University of Brasília

Computer Science Department

Continuous Time Modeling Made Functional: Solving Differential Equations with Haskell

Eduardo L. Rocha

Monograph submitted in partial fullfilment of
the requirements to the Computer Engineering Program at University of Brasília

Advisor
Prof. José Edil Guimarães

Brasília
2022

University of Brasília

Computer Science Department

Continuous Time Modeling Made Functional: Solving Differential Equations with Haskell

Eduardo L. Rocha

Monograph submitted in partial fullfilment of
the requirements to the Computer Engineering Program at University of Brasília

Prof. José Edil Guimarães (Advisor)
ENE/UnB

Prof. Vander Ramos Alves
CIC/UnB

Dr. George Ungureanu
Ericsson Sweden

Prof. João José Costa Gondim
Coordinator of Computer Engineering Program at University of Brasília

Brasília, May 11, 2022

"If you don't know, the thing to do is not to get scared, but to learn."

— Ayn Rand, *Atlas Shrugged*

Dedicated to

I dedicate this milestone to my family. To my sister Alexya Lemos, for the memorable moments of joy and fun. To my father Rodolfo Rocha, for sharing with me his wise and insightful perceptions about life. And foremost, to my mom Dania Lemos, for her relentless will of teaching me the basics of everything since infancy, for her always-present joyful smiles to alleviate my painful academic journey, for being my golden guardian during my countless unhealthy decisions in pursuit of doing the best regardless of the cost, for always celebrating my minor and major achievements to make me believe in self-steem by merit and for being an inspirational example of an unstoppable warrior for me to follow.

Acknowledgements

First and foremost, I thank my advisor José Edil Guimarães. He trusted that my effort could go on and beyond, surpassing my own expectations and limits, even without having any experience with the chosen domain of the project. All the endless meetings, including on the weekends, filled with thoughtful advice and helpful comments, will be the most remarkable memory of the best teacher I have encountered to this day.

I'm thankful for my colleague Breno Fatureto for inviting me to this beautiful and elegant world of functional programming, which is the core foundation of the present work, even though this being an uncommon paradigm to explore during graduation.

I'm grateful for the company I'm currently working in, Datarisk, where I'm surrounded by phenomenal people who always remind me of the importance of pushing forward, diminishing the fear that naturally comes when entering new countries.

Finally, a special thanks for everybody that took any amount of time to read any draft I had of this thesis, providing honest feedback to enhance my final project before being graduated as a computer engineer.

Abstract

Physical phenomena is difficult to properly model due to its continuous nature. Its paralellism and nuances were a challenge before the transistor, and even after the digital computer still is an unsolved issue. In the past, some formalism were brought with the General Purpose Analog Computer proposed by Shannon in the 1940s. Unfortunately, this formal foundation was lost in time, with *ad-hoc* practices becoming mainstream to simulate continuous time. In this work, we propose a domain-specific language (DSL) written in Haskell that resembles GPAC's concepts. The main goal is to take advantage of high level abtractions to execute systems of differential equations, which describe physical problems mathematically. We evaluate performance and domain problems and address them accordingly. Future improvements for the DSL are also explored and detailed.

Keywords: differential equations, continuous systems, GPAC, integrator

Resumo

Fenômenos físicos são difíceis de modelar propriamente devido a sua natureza contínua. O paralelismo e nuances envolvidos eram um desafio antes do transistor, e mesmo depois do computador digital esse problema continua insolúvel. No passado, algum formalismo foi trazido pelo computador analógico de propósito geral (GPAC) por Shannon nos anos 1940. Infelizmente, essa base formal foi perdida com o tempo, e práticas *ad-hoc* tornaram-se comuns para simular o tempo contínuo. Neste trabalho, propomos uma linguagem de domínio específico (DSL) escrita em Haskell que assemelha-se aos conceitos do GPAC. O principal objetivo é aproveitar de abstrações de mais alto nível para executar sistemas de equações diferenciais, que descrevem sistemas físicos matematicamente. Nós avaliamos performance and problemas de domínio e os endereçamos propriamente. Melhorias futuras para a DSL também são exploradas e datalhadas.

Palavras-chave: equações diferenciais, sistemas contínuos, GPAC, integrador

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Continuous behaviours are deeply embedded into the real world. However, even our most advanced computers are not capable of completely modeling such phenomena due to its discrete nature; thus becoming a still-unsolved challenge. Cyber-physical systems (CPS) — the integration of computers and physical processes [3, 4] — tackles this problem by attempting to include into the *semantics* of computing the physical notion of *time* [4], i.e., treating time as a measurement of *correctness*, not *performance* [3] nor just an accident of implementation [4]. Additionally, many systems perform in parallel, which requires precise and sensitive management of time; a non-achievable goal by using traditional computing abstractions, e.g., *threads* [4].

Examples of these concepts are older than the digital computers; analog computers were used to model battleships' fire systems and core functionalities of fly-by-wire aircraft [5]. The mechanical metrics involved in these problems change continuously, such as space, speed and area, e.g., the firing's range and velocity are crucial in fire systems, and surfaces of control are indispensable to model aircraft's flaps. The main goal of such models was, and still is, to abstract away the continuous facet of the scenario to the computer. In this manner, the human in the loop aspect only matters when interfacing with the computer, with all the heavy-lifting being done by formalized use of shafts and gears in analog machines [6], and by **software** after the digital era.

Within software, the aforementioned issues — the lack of time semantics and the wrong tools for implementing concurrency — are only a glimpse of serious concerns orbiting around CPS. The main villain is that today's computer science and engineering primarily focus on matching software demands, not expressing essential aspects of physical systems [4, 7]. Further, its sidekick is the weak formalism surrounding the semantics of model-based design tools; modeling languages whose semantics are defined by the tools

rather than by the language itself [7], encouraging ad-hoc design practices. With this in mind, Lee advocated that leveraging better formal abstractions is the paramount goal to advance continuous time modeling [4, 7]. More importantly, these new ideas need to embrace the physical world, taking into account predictability, reliability and interoperability.

The development of a *model of computation* (MoC) to define and express models is the major hero towards this better set of abstractions, given that it provides clear, formal and well-defined semantics [3]. These MoCs determine how concurrency works in the model, choose which communication protocols will be used, define whether different components share the notion of time, as well as whether and how they share state [3, 7]. Also, Sangiovanni and Lee [8] proposed a formalized denotational framework to allow understanding and comparison between mixtures of MoCs, thus solving the heterogeneity issue that raises naturally in many situations during design [3, 7]. Moreover, their framework also describes how to compose different MoCs, along with addressing the absence of time in models, via what is defined as *tagged systems* — a relationship between a *tag*, generally used to order events, and an output value.

Ingo et al. went even further [9] by presenting an example of a framework based on the idea of tagged systems, known as *ForSyDe*. The tool's main goal is to push system design to a higher level of abstraction, by combining MoCs with the functional programming paradigm. The technique separates the design into two phases, specification and synthesis. The former stage, specification, focus on creating a high-level abstraction model, in which mathematical formalism is taken into account. The latter part, synthesis, is responsible for applying design transformations — the model is adapted to ForSyDe's semantics — and mapping this result onto a chosen architecture for later be implemented in a target programming language or hardware platform [9]. Afterward, Seyed-Hosein and Ingo [10] created a co-simulation architecture for multiple models based on ForSyDe's methodology, addressing heterogeneity across languages and tools with different semantics. One example of such tools treated in the reference is Simulink [1], the de facto model-based design tool that lacks a formal semantics basis [10]. Simulink being the standard tool for modeling means that, despite all the effort into utilizing a formal approach to model-based design, this is still an open problem.

1.2 Proposal

The aforementioned work — the formal notion of MoCs, the ForSyDe framework and its interaction with modeling-related tools like Simulink — comprises the domain of model-

[1] Simulink documentation.

based design or **model-based engineering**. Furthermore, the main goal of the present work contribute to this area of CPS by creating a domain-specific language tool (DSL) for simulating continuous-time systems that addresses the absence of a formal basis. Thus, this tool will help to cope with the incompatibility of the mentioned sets of abstractions [4] — the discreteness of digital computers with the continuous nature of physical phenomena.

The proposed DSL has two special properties of interest: it needs to be a set of well-defined *operational* semantics, thus being **executable**, and it needs to be related to a *formalized* reasoning process. The former aspect provides **verification via simulation**, a type of verification that is useful when dealing with **non-preserving** semantic transformations, i.e., modifications and tweaks in the model that do not assure that properties are being preserved. Such phenomena are common within the engineering domain, given that a lot of refinement goes into the modeling process in which previous proof-proved properties are not guaranteed to be maintained after iterations with the model. A work-around solution for this problem would be to prove again that the features are in fact present in the new model; an impractical activity when models start to scale in size and complexity. Thus, by using an executable tool as a virtual workbench, models that suffered from those transformations could be extensively tested and verified.

In order to address the latter property, a solid and formal foundation, the tool is inspired by the general-purpose analog computer (GPAC) formal guidelines, proposed by Shannon [6] in 1941. This concept was developed to model a Differential Analyzer — an analog computer composed by a set of interconnected gears and shafts intended to solve numerical problems [11]. The mechanical parts represents *physical quantities* and their interaction results in solving differential equations, a common activity in engineering, physics and other branches of science [6]. The model was based on a set of black boxes, so-called *circuits* or *analog units*, and a set of proved theorems that guarantees that the composition of these units are the minimum necessary to model the system, given some conditions. For instance, if a system is composed by a set of *differentially algebraic* equations with prescribed initial conditions [5], then a GPAC circuit can be built to model it. Later on, some extensions of the original GPAC were developed, going from solving unaddressed problems contained in the original scope of the model [5] all the way to make GPAC capable of expressing generable functions, Turing universality and hypertranscendental functions [11, 12]. Furthermore, although the analog computer has been forgotten in favor of its digital counterpart [5], recent studies in the development of hybrid systems [1] brought GPAC back to the spotlight in the CPS domain.

With these two core properties in mind, the proposed DSL will translate GPAC's original set of black boxes to some executable software.

1.3 Goal

The main goal of the present work is to build an executable software that can solve differential equations and resembles the core idea of the GPAC model. The programming language of choice was **Haskell**, due to a variety of different reasons. First, this is already being used in the CPS domain in some degree, as showed by the ForSyDe framework [9, 10]. Second, Lee describes a lot of properties [3] that matches the functional programming paradigm almost perfectly:

- Prevent misconnected MoCs by using great interfaces in between ⇒ Such interfaces can be built using Haskell's **strong type system**
- Enable composition of MoCs ⇒ Composition is a first-class feature in functional programming languages
- It should be possible to conjoin a functional model with an implementation model ⇒ Functions programming languages makes a clear the separation between the *denotational* aspect of the program, i.e., its meaning, from the *operational* functionality
- All too often the semantics emerge accidentally from the software implementation rather than being built-in from the start ⇒ A denotative approach with no regard for implementation details is common in the functional paradigm
- The challenge is to define MoCs that are sufficiently expressive and have strong formal properties that enable systematic validation of designs and correct-by-construction synthesis of implementations ⇒ Functional languages are commonly used for formal mathematical applications, such as proof of theorems and properties, as well as also being known for "correct-by-construction" approaches

Thus, we believe that the use of functional programming for modeling continuous time is not a coincidence; properties that are established as fundamental to leverage better abstractions for CPS simulation seem to be within the functional programming paradigm. Furthermore, this implementation is based on `Aivika` [2] — an open source multi-method library for simulating a variety of paradigms, including partial support for physical dynamics, written in Haskell. Our version is modified for our needs, such as demonstrating similarities between the implementation and GPAC, shrinking some functionality in favor of focusing on continuous time modeling, and re-thinking the overall organization of the project for better understanding. So, this reduced and refactored version of `Aivika`, so-called `Rivika` [3], will be a Haskell Embedded Domain-Specific Language (HEDSL) within

[2] `Aivika` source code.
[3] `Rivika` source code.

the model-based engineering domain. So, the built DSL will explore Haskell's specific features and details, such as the type system and typeclasses, to solve differential equations. Figure 1.1 shows a side-by-side comparison between a physical system and a valid model created in `Rivika`.

```
sigma = 10.0
rho = 28.0
beta = 8.0 / 3.0

lorenzModel :: Model Vector
lorenzModel =
  do integX <- newInteg 1.0
     integY <- newInteg 1.0
     integZ <- newInteg 1.0
     let x = readInteg integX
         y = readInteg integY
         z = readInteg integZ
     diffInteg integX (sigma * (y - x))
     diffInteg integY (x * (rho - z) - y)
     diffInteg integZ (x * y - beta * z)
     return $ sequence [x, y, z]
```

$$\sigma = 10.0$$

$$\rho = 28.0$$

$$\beta = \frac{8.0}{3.0}$$

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z)$$

$$\frac{dz}{dt} = xy - \beta z$$

Figure 1.1: The translation between the world of software and the mathematical description of differential equations are explicit in `Rivika`.

