision explicit. Another decision is whether structs with the same fields in different order should be distinct or not - this too is embodied in our choice of a mathematical n-tuple (for which order does indeed matter, unlike a set).

Denotational semantics is beyond the scope of this work, and we shall not deal with it further. It was introduced here as a prerequisite for discussing denotational design, which is the next topic.

1.4 Denotational Design and functional programming

Conal Elliott (in [18]) recently suggested, and explained how, to carry over the idea of denotational semantics to the design of software programs, for *denotational design*. The idea is to give precise meanings to the elements used in our software design (the data types), similar to what we do for syntactical

expressions in a language. We shall again use the semantic function $[[\cdot]]$ which now maps data types or programmatic interfaces to their appropriate semantical meaning.

In his paper, Elliott gives a good example to introduce denotational design. As a preliminary step, let us define the data type `Maybe a` (where `a` is a data type parameter), a data type whose values can have two possible structures:

- `Nothing` - a ‘singleton’ value that represents *no result*,
- `Just x` - a value containing the result `x` (which has type `a`).

For example, `Maybe Int` is the type of “maybe integers”. Some values of this type are: `Nothing`, `Just 0`, `Just 3`, and `Just -54`.

The example given by Elliott is that of the `Map k v` data type, the type of a dictionary with keys of type `k` and values of type `v`. The dictionary’s keys are a subset of all values of type `k`. The natural semantic meaning of this type is a *total function*, from all of `k` to `Maybe v` (notation: $k \rightarrow \text{Maybe } v$). The function maps non-keyed values in `k` to `Nothing`, and existing keys to `Just x` (with `x` being the value associated with the given key). We can write the semantic function of this data type:

$$[[\text{Map } k \ v \]] = k \rightarrow \text{Maybe } v$$

Now that we have a simple mathematical model (in this case, a total function), we can easily deduce the meaning for each of the primitive operations on maps. They are: *empty* which returns the empty dictionary, *insert k v m* - inserts a key `k` with value `v` into the dictionary `m`, and *lookup m k* - performs the actual lookup. Remember that for a dictionary `m` the semantic function $[[m \]]$ returns a *function*, as we defined above. Obviously,

- $[[\text{empty} \]]$ = a function that returns `Nothing` for any argument
- $[[\text{insert } k \ v \ m \]]$ = a function that, given a key argument `k'` maps:
 - the key equal to `k` to the value `v`, and
 - any other key by applying: $[[m \]]$ on `k'`
- $[[\text{lookup } m \ k \]]$ = the result of applying the function $[[m \]]$ on the key `k`.

Or, making use of the lambda-syntax:

- $[[\text{empty} \]] = \lambda x. \text{Nothing}$
- $[[\text{insert } k \ v \ m \]] = \lambda k'. \text{if } k = k' \text{ then } v \text{ else } [[m]] \ k'$
- $[[\text{lookup } m \ k \]] = [[m]] \ k$

The example is meant to illustrate how we can give a precise mathematical meaning to a data type, in this case the mapping data structure. Elliott’s paper goes much further and shows the many advantages gained by pursuing this approach.

1.4.1 Implementing a denotational design

Once we have designed our system denotationally, it’s crucial that the implementation will adhere the semantic model we have chosen. Obviously some languages are better suited for this task than others. Implementing a denotational design in C entails a large mental overhead of constantly translating our ideas into the limited expressiveness of C. The feature we want in our language is to make it easy to define data types resembling the mathematical meanings in our model. *Functional programming* with its focus on mathematical functions as well as powerful type systems greatly aid the proper implementation of denotational designs. In this project we will see several examples for why this is true.

1.5 Functional reactive programming

Let us return to the problem of programming reactive systems. *Reactive programming* languages try to simplify this process by offering abstractions that are specifically useful for reactive systems. However,

most reactive programming language suffer the same problems relating to semantic tractability described previously, because they involve abstractions that are complicated to model precisely. They do not take full advantage of pure functional programming with an expressive type system, such as the one offered by Haskell. Thus, they do not encourage denotational design or make it easy to implement one. A relatively new paradigm known as *functional reactive programming* (FRP) synthesizes the two approaches: reactivity, and the usage of functional programming for denotational design. FRP aspires to encourage thinking in terms of a clear semantic representation of the reactive system, and to make it easier to translate this into an implementation. This project will evaluate FRP for the implementation of a robotic system (as in [13]).

The semantic model that lies in the heart of any reactive system is very simple. It is a function from time to some type. For example, a temperature sensor is a function: $\text{Time} \rightarrow \text{Temperature}$. A system that translates temperature to an audible pitch can thus be modelled as $(\text{Time} \rightarrow \text{Temperature}) \rightarrow (\text{Time} \rightarrow \text{Pitch})$. We shall show how this way of thinking can be used to design a more complicated system, and argue that thinking in these terms leads to simpler and clearer designs for reactive systems. The project therefore concentrates on the usage of denotational design for a robotic system, and using an FRP framework to implement the design with minimal loss of clarity.

Chapter 2

Research proposal

2.1 Project name

Functional Reactive Programming (FRP) development tools for Robotics applications.

2.2 Research Goals

The objectives of this research project are:

1. The evaluation of FRP as a semantic model for a robotic system.
2. Estimate the advantages and disadvantages of FRP compared to other models of reactive systems.
3. Designing and implementing a robotic controller program (for a robot based on the Segway RMP 200) using FRP, and comparing the result to an equivalent implementation in an alternative reactive language (such as Simulink, Erlang, or Microsoft Robotics Studio).

2.3 Methodology

After a review of existing models for reactive systems programming, the research will be performed as follows:

1. Comparison of the FRP model with existing ones.
2. Testing FRP for implementing a simple simulated robot.
3. Implementation of a robotic system using an FRP framework.
4. Comparison of the FRP-based implementation to another that will use Simulink, Erlang or another reactive environment (to be determined). The comparison will check:
 - (a) Code size and conciseness.
 - (b) Modularity.
 - (c) Tractability (how easy is it to reason about the correctness of the code).
 - (d) Binary size.
 - (e) Since no satisfactory implementation of the FRP model exists, performance will not be examined, but it will be noted.

2.4 Expectations

Due to its functional nature we expect the FRP model to ease the design of the robotic system, compared to other models. However, because of FRP's young age we may find the available implementations inadequate, and we will probably encounter some trouble (such as bugs) in FRP's own implementation. However, we will try to show that in principal FRP is a good, concise model and has a promising future.

Chapter 3

Research Review

This section shall give a detailed explanation of the background for FRP, and the alternative paradigms for the design (and implementation) of reactive systems. We shall start by reviewing *functional programming*, which forms the basis for FRP, with a focus on Haskell.

3.1 Functional Programming, and Haskell

Functional programming ¹ (FP) treats programs as *mathematical functions from input to output*. There are no side effects such as modification of state variables ². As a result, functional programs are inherently more declarative, programming in FP forces you to think more about the meaning of your program. Functional programs are also easier to modularize and to reason about. FP languages are based on strong mathematical foundations, eliminating much of the feature redundancy present in other languages, and encouraging composability (which is the source of increased modularity). We shall focus our attention on a particular language - Haskell ³ - but the concepts are valid for many other functional languages as well. For a very thorough, interesting review of Haskell's history and features, see [16].

The term “functional programming” is often contrasted with “imperative” style programming. Imperative languages deal with the sequencing of state-modifying, destructive operations. Usually people refer to C++, Java, Python, etc. as imperative languages ⁴. However, the real difference between imperative and functional programming lies in the interfaces that they encourage. Imperative style encourages mutable objects as interfaces (“Object-oriented”) and precludes most other ways. Functional style encourages declarative, composable interfaces, which are cleaner and have more tractable semantics.

Finally, functional programming does not mean abandonment of the “list of statements” style. In fact, Haskell provides the tools to perform rich, elegant imperative programming. The idea will be explained in the section about Monad Transformers. This can be used to revert back to completely imperative programming, but the true benefit lies in exposing functional interfaces, in not abandoning the clean semantics that FP encourages.

3.1.1 Example 1: Working with lists

The best way to show why FP is great, is by examples (this example builds on [3]). We will start with some simple operations on lists. Here is an implementation of list summation in imperative pseudo-code, where a list is represented as a linked list:

¹ Thanks go to Eyal Lotem for mentoring me in Haskell and FRP.

² Of course, a useful application of functional programming must eventually perform some I/O. Haskell's library provides an abstraction (the Monad) to handle this and other issues without breaking the functional property of the program.

³ <http://www.haskell.org>

⁴ But as Conal Elliott pointed out satirically, even C is purely functional (<http://conal.net/blog/posts/the-c-language-is-purely-functional/>).

```

SumList L
  sum = 0
  while L != NIL
    sum += L.value
    L = L.next
  return sum

```

Here is the same function in naive FP. A list of integers in FP can be defined as follows:

```

data List = Nil | Cons Integer List

```

‘data’ is a keyword to define a new type. A List is either :

- Nil (an empty list), or
- Cons <integer> <rest of the list>

For example, [1,2,3] is really: Cons 1 (Cons 2 (Cons 3 Nil)). Now we can define the summation function:

```

sum Nil = 0
sum (Cons x xs) = x + sum xs

```

This syntax uses *pattern matching*. The function sum handles two inputs: Nil, or ‘Cons some number rest of list’. If the input is an empty list, Nil, sum returns 0. If not, it returns the current ‘head of the list’ added to the sum of the rest of the list. Notice that recursion is a natural tool in functional programming. In Haskell, Nil = [], and Cons is an infix operator, :. So we may re-write that example as:

```

sum [] = 0
sum (x:xs) = x + sum xs

```

Now, any infix operator (including + and :) can be used as a prefix function, enclosed by parenthesis, so we can write:

```

sum [] = 0
sum ((:) x xs) = (+) x (sum xs)

```

When applying a function on a value, instead of writing ‘f(x)’ in Haskell we write f x. Now, we may notice an interesting pattern. Our sum function simply replaces [] with 0, and (:) is replaced by (+) while we recurse on the rest of the list. We go from:

```

1 : (2 : (3 : []))

```

To:

```

1 + (2 + (3 + 0))

```

We can therefore write a generalized function, that does exactly this. We shall call it foldr (right-associative fold) and it will take a function f to replace the Cons (:) and a value x to replace the Nil []:

```

foldr f x [] = x
foldr f x (y:ys) = f y (foldr f x ys)

```

```

sum 1 = foldr (+) 0 1

```

In Haskell functions are usually *curried*. This refers to the fact that a function of several arguments doesn't really exist. Instead, applying the first argument returns a new function on which we can apply the second argument, etc. until the last argument's function returns the result (see the appendix about lambda calculus). The type of (+) on integers is `Integer → Integer → Integer`, rather than `(Integer, Integer) → Integer`. Thus, our `foldr` function can be called without the last argument (the list to fold) and the result will be a function, which takes only the list and returns the fold that we requested. So, we can simplify the definition of `sum` by eliminating the '1' argument (this is called η -reduction in lambda calculus):

```
sum = foldr (+) 0
```

Now we can compare `sum`, the FP version, with `SumList` defined above, the imperative version, and see how FP leads to cleaner code. We can also define `product`, which multiplies the list's values as follows:

```
product = foldr (*) 1
```

As a final example, consider a list of lists (such as `[[1,2,3],[2,4,1],[6,4,2]]`). We can easily concatenate the inner lists using `foldr`, using the infix operator `++` which concatenates two lists:

```
concat = foldr (++) []
```

The result of `concat [[1,2],[3,4]]` would be `[1,2,3,4]`.

These examples using `foldr` demonstrate both conciseness and modularity. Truth is that we *can* do this in some of the modern imperative languages - but in FP this is the usual, encouraged and easy way to work.

3.1.2 The Power of Types

Haskell is an FP language with strong, static typing. This means that at compile time, every expression must have a known type, and the types of functions and other expressions must match. This contrasts languages such as Python where types are determined dynamically, at runtime. Haskell also has *type inference*, which means that we don't really need to tell the compiler what is the type of each definition - in general the compiler automatically finds the most general type that fits. Thus, there is no need for cumbersome type specifications (unlike C++, for example).

Type Polymorphism and Type Variables

We've define the `List` type above to be specific for `Integer` values. Normally we won't do such a thing. Instead, we'll use a *type variable* to say that a `List` can be parameterized over any type:

```
data List a = Nil | Cons a (List a)
```

The definition defines a family of types, `List a`. The `a` is a type variable, and it's used on the right-hand-side when we define the `Cons` data constructor: `Cons a (List a)`. This means that `Cons` takes two arguments: one of type 'a', which depends on the type of the list we are working with, and another of type `List a`. In C++ we would call this a template and write it as `List<T>` (T replacing `a`). An actual list of `Integers` will have type `List Integer`, roughly corresponding to `List<int>` in C++.

Types can determine the Meaning

Let us take a look at a simple polymorphic type signature (for a much deeper discussion see [18]):

```
id :: a -> a
```

The function 'id' takes a value of any type, and returns a value of the *same* type (because the return type uses the same type variable of the argument). The function knows nothing about the type `a` because it's an unconstrained type variable - it can be *any* type. Thus, the only way for us to return a value of this type, is to use the same value that we were given. Therefore, `id` *must* be the identity function,

```
id x = x
```

The point here is that in many cases, by reviewing the type signature alone with no further information, we can tell exactly what the function does. Another example is `const`:

```
const :: a -> b -> a
```

Recalling currying, this type signature means that `const` takes two arguments and returns one. The types of the two arguments are not constrained, but the result type is the same as the first argument (the same type variable is used, `a`). Since we have no knowledge of what these types are, we must return the first argument, as the result. So `const` ignores its second argument, always returning the first argument. Indeed, `const` is the constant function.

```
const x y = x
```

```
f = const 5
f "Hello" -- will be evaluated to 5
```

The definition of `const` above uses a *lambda expression*, also known as an anonymous function. The syntax `\y → expr` means: a function that takes one argument, binds it to the variable `y`, and returns the expression `expr` (which may contain references to the variable `y`).

Here is a more interesting example. A parameterized type for maps may be: `Map k v` (where `k` is the type of the keys, `v` is the type of the values). Consider the following type signature:

```
f :: (a -> b -> c) -> Map k a -> Map k b -> Map k c
```

Looking at the signature we see that:

- Argument 1 is a *function* that takes two arguments (of types `a` and `b`) and returns a value of type `c`.
- Arguments 2 & 3 are Maps with corresponding values of types `a` and `b`.
- The result is a `Map k c`.

Obviously the only way to get `c`-type values is to apply the first argument to the values of matching keys in the two input maps. We conclude that `f` must be an intersection - it can only output values for keys that exist in *both* input maps.

Type classes

Type classes (not to be confused with Java/C++ style Classes), are a way to define interfaces. For example, the `Show` class defines a function, `show`, which formats a value into a `String`:

```
class Show a where
show :: a -> String
```

An *instance* of a type class is an actual, specific type (which may itself be parameterized, such as `List a`, which is really known as `[a]` in Haskell) that implements the interface required by the class. For example:

```
instance Show a => Show [a] where
show [] = "[]"
show (x:xs) = "(" ++ show x ++ ":" ++ show xs
```

Where `++` is used to concatenate Strings. The part `'Show a =>'` means, “constrain the type of `a` to be an instance of `Show`”. We do this here because we want to show also the individual values stored within the list. So to show a list of type `[a]`, we must require that `a` itself is an instance of `Show`. This instance definition allows us to run `show` on lists:

```
show [1, 55, 3] -- will be evaluated to: "(1):(55):(3):[]",
-- numbers formatted so because the default 'show'
-- for Integers is decimal notation.
```

Haskell allows constrained types also in function definitions. For example, we can require that the type of an argument to a function is an instance of `Show`.