1.4 Outline

Although written in Haskell, a high level programming language, `Rivika` explores a mix of advanced abstractions with some direct memory manipulation — usually associated with low level programming languages. Hence, the proposed software will be explained in multiple chapters, each one with a separate and concrete objective.

Chapter 2, *Design Philosophy*, will explain basic Haskell concepts, such as the type system and different sorts of polymorphism, and it will bind them to numerical methods and GPAC's circuits. The next chapter, *Effectful Integrals*, is dedicated to introduce GPAC's integrator representative in software, alongside further improvements in the overall modeling of physical systems. The follow-up chapter, *Execution Walkthrough*, will discuss how the proposed types aligns with mathematical definitions introduced in Chapter 2. Moreover, how to execute a simulation as well as a guided example are presented. At the end, some issues will be identified with the implementation at that point. Chapters 5 and 6, *Travelling across Domains* and *Caching the Speed Pill* respectively, address these concerns. Finally, limitations, future improvements and final thoughts are drawn in chapter 7, *Conclusion*.

Chapter 2

Design Philosophy

In the previous chapter, the importance of making a bridge between two different sets of abstractions — computers and the physical domain — was established. This chapter will explain the core philosophy behind the implementation of this link, starting with an introduction to GPAC, followed by the type system used in Haskell, as well as understanding how to model the main entities of the problem. At the end, the presented modeling strategy will justify the data types used in the solution, paving the way for the next chapter *Effectful Integrals*.

2.1 Shannon's Foundation: GPAC

The General Purpose Computer or GPAC is a model for the Differential Analyzer — a mechanical machine controlled by a human operator [12]. This machine is composed by a set of shafts interconnected in such a manner that a given differential equation is expressed by a shaft and other mechanical units transmit their values across the entire machine [6, 11]. For instance, shafts that represent independent variables directly interact with shafts that depicts dependent variables. The machine is primarily composed by four types of units: gear boxes, adders, integrators and input tables [6]. These units provide useful operations to the machine, such as multiplication, addition, integration and saving the computed values. The main goal of this machine is to solve ordinary differential equations via numerical solutions.

In order to add a formal basis to the machine, Shannon built the GPAC model, a mathematical model sustained by proofs and axioms [6]. The end result was a set of rules for which types of equations can be modeled as well as which units are the minimum necessary for modeling them and how they can be combined. All algebraic functions (e.g. quotients of polynomials and irrational algebraic functions) and algebraic-trascendental functions (e.g. exponentials, logarithms, trigonometric, Bessel, elliptic and probability

functions) can be modeled using a GPAC circuit [1, 6]. Moreover, the four preceding mechanical units were renamed and together created the minimum set of **circuits** for a given GPAC [1]. Figure 2.1 portrays these basic units, followed by descriptions of their behaviour, inputs and outputs.

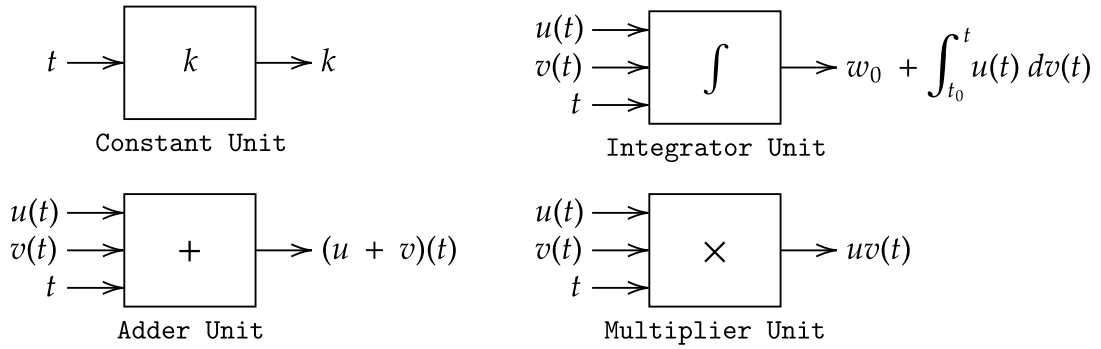


Figure 2.1: The combination of these four basic units compose any GPAC circuit (taken from [1] with permission).

- Constant Function: This unit generates a real constant output for any time t.
- Adder: It generates the sum of two given inputs with both varying in time, i.e., it produces $w = u + v$ for all variations of u and v.
- Multiplier: The product of two given inputs is generated for all moments in time, i.e., $w = uv$ is the output.
- Integrator: Given two inputs — $u(t)$ and $v(t)$ — and an initial condition w_0 at time t_0, the unit generates the output $w(t) = w_0 + \int_{t_0}^{t} u(t)\, dv(t)$, where u is the *integrand* and v is the *variable of integration*.

Composition rules that restrict how these units can be hooked to one another. Shannon established that a valid GPAC is the one in which two inputs and two outputs are not interconnected and the inputs are only driven by either the independent variable t (usually *time*) or by a single unit output [1, 5, 6]. Daniel's GPAC extension, FF-GPAC [5], added new constraints related to no-feedback GPAC configurations while still using the same four basic units. These structures, so-called *polynomial circuits* [1, 11], are being displayed in Figure 2.2 and they are made by only using constant function units, adders and multipliers. Also, such circuits are *combinational*, meaning that they compute values in a *point-wise* manner between the given inputs. Thus, FF-GPAC's composition rules are the following:

- An input of a polynomial circuit should be the input t or the output of an integrator. Feedback can only be done from the output of integrators to inputs of polynomial circuits.

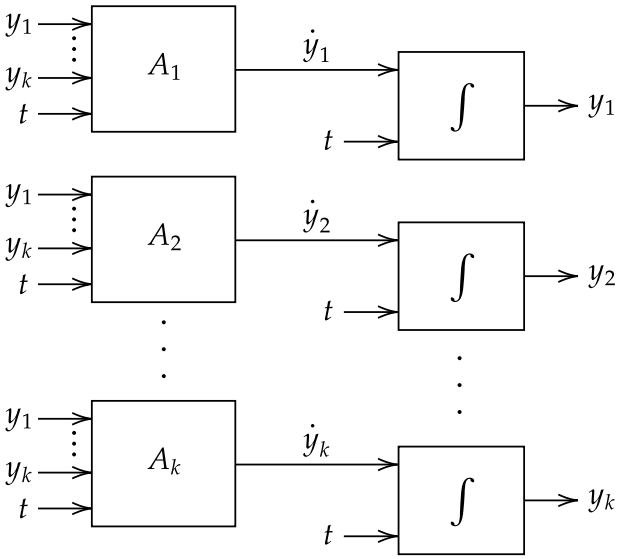


Figure 2.2: Polynomial circuits resembles combinational circuits, in which the circuit respond instantly to changes on its inputs (taken from [1] with permission).

- Each polynomial circuit admit multiple inputs.
- Each integrand input of an integrator should be generated by the output of a polynomial unit.
- Each variable of integration of an integrator is the input t.

During the definition of the DSL, parallels will map the aforementioned basic units and composition rules to the implementation. With this strategy, all the mathematical formalism leveraged for analog computers will drive the implementation in the digital computer. Although we do not formally prove a refinment between the GPAC theory, i.e., our especification, and the final implementation, `Rivika` is an attempt to build a tool with formalism taken into account; one of the most frequent critiques in the CPS domain, as explained in the previous chapter.

2.2 The Shape of Information

Types in programming languages represent the format of information. Figure 2.3 illustrates types with an imaginary representation of their shape and Figure 2.4 shows how types can be used to restrain which data can be plumbered into and from a function. In the latter image, function *lessThan10* has the type signature `Int -> Bool`, meaning that it accepts `Int` data as input and produces `Bool` data as the output. These types are used

to make constratins and add a safety layer in compile time, given that using data with different types as input, e.g, `Char` or `Double`, is regarded as a **type error**.

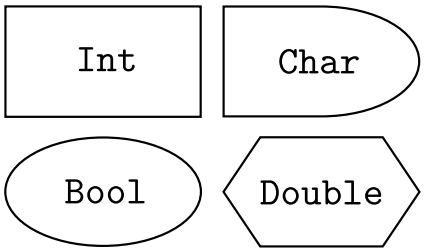


Figure 2.3: Types are not just labels; they enhance the manipulated data with new information. Their difference in shape can work as the interface for the data.

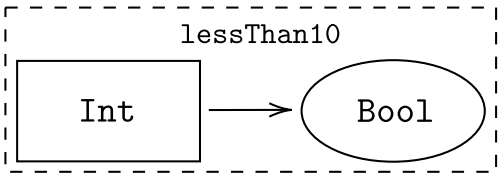


Figure 2.4: Functions' signatures are contracts; they specify which shape the input information has as well as which shape the output information will have.

Primitive types, e.g., `Int`, `Double` and `Char`, can be **composed** to create more powerful data types, capable of modeling complicated data structures. In this context, composition means binding or gluing existent types together to create more sophisticated abstractions, such as recursive structures and records of information. Two **algebraic data types** are the type composition mechanism provided by Haskell to bind existent types together.

The sum type, also known as tagged union in type theory, is an algebraic data type that introduces **choice** across multiple options using a single label. For instance, a type named `Parity` can represent the parity of a natural number. It has two options or representatives: `Even` **or** `Odd`, where these are mutually exclusive. When using this type either of them will be of type `Parity`. A given sum type can have any number of representatives, but only one of them can be used at a given moment. Figure 2.5 depicts examples of sum types with their syntax in the language, in which a given entry of the type can only assume one of the available possibilities. Another use case depicted in the image is the type `DigitalStates`, which describes the possible states in digital circuits as one of three options: `High`, `Low` and `Z`.

```haskell
data Parity = Even | Odd

data DigitalStates = High | Low | Z
```

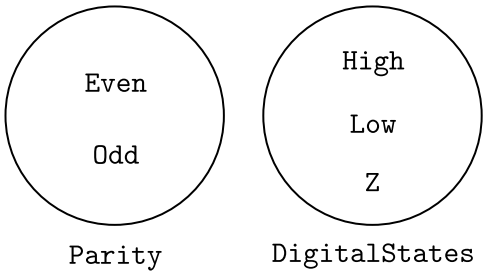


Figure 2.5: Sum types can be understood in terms of sets, in which the members of the set are available candidates for the outer shell type. Parity and possible values in digital states are examples.