Note: Haskell’s implementation of the `Show` class is slightly different, allowing for more elegant `show` outputs in some cases.

Monads

Purely function languages share a common problem: how to represent I/O? Languages such as Scheme simply forgo pureness and allow “magic functions” which perform I/O behind the scenes, without reflecting this in their type. Others, such as Miranda and other precursors to Haskell, sport continuation-style or stream-based approaches, which remain pure but result in code that’s much harder to understand. Haskell (since version 1.3 in 1996, see [16]) has embraced an approach first recognized by Wadler ([6]) using Monads. This approach results in readable, imperative-looking code that can be used for I/O, yet is purely functional. Thus, a central type class in Haskell is the *Monad*. A basic definition of the `Monad` type class, is:

```
class Monad m where
  (>>=)      :: m a -> (a -> m b) -> m b
  return     :: a -> m a
```

A `Monad` instance is a “computation”, built from a sequence of “computations” interlaced with functions on intermediate results.

- The function `return` takes a value and ‘builds’ an empty computation, containing only a value.
- `>>=` performs the sequencing and interlacing described above.

Many liters of (digital) ink have been spilled explaining the meaning of Monads, but we will go straight to an example.

3.1.3 Example 2: The Maybe Monad

Often, we write functions that deal with two possibilities:

1. A value/result of some type
2. No value or result

The second possibility can be used to express failure (like an exception), whereas the first signifies success and contains the value that we wish to use. We therefore define the following type:

```
data Maybe a = Just a | Nothing
```

The type `Maybe a` can be either `Nothing`, or just a value of type `a`. The `Maybe a` type is an instance of the `Monad` type class, and is a good example of the usefulness of Monads:

```
instance Monad Maybe where
  return     = Just

Nothing >>= f = Nothing
(Just x) >>= f = f x
```

Note: The above code includes two function definitions: `return`, and `>>=`. Haskell's syntax for function definitions allows the function name to appear infix (between the patterns) as demonstrated above for `>>=`.

- A “new” `Maybe` computation, constructed with `return` with some value `x`, is simply `Just x`. According to the Monad laws, this must be an empty computation. So we see that `Just x` is not considered an computation in the `Maybe` Monad, it really is “Just” a value. `Nothing` complements this by representing a failure computation, with no value.
- If we try to sequence a `Nothing` with some function (the ‘next step’ in the sequence), we get `Nothing` again. This corresponds to an exception in imperative languages - if `Nothing` (an exception) is encountered anywhere, the entire expression evaluates to `Nothing` (an exception).
- If we sequence a `Just x` with some function, we pass the value `x` to the function (which creates the next computation).

Haskell has a special syntax for working with Monads, the ‘do notation’ which allows the sequence of computations to be listed one per line, hiding the use of `>>=`. Here is an example:

```
find :: Key -> Maybe String
something :: String -> Maybe Char

computation = do
  x <- find key
  r <- something x
  return (x, r)
```

Without further details, we’ll only note that this example effectively implements error handling. If ‘find key’ fails (returning ‘Nothing’), the whole expression will return `Nothing`. The same applies for `something x`.

The `do` syntax makes it easier to write in the sequence-of-computations style of programming. We’ve used in-language constructs to implement something similar to exceptions, which is usually a built-in feature of a language. Similarly, Monads can be used to implement a variety of other extensions and features for imperative programming. For example, the `State` monad provides a state value which can be transformed between computations, allowing us to model stateful computation sequences. Haskell is therefore a basis for building imperative programs with exceptional extensibility that even full-fledged imperative languages don’t supply. The advantage is triple:

- We can extend our imperative programming language with arbitrary features,
- We can specify the minimal set of features our imperative sub-program uses, which makes it easier to analyze, and
- The features we use are specified explicitly.

Monad Transformers

As a final point, we’ll mention `Monad Transformers`. These allow use to combine Monads in arbitrary ways, effectively allowing us to mix & match language features to suit our needs. For example, combining ‘`State`’ and ‘`Maybe`’ monads gives us a language with exceptions and state. We can choose how to combine these two monads - an exception will preserve the current state (allowing us for example to resume an aborted for loop) or it will not (forbidding us to resume). In some cases we may want the first, and in others the latter, and Haskell gives us the power to choose. This level of extensibility is not available in any imperative language that I know of.

3.2 Reactive Systems Programming

3.2.1 What is a Reactive System?

As previously explained, a reactive system is one that is required to continuously react to time-varying input. Reactive systems are in essence *non-terminating*, often they are concurrent, and react to perpetual *external* stimuli whose source may be the physical world or another reactive system - rather than finite input data. In contrast, non-reactive systems transform input into output with no concept of a changing, time-dependent input. Sometimes, when we sketch out the design of a reactive system we begin by drawing a data flow diagram describing input streams entering the system, and how they are transformed and manipulated to yield response streams. This is why reactive programming is closely related to “data flow programming”. In the general case, however, there is some logic in the system that isn’t (or can’t be) naturally modelled as a flow of data. Even in such cases where the top-level design clearly appears as a data-flow system, at the lower level we need to somehow implement the “boxes” through which data flows. For this reason we need more than just simple ‘transcoding’ transformations.

A few examples of reactive systems are:

1. **Graphical User Interfaces** need to react to the user’s perpetual mouse-fiddling and keyboard-bashing. The usual way of “onClick”-style methods (event handlers) clutters the implementation with details not related to the actual user interface being constructed. FRP has in fact originated from this field, with Elliott and Hudak’s work on Fran ([8]).
2. **Robots** are the classic field of reactive programming. Robots need to continuously respond to environmental stimuli. For example, a simple robot that navigates a two-dimensional environment using range sensors. The range sensors provide changing input values, and the robot must supply the motor with a changing output value. A similar robot was treated by Hudak et al ([13]) using FRP. A more complicated robotic system has been the subject of at least one comparative study ([7]).
3. Parties in a **network** can be thought of as reactive systems sending and responding to messages. This idea is generalized by the Actor Model which is the basis for languages such as Erlang (in which the “actors” are concurrent). It is also similar to the idea of Object Oriented Programming, where the “objects” send messages to each other (this is embodied by method-calling).

3.2.2 Approaches to Reactive Programming

Reactive programming focuses on ever-changing inputs (discrete or continuous streams of data) and is particularly useful for robots and interactive systems. There are different systems and approaches to reactive programming, which can be summarized as a table showing which language provides or takes which approach. The programming languages and environments are compared on the following points:

- Programming Model - the general approach used by the language
- Discrete / Continuous - how time is handled by the language.
- Declarative / Imperative (programming style)
- Type System - Dynamic or Static, Strong (enforces type correctness) or Weak (does not always enforce), relative expressive power of the type system
- Semantic Model - if such a model exists for the language, what is it?

Table 3.1 compares several reactive languages on these points. In some cases the classification isn’t clear, and a question mark (?) appears.

3.2.3 Advantages and disadvantages of existing methods

All the **advantages** offered by the existing reactive programming languages are in the “short term”, or in terms of the implementation only. They are mainly:

Name	Reactive Model	Declarative / Imperative	Type System	Properties of semantic model
Erlang ([15])	Actors with discrete messages	Functional (impure)	Dynamic, Strong	Complicated, not well defined
Esterel ([7])	Synchronous (discrete time)	Imperative	Static, Strong	Well defined, composable
FRP using Haskell	Continuous and discrete time	Functional (pure)	Static, Strong, Powerful	Simple, well defined, composable
Lucid Synchrone ([22])	Synchronous (discrete time)	Functional (impure)	Static, Strong	?
Microsoft RDS ([23])	Discrete time?	Imperative	(Depends on language)	Complicated, undefined
Signal ([7])	Synchronous (discrete time)	Declarative	Static, Strong	?
Simulink ([24])	Continuous and discrete time	Imperative	? (Dynamic, Weak?)	Linear systems for some parts, complicated/undefined otherwise

Table 3.1: Comparison of languages for reactive programming.

- High performance (in the case of Signal, Esterel and possibly Erlang and Microsoft RDS).
- Large libraries tailored for specific tasks (especially Simulink, on a much lesser degree also Microsoft RDS).
- Portability (Erlang, Microsoft RDS, also Simulink using additional compilation tools).
- Easy to learn or using popular knowledge (Microsoft RDS due to its usage of .NET languages, Simulink is well known in the engineering community as a prototyping tool).
- Highly dependable framework/runtime implementations, under the assumption that your implementation is correct (Esterel, Erlang, Signal).

The main **disadvantage** suffered by all existing methods, and the main focus of the project is the *complicated semantic model* which leads to ultimately complicated designs. It is very hard to reason about the design because it uses abstractions which are themselves hard to detail precisely. Some of the existing tools (especially Simulink with its great support for numerical systems) *do* offer substantial features on this point, but are limited to specific types of problems, and do not offer a general solution. It is not possible to naturally and easily combine general, modular, high-level design with the reactively-programmed aspect of the system in any of the known languages. FRP aims to solve this problem by encouraging denotational design and supplying the generally required abstractions for reactive programming.

3.2.4 Details of some reactive languages

Simulink

Simulink (produced by The MathWorks, [24]) is an environment for building and running time-based numerical systems with block-diagram semantics. It supplies an impressive library of ‘blocks’ which perform a variety of operations including communications, controls, signal processing, video processing, and image processing. Simulink provides a GUI for constructing, editing and controlling the systems. It also allows compilation into C source code (which can then be used on various platforms). In addition to the built-in blocks, Simulink allows definition of new blocks that run MATLAB, C, C++, or Fortran code. The connections between blocks are “signals”, which carry numerical types (integers, floating point numbers, enumerations, vectors, etc.).

Lucid Sychrone

Lucid Sychrone ([22]) is an experimental, strongly typed, semi-functional (side effects allowed) language designed for reactive systems. It uses Objective Caml as a base language. Its features include:

- Streams are represented as values. For example, a stream of integers is represented as a value of type integer.
- Provisions for safety guarantees on generated code (for example, can detect non-causal function definitions).
- Supports product and sum data types and pattern matching.
- Limited support for higher-order streaming of functions, and functions of streams.

Lucid Sychrone is open source.

Erlang

Erlang ([20]) is an open source, relatively popular environment and language that uses message passing between processes. It was designed for the programming of telephony systems. Although Erlang was not designed with “reactiveness” as a main goal, its good support for parallelization and actor-based model can be useful for reactive systems. Each process runs a dynamically typed, semi-functional program to send messages and handle incoming ones. An Erlang process is not an operating system processes. Erlang implements its own processes which can migrate in a network and run on different machines, including virtual ones. For details about Erlang’s motivation and history, see [15].

Microsoft Robotics Studio

Robotics Developer Studio (or RDS, see [23]) is an environment produced by Microsoft aimed specifically at robot development. As described by the official website, it is:

“...a Windows-based environment for hobbyist, academic and commercial developers to create robotics applications for a variety of hardware platforms. RDS includes a lightweight REST-style, service-oriented runtime, a set of visual authoring and simulation tools, as well as tutorials and sample code to help get started.”

The RDS includes:

- A simulation tool for robotics environments, including physics simulation.
- Various tools for controlling and monitoring robots from the PC.
- Development in one of the following languages:
 - A visual programming tool, “Microsoft Visual Programming Language”, to ease development especially for non-programmers.
 - VB.Net
 - C#
 - C++
 - Python

Other reactive programming tools

- Signal, SyncCharts, Argos, Lustre, Esterel - mostly obsolete languages, all synchronous data-flow and event-based. Esterel is still used by European companies producing railroad controllers, but since commercialization its use is limited to niche markets. Some of these achieved very high performance due to compilation of the source code into finite state machines, which can be translated into fast software or even directly into hardware.

- The flow-based programming book (<http://www.jpaulmorrison.com/fbp/>) about data-flow-based programming, lists many more languages, not all of which are suited for reactive programming. The author also distributes a tool for visual data-flow programming.

3.3 FRP - Functional Reactive Programming

Functional reactive programming is a paradigm that combines a reactive approach to systems with functional programming (in Haskell). It has its roots in GUI and graphics programming (see [8]). FRP is based on the following two basic abstractions for designing systems:

- *Behaviors* (also called *Signals*) are time-varying values. They are semantically modelled as functions: $\text{Time} \rightarrow \mathbf{a}$ where \mathbf{a} is the value type.
- *Events* are discrete occurrences in time. In Elliott’s Reactive FRP ([19]) they are semantically modelled as sequences of time-value pairs: $[(\text{Time}, \mathbf{a})]$.

An important feature of FRP can already be observed - the semantic model is defined clearly and simply, right from the start. The model is based on the mathematically trivial notions of functions (for Behaviors) and sequences (for Events).

An alternative model for events (chosen by the developers of Yampa, see [11]) is a function $\text{Time} \rightarrow \text{Maybe Value Type}$, which somewhat resembles the notion of an ‘impulse train’ from engineering. However, by this model an event is really a continuous signal whose value may be absent at some points in time.

Currently, all FRP implementations are open source and are developed by individuals or academic research groups.

3.3.1 FRP Frameworks and Variants

Currently, there is no implementation or framework of FRP that can be recognized as “complete”. Most implementations haven’t been used enough to be deemed dependable, and there are *conceptual* differences in the way the different implementors decided to interpret FRP. Thus, the name FRP is used to describe a general approach rather than a specific set of abstractions or a specific implementation framework. There are, however, two notable implementations, and they mirror two approaches to FRP. They are:

1. Reactive - by Conal Elliott, as recently described in [19].
2. Yampa - from Yale, as described in [11] and [13].

There is still an ongoing (mini) argument in the FRP community about the pros and cons of each approach. The main advantage of Yampa is that its implementation has been tested and used, specifically for robotics. During the project’s development we shall decide which framework to use, based on trial and error. Initially, Yampa seems to be the best choice from a practical point of view.

Reactive

As mentioned, Reactive builds on the two abstractions of behaviors and events. Borrowing from the denotational semantic style presented in the introduction, we can say that:

- $[[\text{Behavior } \mathbf{a}]] = \text{Time} \rightarrow \mathbf{a}$
- $[[\text{Event } \mathbf{a}]] = [(\text{Time}, \mathbf{a})]$

The *meaning* of the type **Behavior** \mathbf{a} is a time-dependent value, a function from time to the value’s type. The *meaning* of **Event** \mathbf{a} is a sequence of pairs, each pair containing the occurrence time of the event, and a corresponding value. For convenience, following Elliott’s notation we define:

- $B_{\mathbf{a}} = \text{Time} \rightarrow \mathbf{a}$
- $E_{\mathbf{a}} = [(\text{Time}, \mathbf{a})]$

Note that $B_{\mathbf{a}}$ and $E_{\mathbf{a}}$ live in the semantic world. They are definitions of *meanings*.

Functor, Applicative and Monad

One of Reactive's great strengths is that it builds on existing abstractions. The most important ones are the highly general functor, applicative and monad. Monads are described in the research review in the section about functional programming and Haskell. Applicatives and functors are respectively more general than monad, and they too are based on concepts from the mathematics of Category Theory.

Functor

If f is a functor type, it provides the following interface:

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

And satisfies the rules:

```
fmap id = id
fmap (h . g) = (fmap h) . (fmap g)
```

Where the dot ($.$) is the function composition operator. `fmap` can be thought of as something that *lifts* a function ($a \rightarrow b$) into the functor's world, and applies it on an input functor `f a` to produce the result functor, `f b`.

For example, we may say that a sequence (a list) with values of type `a` (denoted `[a]`) is a functor. `fmap` for lists must then have type $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$ - it applies a function on a list of `a`'s and returns a list of `b`'s. The natural meaning for `fmap` in this context is that it applies the given function on each element of the first list separately, and returns the list of the results. Incidentally, this is exactly what the `map` function does. It's also easy to see that this `fmap` satisfies the rules defined above for functors.