The second type composition mechanism available is the product type, which **combines** using a type constructor. While the sum type adds choice in the language, this data type requires multiple types to assemble a new one in a mutually inclusive manner. For example, a digital clock composed by two numbers, hours and minutes, can be portrayed by the type `ClockTime`, which is a combination of two separate numbers combined by the wrapper `Time`. In order to have any possible time, it is necessary to provide **both** parts. Effectively, the product type executes a cartesian product with its parts. Figure 2.6 illustrates the syntax used in Haskell to create product types as well as another example of combined data, the type `SpacePosition`. It represents spatial position in three dimensional space, combining spatial coordinates in a single place.

```haskell
data ClockTime = Time Int Int

data SpacePosition = Point Double Double Double

data SpacePosition = Point { x :: Double,
                             y :: Double,
                             z :: Double }
```

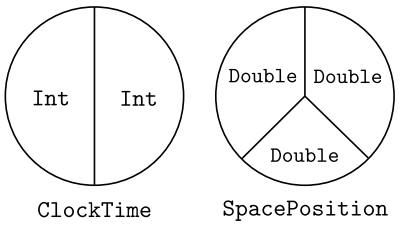


Figure 2.6: Product types are a combination of different sets, where you pick a representative from each one. Digital clocks' time and objects' coordinates in space are common use cases. In Haskell, a product type can be defined using a **record** alongside with the constructor, where the labels for each member inside it are explicit.

Within algebraic data types, it is possible to abstract the **structure** out, meaning that the outer shell of the type can be understood as a common pattern changing only the internal content. For instance, if a given application can take advantage of integer values but want to use the same configuration as the one presented in the `SpacePosition` data type, it's possible to add this customization. This feature is known as *parametric polymorphism*, a powerful tool available in Haskell's type system. An example is presented in Figure 2.7 using the `SpacePosition` type structure, where its internal types are being parametrized, thus allowing the use of other types internally, such as `Float`, `Int` and `Double`.

In some situations, changing the type of the structure is not the desired property of interest. There are applications where some sort of **behaviour** is a necessity, e.g., the ability of comparing two instances of a custom type. This nature of polymorphism is known as *ad hoc polymorphism*, which is implemented in Haskell via what is similar to java-like interfaces, so-called **typeclasses**. However, establishing a contract with a typeclass differs from an interface in a fundamental aspect: rather than inheritance being given to the type, it has a lawful implementation, meaning that **mathematical formalism** is assured for it. As an example, the implementation of the typeclass `Eq` gives to the type all comparable operations (== and ! =). Figure 2.8 shows the implementation of `Ord`

```haskell
data SpacePosition a = Point a a a

data SpacePosition a = Point { x :: a,
                               y :: a,
                               z :: a }
```

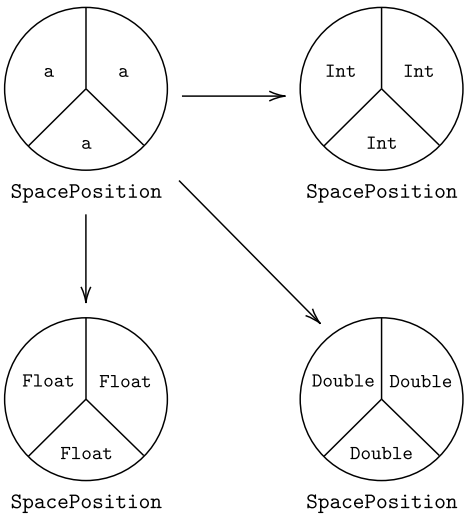


Figure 2.7: Depending on the application, different representations of the same structure need to used due to the domain of interest and/or memory constraints.

typeclass for the presented `ClockTime`, giving it capabilities for sorting instances of such type.

```haskell
data ClockTime = Time Int Int

instance Ord ClockTime where
  (Time a b) <= (Time c d)
    = (a <= c) && (b <= d)
```

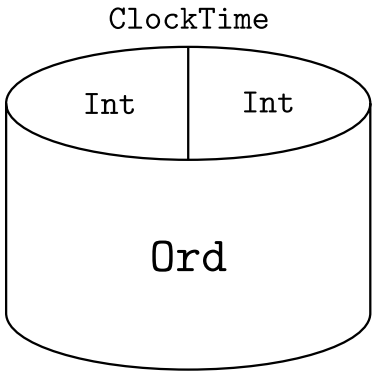


Figure 2.8: The minimum requirement for the `Ord` typeclass is the <= operator, meaning that the functions <, <=, >, >=, `max` and `min` are now unlocked for the type `ClockTime` after the implementation. Typeclasses can be viewed as a third dimension in a type.

Algebraic data types, when combined with polymorphism, are a powerful tool in programming, being a useful way to model the domain of interest. However, both sum and product types cannot portray by themselves the intuition of a **procedure**. A data transformation process, as showed in Figure 2.4, can be utilized in a variety of different ways. Imagine, for instance, a system where validation can vary according to the current situation. Any validation algorithm would be using the same data, such as a record called `SystemData`, and returning a boolean as the result of the validation, but the internals of these functions would be totally different. This is represented in Figure 2.9. In Haskell,

this motivates the use of functions as **first class citizens**, meaning that they are values and can be treated equally in comparison with data types that carries information, such as being used as arguments to another functions, so-called high order functions.

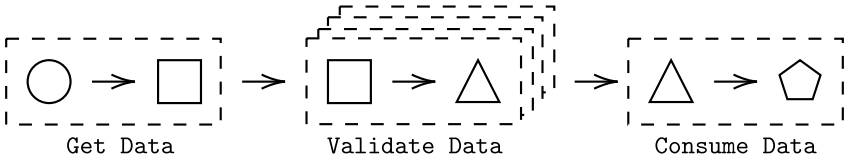


Figure 2.9: Replacements for the validation function within a pipeline like the above is common.

2.3 Modeling Reality

The continuous time problem explained in the introduction was initially addressed by mathematics, which represents physical quantities by **differential equations**. This set of equations establishes a relationship between functions and their respective derivatives; the function express the variable of interest and its derivative describe how it changes over time. It is common in the engineering and in the physics domain to know the rate of change of a given variable, but the function itself is still unknown. These variables describe the state of the system, e.g, velocity, water flow, electrical current, etc. When those variables are allowed to vary continuously — in arbitrarily small increments — differential equations arise as the standard tool to describe them.

While some differential equations have more than one independent variable per function, being classified as a **partial differential equation**, some phenomena can be modeled with only one independent variable per function in a given set, being described as a set of **ordinary differential equations**. However, because the majority of such equations does not have an analytical solution, i.e., cannot be described as a combination of other analytical formulas, numerical procedures are used to solve the system. These mechanisms **quantize** the physical time duration into an interval of numbers, each spaced by a **time step** from the other, and the sequence starts from an **initial value**. Afterward, the derivative is used to calculate the slope or the direction in which the tangent of the function is moving in time in order to predict the value of the next step, i.e., determine which point better represents the function in the next time step. The order of the method varies its precision during the prediction of the steps, e.g, the Runge-Kutta method of 4th order is more precise than the Euler method or the Runge-Kutta of 2nd order.

These numerical methods are used to solve problems specified by the following mathematical relations:

$$\dot{y}(t) = f(t, y(t)) \quad y(t_0) = y_0 \tag{2.1}$$

As showed, both the derivative and the function — the mathematical formulation of the system — varies according to **time**. Both acts as functions in which for a given time value, it produces a numerical outcome. Moreover, this equality assumes that the next step following the derivative's direction will not be that different from the actual value of the function y if the time step is small enough. Further, it is assumed that in case of a small enough time step, the difference between time samples is h, i.e., the time step. In order to model this mathematical relationship between the functions and its respective derivative, these methods use iteration-based approximations. For intance, the following equation represents one step of the first-order Euler method, the simplest numerical method:

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2.2}$$

So, the next step or iteration of the function y_{n+1} can be computed by the sum of the previous step y_n with the predicted value obtained by the derivative $f(t_n, y_n)$ multiplied by the time step h. Figure 2.10 provides an example of a step-by-step solution of one differential equation using the Euler method. In this case, the unknown function is a modified exponential function, and the time of interest is $t = 5$.

$$\dot{y} = y + t \qquad y(0) = 1$$

$$\downarrow$$

$$y_{n+1} = y_n + hf(t_n, y_n) \quad h = 1 \quad t_{n+1} = t_n + h \quad f(t, y) = y + t$$

$$y_1 = y_0 + 1 * f(0, y_0) \rightarrow y_1 = 1 + 1 * (1 + 0) \rightarrow y_1 = 2$$

$$y_2 = y_1 + 1 * f(1, y_1) \rightarrow y_2 = 2 + 1 * (2 + 1) \rightarrow y_2 = 5$$

$$y_3 = y_2 + 1 * f(2, y_2) \rightarrow y_3 = 5 + 1 * (5 + 2) \rightarrow y_3 = 12$$

$$y_4 = y_3 + 1 * f(3, y_3) \rightarrow y_4 = 12 + 1 * (12 + 3) \rightarrow y_4 = 27$$

$$y_5 = y_4 + 1 * f(4, y_4) \rightarrow y_5 = 27 + 1 * (27 + 4) \rightarrow y_5 = 58$$

Figure 2.10: The initial value is used as a starting point for the procedure. The algorithm continues until the time of interest is reached in the unknown function. Due to its large time step, the final answer is really far-off from the expected result.

2.4 Making Mathematics Cyber

Our primary goal is to combine the knowledge levered in section 2.2 — modeling capabilities of Haskell's algebraic type system — with the core notion of differential equations presented in section 2.3. The type system will model equation 2.2, detailed in the previous section.

Any representation of a physical system that can be modeled by a set of differential equations has an outcome value at any given moment in time. The type `Dynamics` in Figure 2.11 is a first draft of representing the continuous physical dynamics [3] — the evolution of a system state in time:

```haskell
type Time = Double
type Outcome = Double
data Dynamics =
     Dynamics (Time -> Outcome)
```

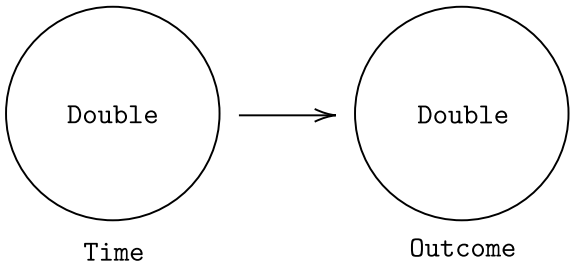


Figure 2.11: In Haskell, the `type` keyword works for alias. The first draft of the `Dynamics` type is a **function**, in which providing a floating point value as time returns another value as outcome.

This type seems to capture the concept, whilst being compatible with the definition of a tagged system presented by Lee and Sangiovanni [8]. However, because numerical methods assume that the time variable is **discrete**, i.e., it is in the form of **iterations** that they solve differential equations. Thus, some tweaks to this type are needed, such as the number of the current iteration, which method is being used, in which stage the method is and when the final time of the simulation will be reached. With this in mind, new types are introduced. Figure 2.12 shows the auxiliary types to build a new version of the `Dynamics` type.

The above auxiliary types serve a common purpose: to provide at any given moment in time, all the information to execute a solver method until the end of the simulation. The type `Interval` determines when the simulation should start and when it should end. The `Method` sum type is used inside the `Solver` type to set solver sensible information, such as the size of the time step, which method will be used and in which stage the method is in at the current moment. Finally, the `Parameters` type combines everything together, alongside with the current time value as well as its discrete counterpart, iteration.

Further, the new `Dynamics` type can also be parametrically polymorphic, removing the limitation of only using `Double` values as the outcome. Figure 2.13 depicts the final type

```haskell
data Interval = Interval { startTime :: Double,
                           stopTime  :: Double
                         } deriving (Eq, Ord, Show)

data Method = Euler
            | RungeKutta2
            | RungeKutta4
            deriving (Eq, Ord, Show)

data Solver = Solver { dt        :: Double,
                       method    :: Method,
                       stage     :: Int
                     } deriving (Eq, Ord, Show)

data Parameters = Parameters { interval  :: Interval,
                               solver    :: Solver,
                               time      :: Double,
                               iteration :: Int
                             } deriving (Eq, Show)
```

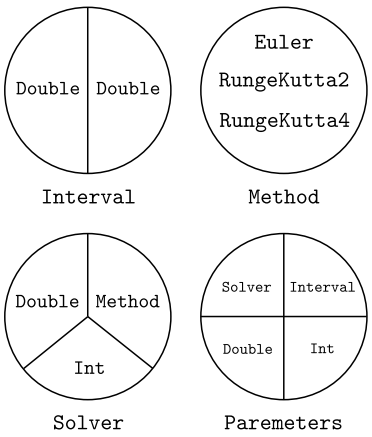


Figure 2.12: The `Parameters` type represents a given moment in time, carrying over all the necessary information to execute a solver step until the time limit is reached. Some useful typeclasses are being derived to these types, given that Haskell is capable of inferring the implementation of typeclasses in simple cases.

for the physical dynamics. The `IO` wrapper is needed to cope with memory management and side effects, all of which will be explained in the next chapter.

```haskell
data Dynamics a =
     Dynamics (Parameters -> IO a)
```

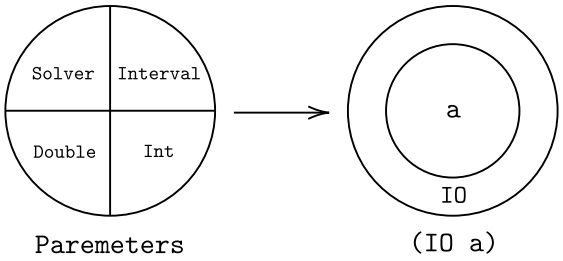


Figure 2.13: The `Dynamics` type is a function of from time related information to an arbitraty outcome value.

This summarizes the main pilars in the design: FF-GPAC, the mathematical definition of the problem and how we are modeling this domain in Haskell. The next chapter, *Effectful Integrals*, will start from this foundation, by adding typeclasses to the `Dynamics` type, and will later describe the last core type before explaining the solver execution: the `Integrator` type. These improvements for the `Dynamics` type and the new `Integrator` type will later be mapped to their FF-GPAC counterparts, explaining that they resemble the basic units mentioned in section 2.1.

Chapter 3

Effectful Integrals

This chapter details the next steps to simulate continuous-time behaviours. It starts by enhancing the previously defined `Dynamics` type by implementing some specific typeclasses. Next, the second core type of the simulation, the `Integrator` type, will be introduced alongside its functions. These improvements will then be compared to FF-GPAC's basic units, our source of formalism within the project. At the end of the chapter, an implicit recursion will be blended with a lot of effectful operations, making the `Integrator` type hard to digest. This will be addressed by a guided Lorenz Attractor example in the next chapter, *Execution Walkthrough*.

3.1 Uplifting the Dynamics Type

The `Dynamics` type needs **algebraic operations** to be better manipulated, i.e., useful operations that can be applied to the type preserving its external structure. These procedures are algebraic laws or properties that enhance the capabilities of the proposed function type wrapped by a `Dynamics` shell. Towards this goal, a few typeclasses need to be implemented.

Across the spectrum of available typeclasses in Haskell, we are interested in the ones that allow data manipulation with a single or multiple `Dynamics` and provide mathematical operations. To address the former group of operations, the typeclasses `Functor`, `Applicative`, `Monad` and `MonadIO` will be implemented. The later group of properties is dedicated to provide mathematical operations, such as $+$ and $\times$, and it can be acquired by implementing the typeclasses `Num`, `Fractional`, and `Floating`.

The typeclasses `Functor`, `Applicative` and `Monad` are all **lifting** operations, meaning that they allow functions to be lifted or involved by the chosen type. While they differ **which** functions will be lifted, i.e., each one of them lift a function with a different type signature, they share the intuition that these functions will be interacting with the

`Dynamics` type. This perspective is crucial for a practical understanding of these patterns. A function with a certain **shape** and details will be lifted using one of those typeclasses and their respective operators.

The `Functor` typeclass, when implemented for the type of interest, let the lifting of functions to be enclosed by the `Dynamics` type. Thus, as depicted in Figure 3.1, the function `a -> b` that comes as a parameter has its values surrounded by the same values wrapped with the `Dynamics` type, i.e., the outcome is a function with the signature `Dynamics a -> Dynamics b`. The code below shows the implementation of the *fmap* function — the minimum requirement to the `Functor` typeclass — to the `Dynamics` type. It is worth noting that, because this type uses an `IO` inside, a second *fmap*, this time related to `IO`, needs to be used in the implementation.

```
1 instance Functor Dynamics where
2   fmap f (Dynamics da) = Dynamics $ \ps -> fmap f (da ps)
```

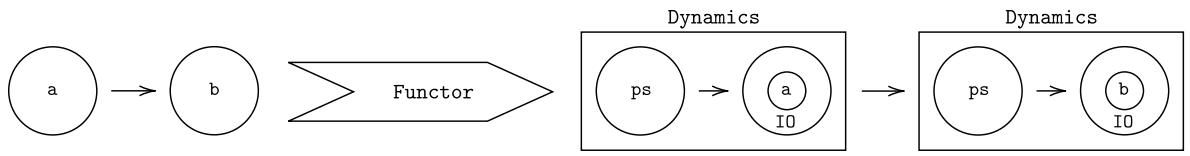
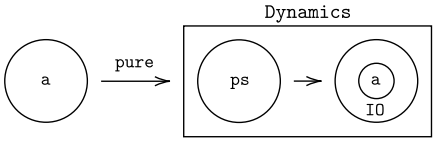


Figure 3.1: Given a parametric record `ps` and a dynamic value `da`, the *fmap* functor of the `Dynamics` type applies the former to the latter. Because the final result is wrapped inside the `IO` shell, a second *fmap* is necessary.

The next typeclass, `Applicative`, deals with functions that are inside the `Dynamics` type. When implemented, this algebraic operation lifts this internal function, wrapped by the type of choice, applying the **external** type to its **internal** members, thus generating again a function with the signature `Dynamics a -> Dynamics b`. The minimum requirements for this typeclass is the function *pure*, a function responsible for wrapping any value with the `Dynamics` wrapper, and the `<*>` operator, which does the aforementioned interaction between the internal values with the outer shell. The implementation of this typeclass is presented in the code bellow, in which the dependency `df` has the signature `Dynamics (a -> b)` and its internal function `a -> b` is being lifted to the `Dynamics` type. Figure 3.2 illustrates the described lifting with `Applicative`.

The third and final lifting is the `Monad` typeclass. In this case, the function being lifted **generates** structure as the outcome, although its dependency is a pure value. As Figure 3.3 portrays, a function with the signature `a -> Dynamics b` can be lifted to the signature `Dynamics a -> Dynamics b` by using the `Monad` typeclass. This new operation for lifting, so-called *bind*, is written below, alongside the *return* function, which is the same *pure* function from the `Applicative` typeclass. Together, these two functions

```haskell
instance Applicative Dynamics where
  pure a = Dynamics $ const (return a)
  (<*>) = appComposition
```




```haskell
appComposition :: Dynamics (a -> b) -> Dynamics a -> Dynamics b
appComposition (Dynamics df) (Dynamics da)
  = Dynamics $ \ps -> df ps >>= \f -> fmap f (da ps)
```

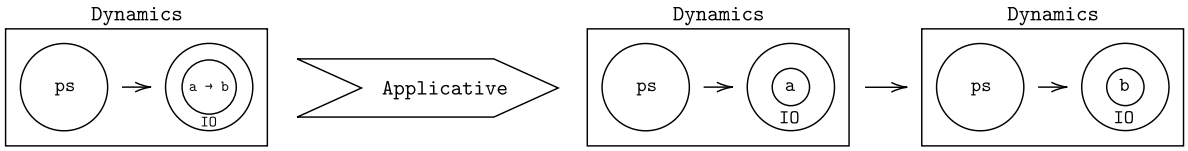
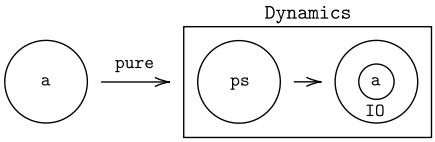


Figure 3.2: With the `Applicative` typeclass, it is possible to cope with functions inside the `Dynamics` type. Again, the *fmap* from `IO` is being used in the implementation.

represent the minimum requirements of the `Monad` typeclass. Figure 3.3 illustrates the aforementioned scenario.

```haskell
instance Monad Dynamics where
  return a = pure a
  m >>= k = bind k m
```




```haskell
bind :: (a -> Dynamics b ) -> Dynamics a -> Dynamics b
bind k (Dynamics m)
  = Dynamics $ \ps -> m ps >>= \a -> (\(Dynamics m') -> m' ps) $ k a
```

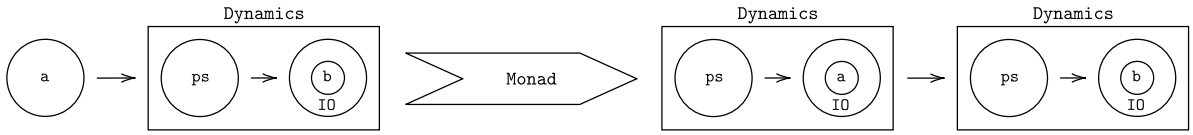


Figure 3.3: The >>= operator used in the implementation is the *bind* from the `IO` shell. This indicates that when dealing with monads within monads, it is frequent to use the implementation of the internal members.

Aside from lifting operations, the final typeclass related to data manipulation is the `MonadIO` typeclass. It comprises only one function, *liftIO*, and its purpose is to change the structure that is wrapping the value, going from an `IO` outer shell to the monad of interest, `Dynamics` in this case. The usefulness of this typeclass will be more clear in the next topic, section 3.3. The implementation is bellow, alongside its visual representation in Figure 3.4.

```haskell
instance MonadIO Dynamics where
  liftIO m = Dynamics $ const m
```

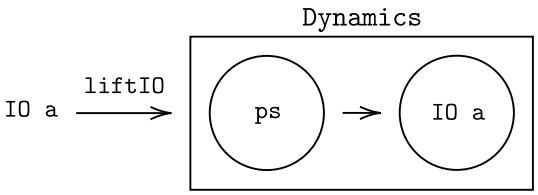


Figure 3.4: The typeclass `MonadIO` transforms a given value wrapped in `IO` into a different monad. In this case, the parameter `m` of the function is the output of the `Dynamics` type.