The advantage of identifying a type as an instance of Functor (or Applicative or Monad or any other type class) is that we can use any generic 'functorish' library functions that are already implemented for our instance. It also helps when we define our own types (and specify what type classes they are instances of), because it encourages us to generalize our ideas. We find ourselves thinking: "What is this like? What does this resemble? What is the essence of this type?".

Specifically in the case of the Reactive framework, *behaviors* are identified as functor instances. The meaning of `fmap` on behaviors is *pointwise function application*. Coincidentally, because behaviors have the semantic meaning of a function (from time to some type), `fmap` is actually simply function composition. Similarly, Reactive's *events* are also functors, and `fmap` applies the given function pointwise on each event's value as it occurs.

Applicative

Applicative (or the "applicative functor") is one step more specific than functors. Applicative types have the added ability to *lift* values into the "container world", whereas functors could only apply pure functions on already-contained values. Another ability is to apply already "contained functions" (*lifted functions*) onto lifted values, whereas functors could only apply pure (unlifted) functions onto lifted values.

The interface for an Applicative type `f` is as follows:

```
pure  :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

The application operator `<*>` is defined as infix (and thus enclosed in parenthesis above). It is easy to see that `fmap` can be realised as follows:

```
fmap :: (a -> b) -> f a -> f b
fmap g fa = (pure g) <*> fa
```

This proves that Applicative is more specific than Functor.

Behaviors are applicative instances. The application operator `<*>` on behaviors is easy to interpret: it takes a behavior of functions, a behavior of values, and at each time t , applies the function (from the first behavior) on the value (from the second behavior).

Monad

Monads were already described, but we shall note how they are related to applicatives (and thus to functors). Intuitively, monad is a specialization of applicative that adds the ability to *choose* the “container” type. In applicatives (and functors) we were able to apply a function on a contained value, and receive a new contained value - but the container itself was out of our control. With monads, one can control which *container* will result from the function application. This capability is embodied in the *bind operation*, which is denoted `>>=`, and has the following type signature (m is the monad instance type):

```
(>>=) :: m a -> (a -> m b) -> m b
```

See the section about functional programming for more about monads. A review of the gradual specialization of the type classes mentioned here is given in [17].

Examples of Reactive functions

Reactive is defined in terms of semantic meanings, and the implementation’s choice of actual interfaces follows these meanings closely. To fully appreciate (and understand) the semantics of Reactive, one may read the appropriate paper ([19]). We can only touch the tip of the iceberg here. To give a feeling of how the library is designed, we shall note a few of its functions:

- **time**, the most trivial behavior. It has type `Behavior Time` and thus has meaning `Time → Time`. It is defined as the *identity function*, so that the value of `time` is always the current time. In engineering terms, the function is:

$$time(t) = t$$

- **merge** operates on events, and its semantic meaning has type $E_a \rightarrow E_a \rightarrow E_a$. This is a function that takes two events and *merges* them into a third. The merged event occurs whenever one of the two input events would occur, with the appropriate value. If both input events occur at the same time, the second one is dropped and the merged event corresponds to the first. **merge** corresponds roughly to the *addition* of two ‘impulse train’ functions, except the case of overlapping deltas occurring at the same time. So we have a rough equivalent of:

$$merge\{e1, e2\}(t) = \begin{cases} e1(t), & e1(t) \neq 0 \\ e2(t), & e1(t) = 0 \end{cases}$$

Note that this notation implicitly assumes a dirac’s delta notation for event occurrences - the merged result is really a discrete sequence of events.

- **switcher** is an operation that uses both events and behaviors to create a new behavior. It has semantic type $B_a \rightarrow E_{B_a} \rightarrow B_a$ - in other words, **switcher** takes an initial behavior and an event whose values are behaviors. Every time an event occurs, the **switcher** switches to the new behavior that is contained in the latest event. Thus, if time starts at t_0 and the first event occurs at time t_1 , second event at t_2 , etc., and if the behaviors contained in the event occurrences are b_1, b_2, \dots then the switcher’s behavior is:

$$switcher\{b_{initial}, e_{b_n}\}(t) = \begin{cases} b_{initial}(t), & t_0 \leq t < t_1 \\ b_1(t), & t_1 \leq t < t_2 \\ b_2(t), & t_2 \leq t < t_3 \\ \vdots & \end{cases}$$

- **stepper** is another operator, one that makes usage of **switcher**. It basically implements a “zero order hold” of events to produce a behavior (behaviors are always continuous). It has semantic type: $a \rightarrow E_a \rightarrow B_a$, taking an initial value and an event source and creating a behavior. If the values contained in the events are v_1, v_2, \dots then the **stepper** is:

$$\text{stepper}\{v_{\text{initial}}, e_{v_n}\}(t) = \begin{cases} v_{\text{initial}}, & t_0 \leq t < t_1 \\ v_1, & t_1 \leq t < t_2 \\ v_2, & t_2 \leq t < t_3 \\ \vdots & \end{cases}$$

Yampa

An alternative approach to Reactive’s is taken by Yampa. In Yampa, behaviors are called *signals* and are not first-class values - one cannot directly manipulate them. Instead, the reactive system is constructed by connecting *signal functions*. A signal function takes an input signal, performs a transformation on it and results in an output signal. Yampa is based on the *arrow* typeclass, which can be seen as a generalization (or specialization?) of the monad. Monads allow computations to be sequences linearly, whereas arrows allow arbitrary computation graphs, including loops and branches (parallel computations). A good introductory paper describes Yampa’s basic concepts in detail and shows how a simple computer game can be implemented using this approach - see [12].

Every SF (signal function) has an input signal and an output signal. A few of the arrow type class operators, which are used for composing SFs (signal functions) in Yampa, are:

- Sequencing - `>>>` - given two SFs A and B, creates a new SF that is equivalent to A followed by B.
- Parallelizing - `&&&` - given two SFs A and B, creates a new SF that accepts a pair signal and feeds the first element of the pair to A, and the second element to B. The output signal is a pair composed from the outputs of A and B.
- Lifting - `arr` - given a pure function **f**, creates an SF whose operation is the application of **f** point-wise on the values of the input signal to produce the output signal.
- Other operators include looping, and applying an SF on the first (or second) element in a paired signal.

To ease graph construction, the arrow library includes a special syntax that resembles the imperative-style `do`-notation that exists for monads. An example given in [12] is:

```
sf = proc (a,b) -> do
  c1 <- sf1 -< a
  c2 <- sf2 -< b
  c  <- sf3 -< (c1, c2)
  rec
    d <- sf4 -< (b,c,d)
  returnA -< (d,c)
```

This syntax can be described as follows:

Make a new signal function (and call it **sf**) that takes a pair of signals (**a**,**b**) as input, and performs the following. Feed **a** through **sf1** to receive **c1**, and likewise **b** through **sf2** to receive **c2**. Then, take the two outputs and use them as a pair of inputs for **sf3** to output **c**. Then, perform a recursion (loop) by feeding the triplet (**b**,**c**,**d**) as inputs to **sf4** where **d** is the output of that signal function. Finally, output a pair of signals (**d**,**c**) from our new signal function, which we called **sf**.

Graphically it can be drawn as in Figure 3.1. An alternative graphic representation is given in Figure 3.2.

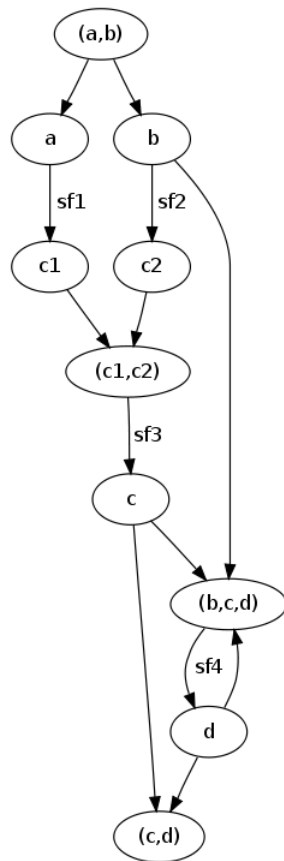


Figure 3.1: Signal function as a graph. This is the composite signal function defined by the example syntax. The nodes are signals (behaviors, values), and the edges are signal *functions*.

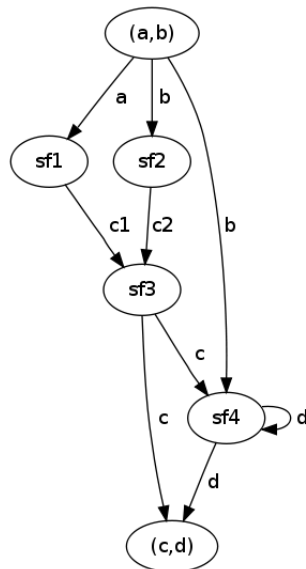


Figure 3.2: Alternative graphic model for composite signal function. The nodes are signal functions, and the edges are signals.

Implementations using Yampa

A few notable implementations that have used Yampa are:

- Frag - an impressive Quake 3 (computer game engine) clone. See <http://www.haskell.org/haskellwiki/Frag>.
- SimBots - a simulated robotics environment, used for a graduate summer course in Oxford University. See [13]. The simulated robots were not unlike the Segway RMP 200 that is used for this project. Unfortunately, the implementation has fallen out of date and does not compile with updated versions of ghc (the popular Haskell compiler), and also uses out-of-date specific versions of packages that have since become standardized.
- A “Space Invaders” game - see [12], the paper was already mentioned here as a good introduction to Yampa.

Explicit vs. Implicit Graphing

One of the differences between Reactive and Yampa lies in the explicitness (or implicitness) of the computation graph. In Reactive, the graph is implicit. The programmer specifies how behaviors and events are related to previous ones using the various reactive primitives offered by the framework. These primitives are functions, and the computation graph is built by composing (or applying) functions. Thus, a function that takes two arguments can be seen as a “node” that has two incoming edges. In Yampa, on the other hand, the computation graph is explicit. We build the reactive system by explicitly connecting “signal functions” using the Arrow operators.

RT-FRP

Another approach to FRP of special importance in the context of robotics, is RT-FRP (see [9] and the PhD dissertation [10]) and its two sub-variants, Hybrid FRP (H-FRP) and Event FRP (E-FRP). RT-FRP is a language for real-time reactive systems, whose special feature is a guarantee on resource bounds. H-FRP is specifically tailored for hybrid (continuous time *and* discrete events) systems, whereas E-FRP is for event-based systems. The original developers of RT-FRP did not deal with actual implementation, and several groups have attacked this problem. One example is that RT-FRP has been used for the implementation of a Lego Mindstorms robot, comparing the results to an equivalent implementation in C (see [14]). I have decided not to use an RT-FRP implementation for the project due to the currently limited amount of resources available on the subject.

3.3.2 Summary

FRP already offers a significant advantage in terms of semantic simplicity, generality and composability. The hope is that eventually *implementation* in FRP will be a practical alternative. This project aims to evaluate the actual advantages of FRP and to compare it to an existing approach for reactive programming. The evaluation will be performed by implementing a robot’s controlling program in both FRP and an alternative language, and comparing the results as described.

Chapter 4

Specification of the robotic system

4.1 Introduction

FRP's usefulness for robotics programming will be evaluated by the implementation a robotic controller. The robot will detect and follow objects (as captured by a video camera) using a simple image processing algorithm. This project focuses on the FRP paradigm, so the image processing / computer vision algorithm will be simple and unimportant. We may pick an 'off-the-shelf' algorithm for this purpose. The robot will be based on the Segway RMP 200. A USB camera that is already attached to an existing RMP unit will be used in conjunction with a laptop computer that will be attached to the robot. The laptop will connect to the RMP's wheel drive controller via USB, allowing a computer program to control the robot's velocity and orientation (the general layout described here can be seen in Figure 4.1). The laptop will run the linux operating system. The controlling program will be designed and implemented using functional reactive programming, using a Haskell-embedded FRP framework.

4.2 The controlling program

As we'll be using FRP, the program's design at the top level is clear and simple. The robot is modelled as a reactive system that transforms time-varying values and events. There are two inputs to the controlling program: video from the camera, and status events from the RMP. The video is modelled as a time-varying image. The following describes my top-level design for the controlling program:

1. The incoming video is processed by the *detector* whose output is a time-varying function with a `Maybe Rectangle` value. If an object was *not* detected, the output will be `Nothing`. Otherwise, the rectangle output specifies the coordinates of a box inside the image that bound the detected object.



Figure 4.1: General structure of the robot

2. The detected bounding box is passed to the *locator*, which maps bounding boxes to distance and orientation coordinates (r, θ) , a vector from the robot to the detected object's inferred location.
3. The location vector is passed to the *controller*, which drives the robot by sending appropriate events to the RMP. For example, if the target object is ahead, the controller may increase the velocity to get closer to the object.
4. The RMP sends events to the controller, with information about current wheel velocity, etc.

The top-level design appears in Figure 4.2.



Figure 4.2: Top-level design of the controlling program

4.3 The Segway RMP 200

4.3.1 Features

The Segway RMP (Robotic Mobility Platform) 200 is a platform for robotics development and testing. It is produced by Segway, Inc. (which publishes more information about the platform at <http://www.segway.com/business/products-solutions/robotic-mobility-platform.php>). It features the following:

- Two individually powered wheels
- Electrically powered (includes batteries)
- Two stabilization modes:
 - Dynamic - the RMP balances itself in an upright position
 - Static - requires additional support to remain upright
- USB and Canbus communication to the internal controller

4.3.2 Specifications

- Speed: 0 - 10 mph / 0 - 16 kph