Finally, there are the typeclasses related to mathematical operations. The typeclasses `Num`, `Fractional` and `Floating` provide unary and binary numerical operations, such as arithmetic operations and trigonometric functions. However, because we want to use them with the `Dynamics` type, their implementation involve lifting. Further, the `Functor` and `Applicative` typeclasses allow us to execute this lifting, since they are designed for this purpose. The code bellow depicts the implementation for unary and binary operations, which are used in the requirements for those typeclasses:

```haskell
unaryOP :: (a -> b) -> Dynamics a -> Dynamics b
unaryOP = fmap

binaryOP :: (a -> b -> c) -> Dynamics a -> Dynamics b -> Dynamics c
binaryOP func da db = (fmap func da) <*> db
```

3.2 GPAC Bind I: Dynamics

After these improvements in the `Dynamics` type, it is possible to map some of them to FF-GPAC's concepts. As we will see shortly, the implemented numerical typeclasses, when combined with the lifting typeclasses (`Functor`, `Applicative`, `Monad`), express three out of four FF-GPAC's basic circuits presented in Figure 2.1 in the previous chapter.

First and foremost, all FF-GPAC units receive *time* as an available input to compute. The `Dynamics` type represents continuous physical dynamics [3], which means that it portrays a function from time to physical output. Hence, it already has time embedded into its definition; a record with type `Parameters` is received as a dependency to obtain the final result at that moment. Furthermore, it remains to model the FF-GPAC's black boxes and the composition rules that bind them together.

The simplest unit of all, `Constant Unit`, can be achieved via the implementation of the `Applicative` and `Num` typeclasses. First, this unit needs to receive the time of simulation at that point, which is an granted by the `Dynamics` type. Next, it needs to return a constant value k for all moments in time. The `Num` given the `Dynamics` type the option of using number representations, such as the types `Int`, `Integer`, `Float` and

`Double`. Further, the `Applicative` typeclass can lift those number-related functions to the desired type by using the *pure* function.

Arithmetic basic units, such as the `Adder Unit` and the `Multiplier Unit`, are being modeled by the `Functor`, `Applicative` and `Num` typeclasses. Those two units use binary operations with physical signals. As demonstrated in the previous section, the combination of numerical and lifting typeclasses let us to model such operations. Figure 3.5 shows FF-GPAC's analog circuits alongside their `Rivika` counterparts. The forth unit and the composition rules will be mapped after describing the second main type of `Rivika`: the `Integrator` type.

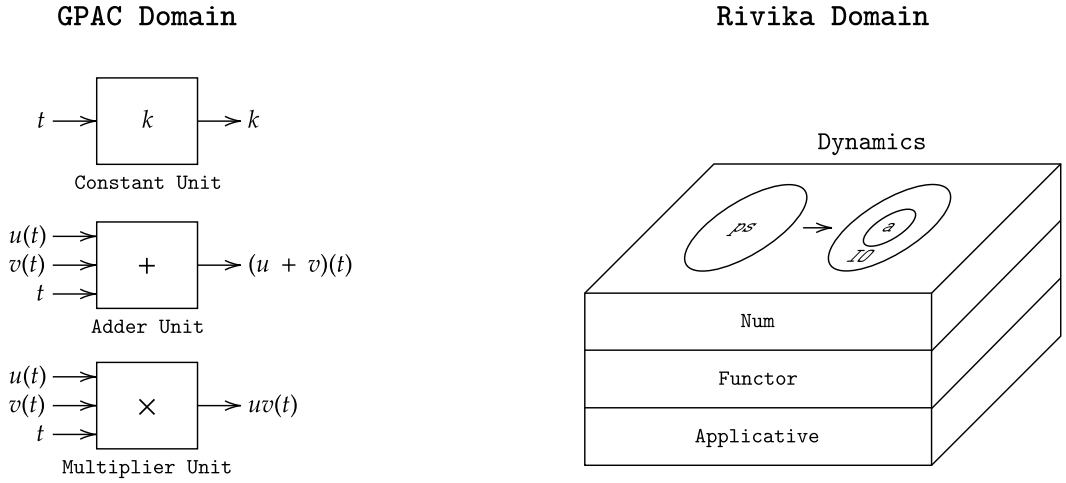


Figure 3.5: The ability of lifting numerical values to the `Dynamics` type resembles three FF-GPAC analog circuits: `Constant`, `Adder` and `Multiplier`.

3.3 Exploiting Impurity

The `Dynamics` type directly interacts with a second type that intensively explores **side effects**. The notion of a side effect correlates to changing a **state**, i.e., if you see a computer program as a state machine, an operation that goes beyond returning a value — it has an observable interference somewhere else — is called a side effect operation or an **impure** functionality. Examples of common use cases goes from modifying memory regions to performing input-output procedures via system-calls. The nature of purity comes from the mathematical domain, in which a function is a procedure that is deterministic, meaning that the output value is always the same if the same input is provided — a false assumption when programming with side effects. An example of an imaginary state machine can be viewed in Figure 3.6.

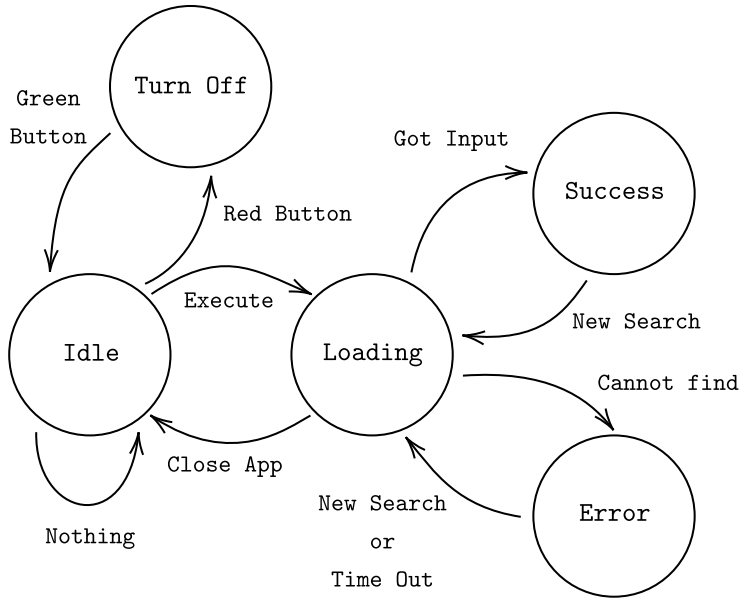


Figure 3.6: State Machines are a common abstraction in computer science due to its easy mapping between function calls and states. Memory regions and peripherals are embedded with the idea of a state, not only pure functions. Further, side effects can even act as the trigger to move from one state to another, meaning that executing a simple function can do more than return a value. Its internal guts can significantly modify the state machine.

In low-level and imperative languages, such as C and Fortran, impurity is present across the program and can be easily and naturally added via **pointers** — addresses to memory regions where values, or even other pointers, can be stored. In contrast, functional programming languages advocate to a more explicit use of such aspect, given that it prioritizes pure and mathematical functions instead of allowing the developer to mix these two facets. So, the developer has to take extra effort to add an effectful function into the program, clearly separating these two different styles of programming.

The second core type of the present work, the `Integrator`, is based on this idea of side effect operations, manipulating data directly in memory, always consulting and modifying data in the impure world. Foremost, it represents a differential equation, as explained in chapter 2, *Design Philosophy* section 2.3, meaning that the `Integrator` type models the calculation of an **integral**. It accomplishes this task by driving the numerical algorithms of a given solver method, implying that this is where the *operational* semantics of our DSL reside.

With this in mind, the `Integrator` type is responsible for executing a given solver method to calculate a given integral. This type comprises the initial value of the system, i.e., the value of a given function at time t_0, and a pointer to a memory region for future use, called `computation`. In Haskell, something similar to a pointer and memory allocation can be made by using the `IORef` type. This memory region is being allocated to be used with the type `Dynamics Double`. Also, the initial value is also represented by `Dynamics Double`, and the initial condition can be lifted to this type because the typeclass `Num` is implemented (section 3.1). It is worth noticing that these pointers are pointing to functions or **computations** and not to double precision values.

```haskell
data Integrator = Integrator { initial     :: Dynamics Double,
                               computation :: IORef (Dynamics Double)
                             }
```

There are three functions that involve the `Integrator` and the `Dynamics` types together: the function *newInteg*, responsible for allocating the memory that the pointer will pointer to, *readInteg*, letting us to read from the pointer, and *diffInteg*, a function that alters the content of the region being pointed. In summary, these functions allow us to create, read and update data from that region, if we have the pointer on-hand. All functions related to the integrator use what's known as `do-notation`, a syntax sugar of the `Monad` typeclass for the bind operator. The code bellow is the implementation of the *newInteg* function, which creates an integrator:

```haskell
newInteg :: Dynamics Double -> Dynamics Integrator
newInteg i =
  do comp <- liftIO $ newIORef $ initialize i
     let integ = Integrator { initial     = i,
                              computation = comp }
     return integ
```

The first step to create an integrator is to manage the initial value, which is a function with the type `Parameters -> IO Double` wrapped in `Dynamics`. After acquiring a given initial value `i`, the integrator needs to assure that any given parameter record is the beginning of the computation process, i.e., it starts from t_0. The `initialize` function fulfills this role, doing a reset in `time`, `iteration` and `stage` in a given parameter record. This is necessary because all the implemented solvers presumes **sequential steps**, starting from the initial condition. So, in order to not allow this error-prone behaviour, the integrator makes sure that the initial state of the system is configured correctly. The next step is to allocate memory to this computation — a procedure that will get you the initial value, while modifying the parameter record dependency of the function accordingly.

The following stage is to do a type conversion, given that in order to create the `Integrator` record, it is necessary to have the type `IORef (Dynamics Double)`. At first glance, this can seem to be an issue because the result of the *newIORef* function is wrapped with the `IO` monad [1]. This conversion is the reason why the `IO` monad is being used in the implementation, and hence forced us to implement the typeclass `MonadIO`. The function `liftIO` is capable of removing the `IO` wrapper and adding an arbitrary monad in its place, `Dynamics` in this case. So, after line 3 the `comp` value has the desired `Dynamics` type. The remaining step of this creation process is to construct the integrator itself by

[1] `IORef` hackage documentation.

building up the record with the correct fields, e.g., the dynamic version of the initial value and the pointer to the constructed computation written in memory (lines 4 and 5).

```
1  readInteg :: Integrator -> Dynamics Double
2  readInteg integ =
3    Dynamics $ \ps ->
4    do (Dynamics m) <- readIORef (computation integ)
5       m ps
```

To read the content of this region, it is necessary to provide the integrator to the *readInteg* function. Its implementation is straightforward: build a new `Dynamics` that applies the given record of `Parameters` (line 5) to what's being stored in the region (line 4). This is accomplished by using `do-notation` with the *readIORef* function [1].

Finally, the function *diffInteg* is a side-effect-only function that changes **which computation** will be used by the integrator. It is worth noticing that after the creation of the integrator, the `computation` pointer is addressing a simple and, initially, useless computation: given an arbitrary record of `Parameters`, it will fix it to assure it is starting at t_0, and it will return the initial value in form of a `Dynamics Double`. To update this behaviour, the *diffInteg* change the content being pointed by the integrator's pointer:

```
1  diffInteg :: Integrator -> Dynamics Double -> Dynamics ()
2  diffInteg integ diff =
3    do let z = Dynamics $ \ps ->
4             do whatToDo <- readIORef (computation integ)
5                let i = initial integ
6                case method (solver ps) of
7                  Euler -> integEuler diff i whatToDo ps
8                  RungeKutta2 -> integRK2 diff i whatToDo ps
9                  RungeKutta4 -> integRK4 diff i whatToDo ps
10     liftIO $ writeIORef (computation integ) z
```

In the beginning of the function (line 3), we create a new computation, so-called `z` — a function wrapped in the `Dynamics` type that receives a `Parameters` record and computes the result based on the solving method. In `z`, the first step is to build a copy of the **same process** being pointed by the `computation` pointer (line 4), and get the initial condition of the system (line 5). Finally, after checking the chosen solver (line 6), it is executed one iteration of the process by calling *integEuler*, or *integRK2* or *integRK4*. After line 10, this entire process `z` is being pointed by the `computation` pointer, being done by the *writeIORef* function [1]. It may seem confusing that inside `z` we are **reading** what is being pointed and later, on the last line of *diffInteg*, this is being used on the final line to update that same pointer. This is necessary, as it will be explained in the next chapter *Execution*

Walkthrough, to allow the use of an **implicit recursion** to assure the sequential aspect needed by the solvers. For now, the core idea is this: the *diffInteg* function alters the **future** computations; it rewrites which procedure will be pointed by the `computation` pointer. This new procedure, which we called `z`, creates an intermediate computation, `whatToDo` (line 4), that **reads** what this pointer is addressing, which is `z` itself.

Initially, this strange behaviour may cause the idea that this computation will never halt. However, Haskell's *laziness* assures that a given computation will not be computed unless it is necessary to continue execution and this is **not** the case in the current stage, given that we are just setting the environment in the memory to further calculate the solution of the system.

3.4 GPAC Bind II: Integrator

The `Integrator` type introduced in the previous section corresponds to FF-GPAC's forth and final basic unit, the integrator. The analog version of the integrator used in FF-GPAC had the goal of using physical systems (shafts and gears) that obeys the same mathematical relations that control other physical or technical phenomenon under investigation [11]. In contrast, the integrator modeled in Rivika uses pointers in a digital computer that point to iteration-based algorithms that can approximate the solution of the problem at a requested moment t in time.

Lastly, there are the composition rules in FF-GPAC — constraints that describe how the units can be interconnected. The following are the same composition rules presented in chapter 2, *Design Philosophy*, section 2.1:

1. An input of a polynomial circuit should be the input t or the output of an integrator. Feedback can only be done from the output of integrators to inputs of polynomial circuits.
2. Each polynomial circuit admit multiple inputs
3. Each integrand input of an integrator should be generated by the output of a polynomial unit.
4. Each variable of integration of an integrator is the input t.

The preceding rules include defining connections with polynomial circuits — an acyclic circuit composed only by constant functions, adders and multipliers. These special circuits are already being modeled in `Rivika` by the `Dynamics` type with a set of typeclasses, as explained in the previous section about GPAC. The **integrator functions**, e.g., *readInteg* and *diffInteg*, represent the composition rules.

Going back to the type signature of the *diffInteg*, `Integrator -> Dynamics Double -> Dynamics ()`, we can interpret this function as a **wiring** operation. This function connects as an input of the integrator, represented by the **Integrator** type, the output of a polynomial circuit, represented by the value with `Dynamics Double` type. Because the operation is just setting up the connections between the two, the functions ends with the type `Dynamics ()`.

A polynomial circuit can have the time t or an output of another integrator as inputs, with restricted feedback (rule 1). This rule is being matched by the following: the `Dynamics` type makes time available to the circuits, and the *readInteg* function allows us to read the output of another integrators. The second rule, related to multiple inputs in the combinational circuit, is being followed because we can link inputs using arithmetic operations, feature provided by the `Num` typeclass. Moreover, because the sole purpose of `Rivika` is to solve differential equations, we are **only** interested in circuits that calculates integrals, meaning that it is guaranteed that the integrand of the integrator will always be the output of a polynomial unit (rule 3), as we saw with the type signature of the *diffInteg* function. The forth rule is also being attended it, given that the solver methods inside the *diffInteg* function always calculate the integral in respect to the time variable. Figure 3.7 summarizes these last mappings between the implementation, and FF-GPAC's integrator and rules of composition.

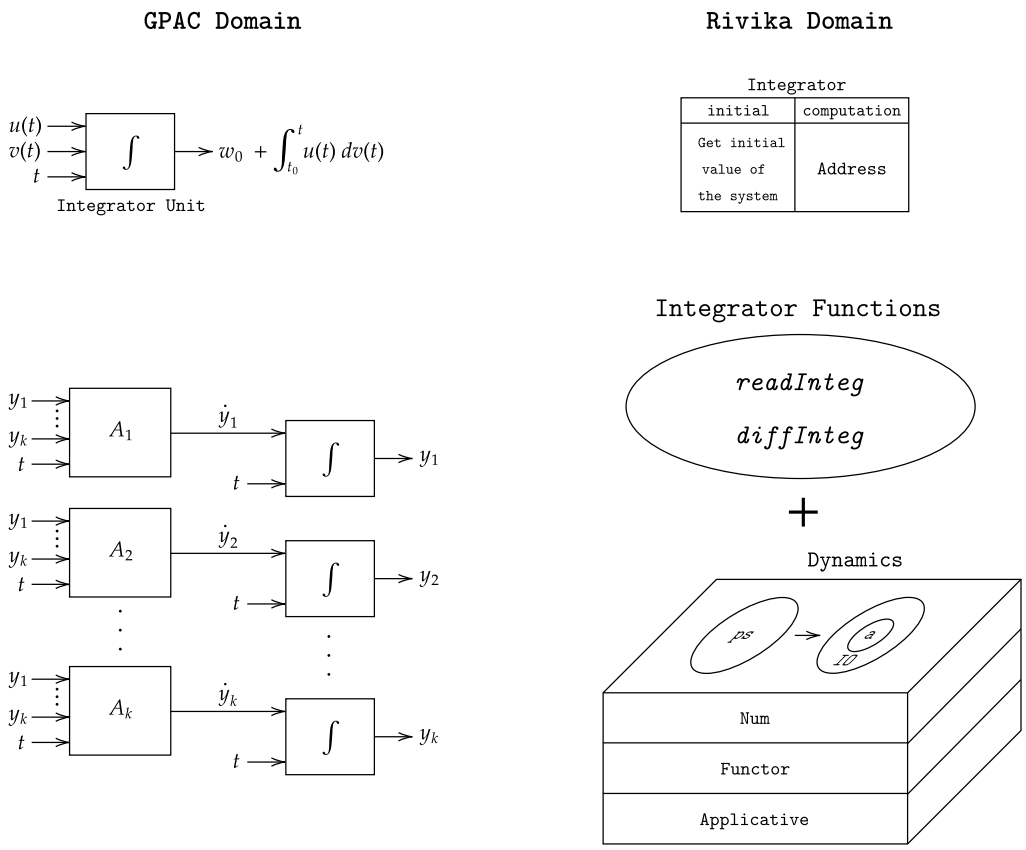


Figure 3.7: The integrator functions attend the rules of composition of FF-GPAC, whilst the `Dynamics` and `Integrator` types match the four basic units.

3.5 Using Recursion to solve Math

The remaining topic of this chapter is to describe in detail how the solver methods are being implemented. There are three solvers currently implemented:

- Euler Method or First-order Runge-Kutta Method
- Second-order Runge-Kutta Method
- Forth-order Runge-Kutta Method

To explain how the solvers work and their nuances, it is useful to go into the implementation of the simplest one — the Euler method. However, the implementation of the solvers use a slightly different function for the next step or iteration in comparison to the one explained in chapter 2. Hence, it is worthwhile to remember how this method originally iterates in terms of its mathematical description and compare it to the new function. From equation 2.2, we can obtain a different function to next step, by subtracting the index from both sides of the equation:

$$y_{n+1} = y_n + hf(t_n, y_n) \rightarrow y_n = y_{n-1} + hf(t_{n-1}, y_{n-1}) \tag{3.1}$$

The value of the current iteration, y_n, can be described in terms of the sum of the previous value and the product between the time step h with the differential equation from the previous iteration and time. With this difference taken into account, the following code is the implementation of the Euler method. In terms of main functionality, the family of Runge-Kutta methods is analogous:

```
1  integEuler :: Dynamics Double
2                -> Dynamics Double
3                -> Dynamics Double
4                -> Parameters -> IO Double
5  integEuler (Dynamics diff) (Dynamics init) (Dynamics compute) ps =
6    case iteration ps of
7      0 ->
8        init ps
9      n -> do
10       let iv  = interval ps
11           sl  = solver ps
12           ty  = iterToTime iv sl (n - 1) 0
13           prevPS = ps { time = ty, iteration = n - 1, solver = sl { stage = 0} }
14       a <- compute prevPS
15       b <- diff prevPS
16       let !v = a + dt (solver ps) * b
17       return v
```

On line 5, it is possible to see which functions are available in order to execute a step in the solver. The dependency `diff` is the representation of the differential equation itself. The initial value, $y(t_0)$, can be obtained by applying any `Parameters` record to the `init` dependency function. The next dependency, `compute`, execute everything previously defined in *diffInteg*; thus effectively executing a new step using the **same** solver. The result of `compute` depends on which parametric record will be applied, meaning that we call a new and different solver step in the current one, potentially building a chain of solver step calls. This mechanism — of executing again a solver step, inside the solver itself — is the aforementioned implicit recursion, described in the earlier section. By changing the `ps` record to the **previous** moment and iteration with the solver starting from initial stage, it is guaranteed that for any step the previous one can be computed, a requirement when using numerical methods.

With this in mind, the solver function treats the initial value case as the base case of the recursion, whilst it treats normally the remaining ones (line 9). In the base case (lines 7 and 8), the calculation can be done by doing an application of `ps` to `init`. Otherwise, it

is necessary to know the result from the previous iteration in order to generate the current one. To address this requirement, the solver builds another parametric record (lines 10 to 13) and call another solver step (line 14). Also, it calculates the value from applying this record to `diff` (line 15), the differential equation, and finally computes the result for the current iteration (line 16). It is worth noting that the use of `let!` is mandatory, given that it forces evaluation of the expression instead of lazily postponing the computation, making it execute everything in order to get the value `v` (line 17).

This finishes this chapter, where we incremented the capabilities of the `Dynamics` type and used it in combination with a brand-new type, the `Integrator`. Together these types represent the mathematical integral operation. The solver methods are involved within this implementation, and they use an implicit recursion to maintain their sequential behaviour. Also, those abstractions were mapped to FF-GPAC's ideas in order to bring some formalism to the project. However, the used mechanisms, such as implicit recursion and memory manipulation, make it hard to visualize how to execute the project given a description of a physical system. The next chapter, *Execution Walkthrough*, will introduce the **driver** of the simulation and present a step-by-step concrete example.

Chapter 4

Execution Walkthrough

Previously, we presented in detail the latter core type of the implementation, the `Integrator`, as well as why it can model an integral when used with the `Dynamics` type. This chapter is a follow-up, and its objectives are threefold: describe how to map a set of differential equations to an executable model, reveal which functions execute a given example and present a guided-example as a proof-of-concept.

4.1 From Models to Models

Systems of differential equations reside in the mathematical domain. In order to **execute** using the `Rivika` DSL, this model needs to be converted into an executable model following the DSL's guidelines. Further, we saw that these requirements resemble FF-GPAC's description of its basic units and rules of composition. Thus, these mappings between the three worlds need to be established. Chapters 2 and 3 explained the mapping between `Rivika` and FF-GPAC. It remains to map the *semantics* of the mathematical world to the *operational* world of `Rivika`. This mapping goes as the following:

- The relationship between the derivatives and their respective functions will be modeled by **feedback** loops with `Integrator` type.
- The initial condition will be modeled by the `initial` pointer within an integrator.
- Combinational aspects, such as addition and multiplication of constants and the time t, will be represented by typeclasses and the `Dynamics` type.