- Weight: 140 lbs / 64 kg
- Range: 12 - 15 miles / 19 - 24 km
- Dimensions: 25" x 29.5" x 24" / 64 x 75 x 61 cm

4.3.3 Cost

The Segway RMP-200 starts at \$21,000 USD. Ben-Gurion University has already acquired several units, so purchase of an RMP was not necessary for this project.

Chapter 5

Evaluating FRP - testing the robotic system

The main focus of the project is not the “robotic intelligence” or image-processing aspects. Rather, it is the design methods and implementation clarity. Therefore the tests will focus on the comparison between the FRP and the alternative implementation.

No special equipment is required for the testing.

1. *Sanity tests.* Both implementations will be tested for sanity, to make sure they actually perform the basic functions that are expected. The sanity tests will include:
 - (a) Image processing operation - check that the robot process images as expected by the chosen algorithm.
 - (b) Wheel drive control - check that the robot commands the RMP properly and performs the desired movements.
 - (c) Integration - check that the robot detects and follows objects.
2. *Comparison tests.* As mentioned in the research proposal, the following aspects will be compared between the two implementations:
 - (a) Code size and conciseness.
 - (b) Modularity.
 - (c) Tractability (how easy is it to reason about the correctness of the code).
 - (d) Binary size.
3. *Performance tests.* Since this isn't a focus of the project, only simple performance tests will be carried out if time allows. The main performance measure is reaction time latency.

Chapter 6

Estimated Budget

Table 6.1 gives an estimated budget for the project (prices are in NIS). All the required equipment (computers, Segway RMP 200, video camera) already exists in the laboratory so the only price for equipment is the per-hour usage price (due to wear).

Name	Price (per hour)	Hours	Units	Total
Salary	20	200	1	4000
Computer Usage	4	200	1	800
Segway RMP 200	75	50	1	3750
USB Video Camera	0.15	50	1	7.5

Table 6.1: Cost estimate

The total budget estimated after rounding is 8600 NIS (approximately \$2300 USD).

Chapter 7

Work plan

The project will be composed of the following stages:

7.1 Stage 1 - Implementation using FRP

Since the FRP implementation is the primary focus of the project, and also the primary concern in terms of unknowns and risks, it will be the first stage.

1. **Test program** - Implementation of a simple robotic program using Yampa FRP, with the goal of assuring that Yampa is an appropriate framework for the project (no serious bugs or compilation problems, etc.).
2. **Detailed design** of the robot's controlling program, using FRP abstractions. This design will be detailed enough to specify the robot's entire operation, including the exact algorithms, rules for safety measures and other details not covered in the top-level design.
3. **Detailed test specification** for the robot's design. The idea of writing the test spec at this stage is to help evoke ideas about possible flaws and problems in the design.
4. **I/O modules** - Implementation of the input/output modules for communication between the controlling program and the RMP (the robot platform), and for receiving video input from the USB video camera. The I/O modules must interface between Haskell (Yampa's host language) and the USB devices.
5. **Implementation** of the design using Yampa. This may become the 'heaviest' part of the project, due to uncertainties in Yampa's stability and performance, which may require multiple iterations of tweaking the implementation.
6. **Testing** the robot, according to the test spec.

7.2 Stage 2 - Implementation using another reactive language

At this stage we will select one of the other reactive programming languages and environments (Simulink, Microsoft Robotics Studio, Lucid Synchronic, or Erlang). The selection will be based on our experience implementing the robot in FRP.

Exactly as above, this stage will include design, implementation and testing.

7.3 Stage 3 - Comparison

At this stage we will compare the FRP design and implementation with the other one. The following points will be evaluated:

- Design tractability, clearness, and ease of analysis:
 - Use of generalized abstractions vs. specifically designed mechanisms.
 - Ability to reason about the design's correctness and limitations.
 - Simplicity, in the sense of making it easy to explain the design to an uninvolved person with basic engineering skills.
- Implementation complexity:
 - Code size (lines of code).
 - Modularity and generality - percentage of the code that is reusable in a different, similar program.
 - Simplicity (in the same sense as for the design, above).
- Correctness: percentage of tests passed.
- Performance is not the focus of this project, but will be measured and noted, if time allows:
 - Latency in reaction to external stimuli.
 - Memory and CPU usage.

7.4 Stage 4 - Conclusion

At this stage we shall finish writing the required reports, and prepare the presentation and poster for the project.

Extensions: if time allows, we will improve the robot or add features not included in the original design.

Chapter 8

Timetable

The planned timetable is given as a Gantt chart in Figure 8.1 and as a table in Table 8.2.

Task	Name	Start	Finish	Work
1	Preliminary report corrections	Nov 15	Dec 2	13d
2	Preliminary Submission Date	Nov 22	Nov 22	
3	Preliminary-Adviser Correction Date	Nov 29	Nov 29	
4	Preliminary-Students Correction Date	Dec 6	Dec 6	
5	Preliminary Evaluation Date	Nov 15	Nov 15	
6	FRP	Dec 3	Feb 18	55d
6.1	Test program	Dec 3	Dec 9	5d
6.2	Detailed Design	Dec 10	Dec 23	10d
6.3	Test Specification	Dec 24	Dec 30	5d
6.4	I/O Modules	Dec 31	Jan 6	5d
6.5	Implementation	Jan 10	Feb 4	20d
6.6	Testing	Feb 7	Feb 18	10d
7	Exams	Jan 17	Feb 4	15d
8	Progress Submission Date	Mar 14	Mar 14	
9	Progress-Adviser Correction Date	Mar 21	Mar 21	
10	Progress-Students Correction Date	Apr 6	Apr 6	
11	Progress Evaluation Date	Apr 11	Apr 11	
12	Alternative Implementation	Feb 21	Apr 22	45d
12.1	Test program	Feb 21	Feb 25	5d
12.2	Detailed Design	Feb 28	Mar 4	5d
12.3	I/O Modules	Mar 7	Mar 18	10d
12.4	Implementation	Mar 21	Apr 8	15d
12.5	Testing	Apr 11	Apr 22	10d
13	Pesach	Mar 29	Apr 7	8d
14	Comparison	Apr 25	Apr 29	5d
15	Poster Date	May 2	May 1	
16	Presentation Date	May 2	May 1	
17	Corrections and bugfixes	May 2	Jun 10	30d
18	Final Submission Date	Aug 1	Aug 1	
19	Final-Adviser Correction Date	Aug 8	Aug 8	
20	Final-Students Correction Date	Aug 15	Aug 15	
21	Final Evaluate Date	Aug 22	Aug 22	

Table 8.2: Timetable

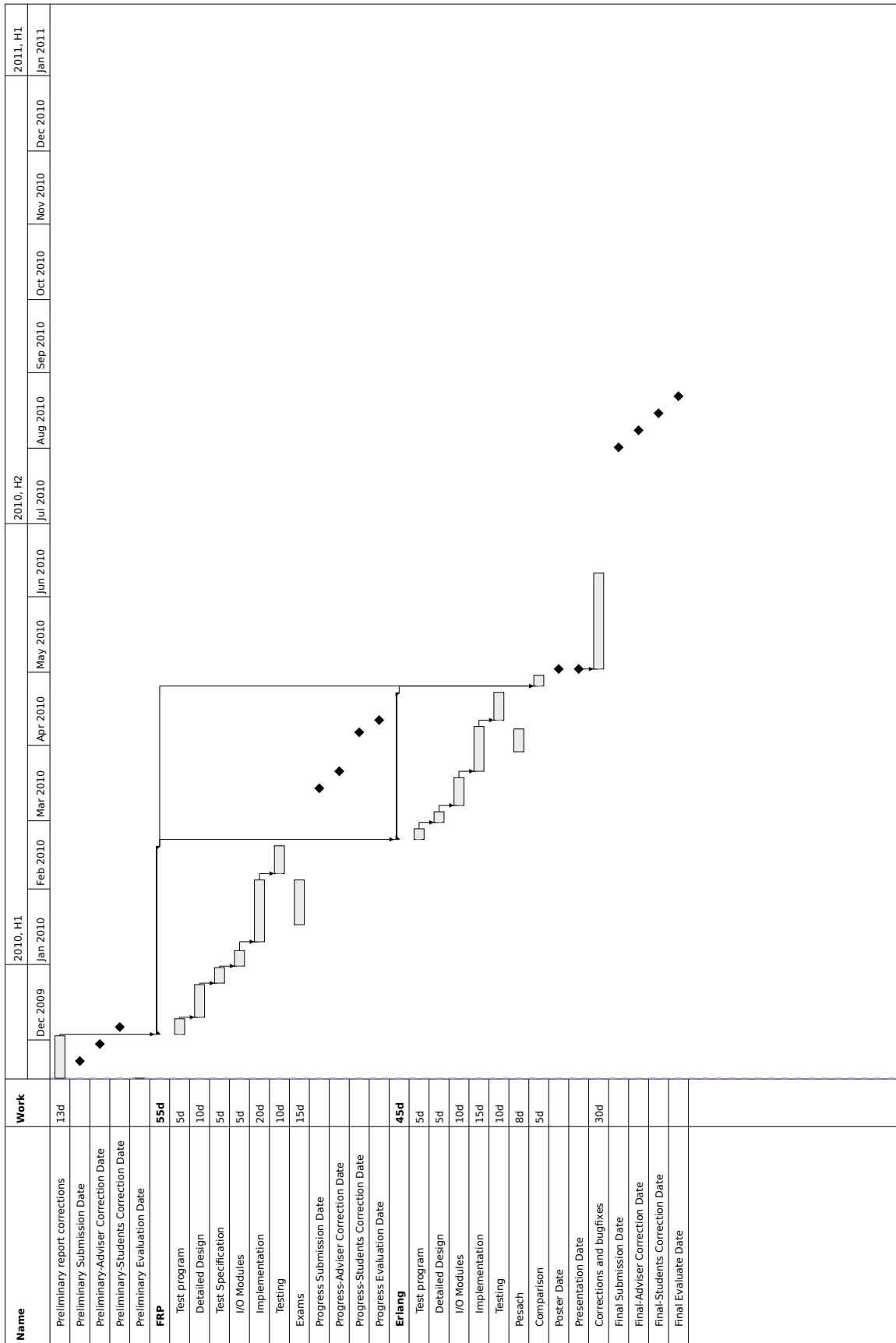


Figure 8.1: Project time plan as Gantt chart

Appendix A

Lambda Calculus at a glance

Lambda calculus is the basis of functional programming. It deals with the definition and evaluation of functional expressions. We take a complex expression and successively *reduce* (transform) it to reach a final canonical form. For the sake of brevity we shall give an incomplete, slightly inaccurate presentation of lambda calculus. The subject itself is beyond the scope of this project; a good self-sustained introduction to lambda calculus, and a nice build-up into functional programming is given in [4].