With that in mind, Figure 4.1 illustrates an example of a model in `Rivika`, alongside its mathematical counterpart. Further, Figure 4.2 shows which FF-GPAC circuit each line is modeling. This pipeline effectively makes `Rivika` a bridge between a physical system, modeled by differential equations, and the FF-GPAC model proposed by Graça [5].

```haskell
t :: Dynamics Double
t = Dynamics $ \ps -> return (time ps)

exampleModel :: Dynamics Double
exampleModel =
  do integ <- newInteg 1
     let y = readInteg integ
     diffInteg integ (y + t)
     y
```

$$\dot{y} = y + t \qquad y(0) = 1$$

Figure 4.1: The integrator functions are essential to create and interconnect combinational and feedback-dependent circuits.

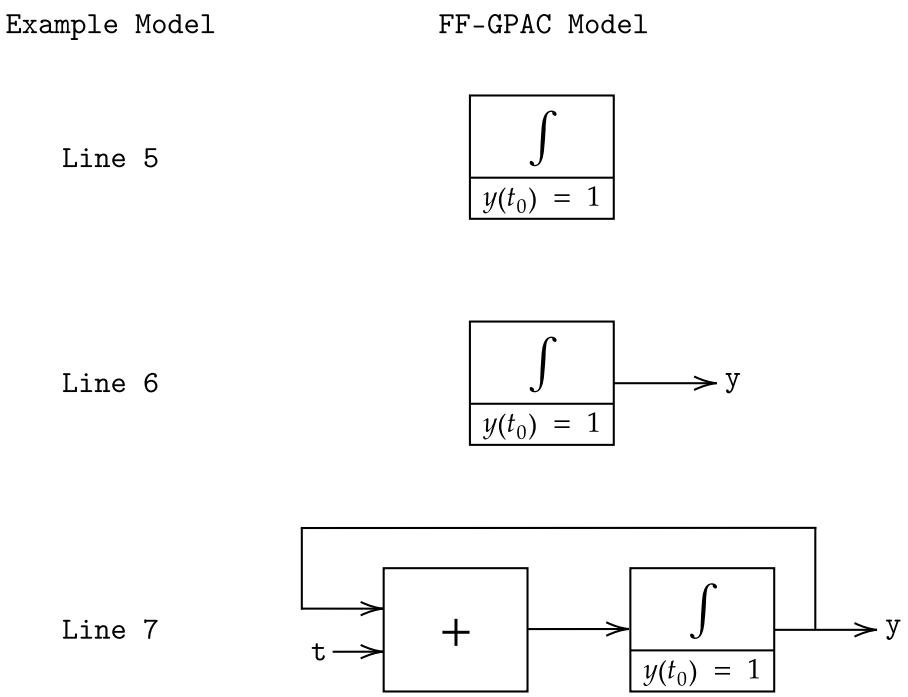


Figure 4.2: The developed DSL translates a system described by differential equations to an executable model that resembles FF-GPAC's description.

In line 5, a record with type `Integrator` is created, with 1 being the initial condition of the system. Line 6 creates a **state variable**, a label that gives us access to the output of an integrator, `integ` in this case. Afterward, in line 7, the *diffInteg* function connects the inputs to a given integrator by creating a combinational circuit, `(y + t)`. Polynomial circuits and integrators' outputs can be used as available inputs, as well as the *time* of the simulation. Finally, line 8 returns the state variable as the output for the **driver**, the main topic of the next section.

There is, however, an useful improvement to be made into the definition of a model within the DSL. The presented example used only a single state variable, although it

is common to have **multiple** state variables, i.e., multiple integrators interacting with each other, modeling different aspects of a given scenario. Moreover, when dealing with multiple state variables, it is important to maintain **synchronization** between them, i.e., the same `Parameters` is being applied to **all** state variables at the same time.

To address both of these requirements, we will use the *sequence* function, available in Haskell's standard library. This function manipulates **nested** structures and change their internal structure. The only requirement is that the outer type have to implement the `Traversable` typeclass. For instance, applying this function to a list of values of type `Maybe` would generate a single `Maybe` value in which its content is a list of the previous content individually wrapped by the `Maybe` type. This is only possible because the external or "bundler" type, list in this case, has implemented the `Traversable` typeclass. Figure 4.3 depicts the example before and after applying the function.

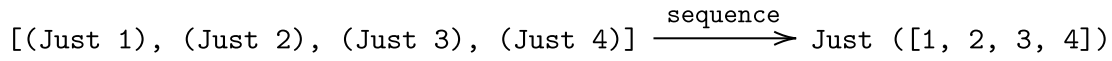


Figure 4.3: Because the list implements the `Traversable` typeclass, it allows this type to use the *traverse* and *sequence* functions, in which both are related to changing the internal behaviour of the nested structures.

Similarly to the preceding example, the list structure will be used to involve all the state variables with type `Dynamics Double`. This tweak is effectively creating a **vector** of state variables whilst sharing the same notion of time across all of them. So, the final type signature of a model is `Dynamics [Double]` or, by using a type aliases for `[Double]` as `Vector`, `Dynamics Vector`. A second alias can be created to make it more descriptive, as exemplified in Figure 4.4:

```haskell
type Vector = [Double]
type Model a = Dynamics a

exampleModel :: Model Vector
exampleModel =
  do integX <- newInteg 1
     integY <- newInteg 1
     let x = readInteg integX
         y = readInteg integY
     diffInteg integX (x * y)
     diffInteg integY (y + t)
     sequence [x, y]
```

$$\dot{x} = y * x \qquad x(0) = 1$$
$$\dot{y} = y + t \qquad y(0) = 1$$

Figure 4.4: A **state vector** comprises multiple state variables and requires the use of the *sequence* function to sync time across all variables.

Finally, when creating a model, the same steps have to be done in the same order, always starting with the integrator functions and finishing with the *sequence* function

being applied to a state vector. So, Figure 4.5 depicts the general pipeline used to create any model in both the semantics and operational perspectives:

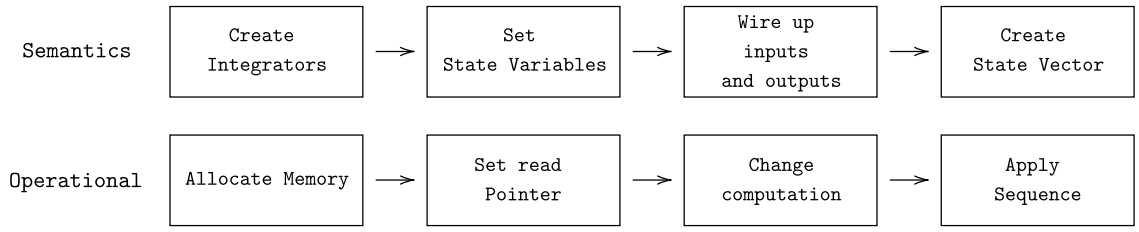


Figure 4.5: When building a model for simulation, the above pipeline is always used, from both points of view. The operations with meaning, i.e., the ones in the `Semantics` pipeline, are mapped to executable operations in the `Operational` pipeline, and vice-versa.

4.2 Driving the Model

Given a physical model translated to an executable one, it remains to understand which functions drive the simulation, i.e., which functions take the simulations details into consideration and generate the output. The function *runDynamics* fulfills this role:

```
1 runDynamics :: Model a -> Interval -> Solver -> IO [a]
2 runDynamics (Dynamics m) iv sl =
3   do let (nl, nu) = iterationBnds iv (dt sl)
4          parameterise n = Parameters { interval = iv,
5                                       time = iterToTime iv sl n 0,
6                                       iteration = n,
7                                       solver = sl { stage = 0 }}
8      sequence $ map (m . parameterise) [nl .. nu]
```

On line 3, we convert the *time* interval of the simulation to an *iteration* interval in the format of a tuple, i.e., the continuous interval becomes the tuple $(0, \frac{stopTime - startTime}{timeStep})$, in which the second value of the tuple is **rounded**. From line 4 to line 7, we are defining an auxiliary function *parameterise*. This function picks a natural number, which represents the iteration index, and creates a new record with the type `Parameters`. Additionally, it uses the auxiliary function *iterToTime* (line 5), which converts the iteration number from the domain of discrete **steps** to the domain of **discrete time**, i.e., the time the solver methods can operate with (chapter 5 will explore more of this concept). This conversion is based on the time step being used, as well as which method and in which stage it is for that specific iteration. Finally, line 8 produces the outcome of the *runDyanmics* function. The final result is the output from a function called *map* piped it as an argument for the *sequence* function.

The *map* operation is provided by the `Functor` of the list monad, and it applies an arbitrary function to the internal members of a list in a **sequential** manner. In this case, the *parameterise* function, composed with the dynamic application `m`, is the one being mapped. Thus, a custom value of the type `Parameters` is taking place of each natural natural number in the list, and this is being applied to the received `Dynamics` value. It produces a list of answers in order, each one wrapped in the `IO` monad. To abstract out the `IO`, thus getting `IO [a]` rather than `[IO a]`, the *sequence* function finishes the implementation.

Additionally, there is an analogous implementation of this function, so-called *runDynamicsFinal*, that return only the final result of the simulation, i.e., $y(stopTime)$, instead of the outputs at the time step samples.

4.3 An attractive example

For the example walkthrough, the same example introduced in the chapter *Introduction* will be used in this section. So, we will be solving a system, composed by a set of chaotic solutions, called **the Lorenz Attractor**. In these types of systems, the ordinary differential equations are used to model chaotic systems, providing solutions based on parameter values and initial conditions. The original differential equations are presented bellow:

$$\sigma = 10.0$$

$$\rho = 28.0$$

$$\beta = \frac{8.0}{3.0}$$

$$\frac{dx}{dt} = \sigma(y(t) - x(t))$$

$$\frac{dy}{dt} = x(t)(\rho - z(t))$$

$$\frac{dz}{dt} = x(t)y(t) - \beta z(t)$$

It is straight-forward to map it to the described domain-specific language (DSL). The remaining details are simulation-related, e.g., which solver method will be used, the

interval of the simulation, as well as the size of the time step. Taking into account that the constants σ, ρ and β need to be set, the code below summarizes it, and Figure 4.6 shows its FF-GPAC circuit:

```
lorenzInterv = Interval { startTime = 0,
                          stopTime = 100 }

lorenzSolver = Solver { dt = 0.01,
                        method = RungeKutta2,
                        stage = 0
                      }

sigma = 10.0
rho = 28.0
beta = 8.0 / 3.0

lorenzModel :: Model Vector
lorenzModel =
  do integX <- newInteg 1.0
     integY <- newInteg 1.0
     integZ <- newInteg 1.0
     let x = readInteg integX
         y = readInteg integY
         z = readInteg integZ
     diffInteg integX (sigma * (y - x))
     diffInteg integY (x * (rho - z) - y)
     diffInteg integZ (x * y - beta * z)
     sequence [x, y, z]

lorenzSystem = runDynamics lorenzModel lorenzInterv lorenzSolver
```