A.1 Expressions

An expression in lambda calculus is one of the following:

- A variable name, such as x .
- A function definition, which is called a *lambda abstraction*. The syntax is: $\lambda x.E$ where λx specifies that x is the name of the function argument, and E is the function body (which can be a compound expression).
- A function application, which is composed of two expressions. It is written $E1 E2$, where $E1$ is a function expression (a lambda abstraction).
- We can also define “built-in” expressions (such as a function *add* or a constant 4).

A variable is *bound* in an expression if it appears as a function argument, and the expression is inside the function definition. For example x is bound in $\lambda x.x$. Otherwise, a variable is *free*.

Functions can only take a single argument. When more are needed, we can nest functions: $\lambda x.\lambda y.E$ takes x , and returns a function that takes y . This is known as *currying* (after the logician Haskell Curry). A common abbreviation is to write $\lambda x y.E$ in place of the nested expression.

A.2 Reduction rules

Lambda calculus also defines rules for transforming expressions (reduction rules):

- **α -conversion:** Changing the name of a bound variable. For example, $\lambda x.x \rightarrow \lambda y.y$.
- **Substitution:** If a variable x appears free in the expression E we can perform a substitution, replacing x with another expression. We denote it like this: $E1[E2/x]$. You can think of it as ‘dividing’ $E1$ by x and ‘multiplying’ by $E2$.
- **β -reduction:** Function application. This rule transforms $(\lambda x.E1) E2$ into $E1[E2/x]$, using substitution.
- **η -conversion:** Reduces ‘redundant’ functions. An expression $\lambda x. f x$ can be η -converted to f , because the two expressions are equivalent.

Note that recursion is not allowed, which seems to strictly limit the expressiveness of the language. Fortunately, there turns out to be a function that we can define *inside* lambda calculus (the ‘Y-combinator’) that essentially allows recursion.

The aforementioned ‘canonical form’ is a unique, final minimal expression (not further reducible) that is guaranteed to exist by the Church-Rosser theorem. There is a specific reduction order that must be taken in order to reach the canonical form.

A.3 Examples of Lambda Expressions

Here are a few examples of expressions:

- $\lambda x.x$ is the identity function. For example, the function application $(\lambda x.x) E$ evaluates to E for any E .
- The natural numbers can be encoded in lambda calculus without built-in constants, as *Church numerals* (invented by the creator of lambda calculus, Alonzo Church). Some definitions:
 - $zero = \lambda f x. x$
 - $succ = \lambda n f x. f (n f x)$

Applying *succ zero* gives the following transformations:

$$\begin{aligned}
 succ\ zero &= (\lambda n f x. f (n f x))\ zero \\
 &= \{\beta - reduction : substitute\ zero\ in\ place\ of\ n\} \\
 &= (\lambda f x. f (zero f x)) \\
 &= (\lambda f x. f ((\lambda f x. x) f x)) \\
 &= \{\beta - reduction\} \\
 &= (\lambda f x. f x)
 \end{aligned}$$

Thus, we have:

- $1 = succ\ zero = \lambda f x. fx$
- $2 = succ\ 1 = \lambda f x. f(fx)$
- $3 = succ\ 2 = \lambda f x. f(f(fx))$
- etc.

Church took the next step to also define functions for addition, multiplication, subtraction, exponentiation, etc.

Bibliography

- [1] Scott D. and Strachey C., “Toward A Mathematical Semantics for Computer Languages”, Proceedings of the Symposium on Computers and Automata, J. Fox, ed., Brooklyn, N.Y.: Polytechnic Press, 1971, pp. 19-46.
- [2] Backus J., “Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs”, Communications of the ACM, vol. 21, 1978, pp. 613-641.
- [3] Hughes J., “Why Functional Programming Matters”, Computer Journal, vol. 32, 1989, pp. 98-107. <http://www.cs.chalmers.se/~rjmh/Papers/whyfp.html>.
- [4] Peyton-Jones S.L., “The Implementation of Functional Programming Languages”, Prentice-Hall International Series in Computer Series, Prentice Hall, 1987. <http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/>
- [5] Dijkstra, E. W., “Real mathematicians don’t prove”, Monograph number 1012, January 24 1988.
- [6] Wadler P., “The essence of functional programming”, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, New Mexico, United States: ACM, 1992, pp. 1-14. <http://homepages.inf.ed.ac.uk/wadler/topics/monads.html>
- [7] Lewerentz C. and Lindner T., eds., “Formal Development of Reactive Systems - Case Study Production Cell”, Lecture Notes in Computer Science vol. 891, Springer-Verlag, 1995.
- [8] Elliott C. and Hudak P., “Functional Reactive Animation”, International Conference on Functional Programming, 1997, pp. 163-173. <http://www.conal.net/fran/>
- [9] Wan Zh., Taha W., Hudak P., “Real-time FRP”, International Conference on Functional Programming, 2001.
- [10] Wan Zh., “Functional Reactive Programming for Real-Time Reactive Systems”, PhD. thesis, 2002, Yale University.
- [11] Nilsson H., Courtney A., and Peterson J., “Functional reactive programming, continued”. In Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, pages 51-64. ACM New York, NY, USA, 2002.
- [12] Courtney A., Nilsson H., and Peterson J., “The Yampa Arcade”, Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell’03), Uppsala, Sweden: ACM Press, 2003, pp. 7-18. <http://haskell.org/yale/papers/haskell-workshop03/>
- [13] Hudak P., Courtney A., Nilsson H., and Peterson J., “Arrows, Robots, and Functional Reactive Programming”, Summer School on Advanced Functional Programming 2002, Oxford University, Springer-Verlag, 2003, pp. 159-187. <http://haskell.org/yale/papers/oxford02/>
- [14] Lee B., Lee D., Woo G., “Functional Reactive Program Translator for Controlling Robot Systems”, 2007 International Conference on Convergence Information Technology, IEEE, 2007, pp. 1322-1325.
- [15] Armstrong J., “A history of Erlang”, Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III, San Diego, California: 2007, pp. 6-1-6-26. <http://doi.acm.org/10.1145/1238844.1238850>. More information on Erlang can be found at <http://www.erlang.org>

- [16] Hudak P., Hughes J., Jones S.P., and Wadler P., “A history of Haskell: being lazy with class”, Proceedings of the third ACM SIGPLAN conference on History of programming languages, San Diego, California: ACM, 2007, pp. 12-1-12-55.
- [17] Yorgey B., “The Typeclassopedia”, in The Monad.Reader Issue 13, March 12 2009, pp. 17-68. <http://themonadreader.wordpress.com/previous-issues/>
- [18] Elliott C., “Denotational design with type class morphisms” (extended version), LambdaPix, 2009. <http://conal.net/papers/type-class-morphisms/>
- [19] Elliott C., “Push-pull functional reactive programming”, in Haskell Symposium, ACM. <http://conal.net/papers/push-pull-frp/>
- [20] Erlang, <http://www.erlang.org/>
- [22] Lucid Sychrone, <http://www.lri.fr/~pouzet/lucid-sychrone/>
- [23] Robotics Development Studio, Microsoft, <http://msdn.microsoft.com/en-us/robotics/>
- [24] Simulink, The Mathworks, <http://www.mathworks.com/products/simulink/>
- [25] ISO/IEC 9899:TC3 (C language), Committee Draft - September 7, 2007, pp. 35, section 6.2.5-20