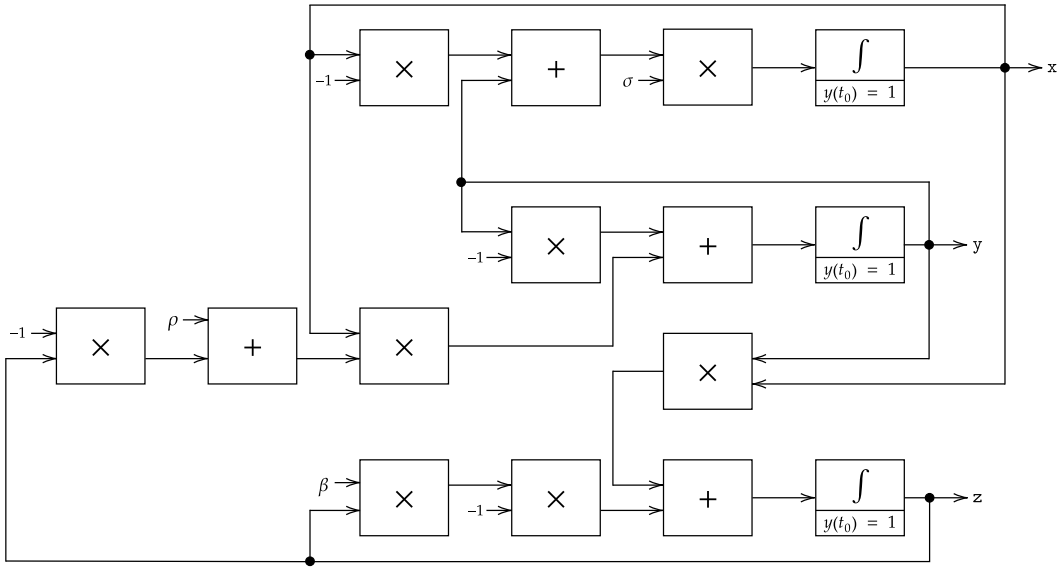


Figure 4.6: Using only FF-GPAC's basic units and their composition rules, it's possible to model the Lorenz Attractor example.

The first two records, `Interval` and `Solver`, sets the environment (lines 1 to 6). The former determines the simulation interval (lines 1 and 2), from start to finish, and the latter configures the solver with 0.01 seconds as the time step, whilst executing the second-order Runge-Kutta method from the initial stage (lines 3 to 6). The *lorenzModel*, presented after setting the constants (lines 7 to 9), executes the aforementioned pipeline to create the model: allocate memory (lines 12 to 14), create read-only pointers (lines 15 to 17), change the computation (lines 18 to 20) and dispatch it (line 21). Finally, the function *lorenzSystem* groups everything together calling the *runDynamics* driver (line 22).

After this overview, let's follow the execution path used by the compiler. Haskell's compiler works in a lazily manner, meaning that it calls for execution only the necessary parts. So, the first step calling *lorenzSystem* is to call the *runDynamics* function with a model, interval and solver configurations. Following its path of execution, the *map* function (inside the driver) forces the application of a parametric record generated by the *parameterise* function to the provided model, *lorenzModel* in this case. Thus, it needs to be executed in order to return from the *runDynamics* function.

To understand the model, we need to follow the execution sequence of the output: `sequence [x, y, z]`, which requires executing all the lines before this line to obtain the all the state variables. For the sake of simplicity, we will follow the execution of the operations related to the x variable, given that the remaining variables have an analogous execution walkthrough. First and foremost, memory is allocated for the integrator to

work with (line 12). Figure 4.7 depicts this idea, as well as being a reminder of what the *newInteg* and *initialize* functions do, described in the chapter *Effectful Integrals*. In this image, the integrator `integX` comprises two fields, `initial` and `computation`. The former is a simple value of the type `Dynamics Double` that, regardless of the parameters record it receives, it returns the initial condition of the system. The latter is a pointer or address that references a specific `Dynamics Double` computation in memory: in the case of receiving a parametric record `ps`, it fixes potential problems with it via the `initialize` block, and it applies this fixed value in order to get `i`, i.e., the initial value 1, the same being saved in the other field of the record, `initial`.

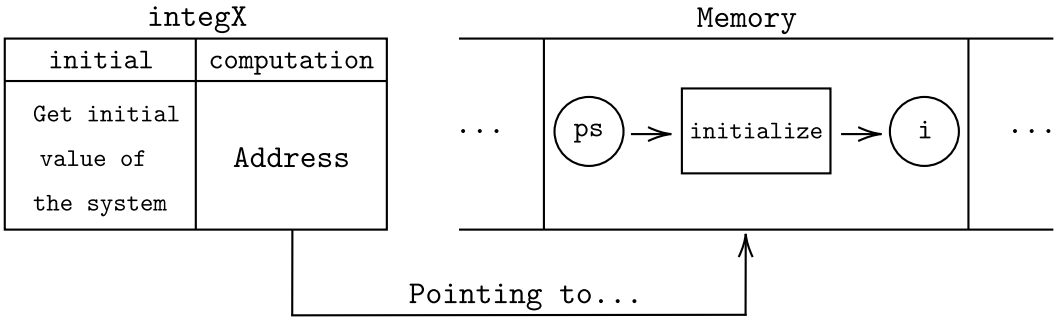


Figure 4.7: After *newInteg*, this record is the final image of the integrator. The function *initialize* gives us protecting against wrong records of the type `Parameters`, assuring it begins from the first iteration, i.e., t_0.

The next step is the creation of the independent state variable x via *readInteg* function (line 15). This variable will read the computations that are executing under the hood by the integrator. The core idea is to read from the computation pointer inside the integrator and create a new `Dynamics Double` value. Figure 4.8 portrays this mental image. When reading a value from an integrator, the computation pointer is being used to access the memory region previously allocated. Also, what's being stored in memory is a `Dynamics Double` value. The state variable, x in this case, combines its received `Parameters` value, so-called `ps`, and **applies** it to the stored dynamic function. The result `v` is then returned.

The final step is to **change** the computation **inside** the memory region (line 18). Until this moment, the stored computation is always returning the value of the system at t_0, whilst changing the obtained parameters record to be correct via the *initialize* function. Our goal is to modify this behaviour to the actual solution of the differential equations via using numerical methods, i.e., using the solver of the simulation. The function *diffInteg* fulfills this role and its functionality is illustrated in Figure 4.9. With the integrator `integX` and the differential equation $\sigma(y - x)$ on hand, this function picks the provided parametric record `ps` and it returns the result of a step of the solver `RK2`, second-order Runge-Kutta method in this case. Additionally, the solver method receives

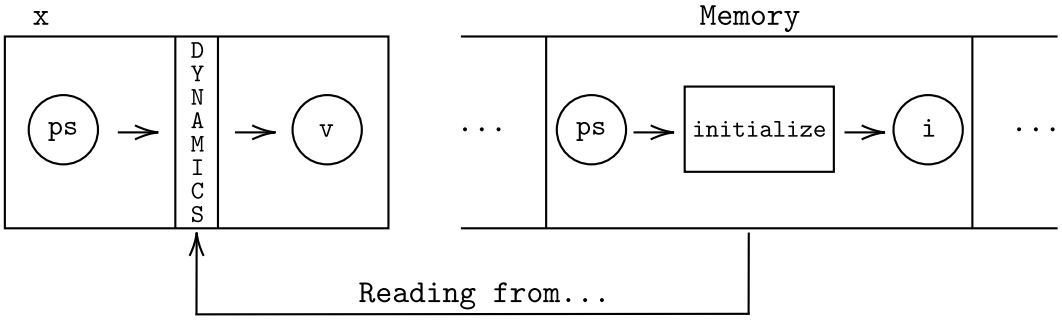


Figure 4.8: After *readInteg*, the final floating point values is obtained by reading from memory a dynamic computation and passing to it the received parameters record. The result of this application, v, is the returned value.

as a dependency what is being pointed by the `computation` pointer, represented by `c` in the image, alongside the differential equation and initial value, pictured by `d` and `i` respectively.

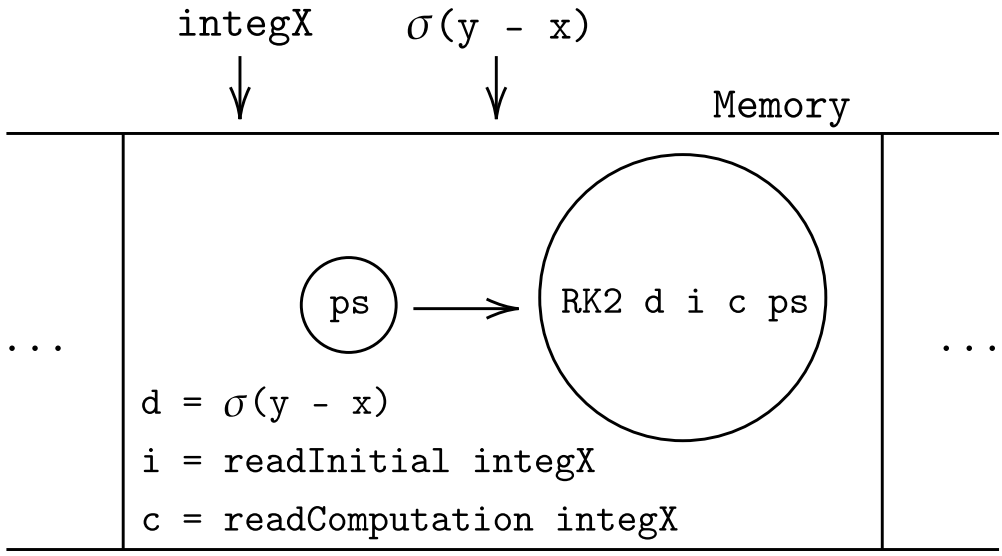


Figure 4.9: The *diffInteg* function only does side effects, meaning that only affects memory. The internal variable `c` is a pointer to the computation *itself*, i.e., the dynamic computation being created references this exact procedure.

Figure 4.10 shows the final image for state variable x after until this point in the execution.

Lastly, the state variable is wrapped inside a list and it is applied to the *sequence* function, as explained in the previous section. This means that the list of variable(s) in the model, with the signature `[Dynamics Double]`, is transformed into a value with the type `Dynamics [Double]`. The transformation can be visually understood when looking

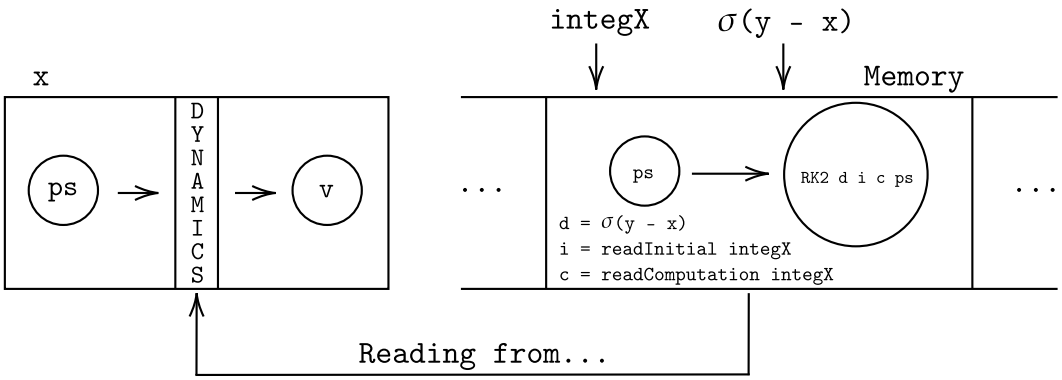


Figure 4.10: After setting up the environment, this is the final depiction of an independent variable. The reader x reads the values computed by the procedure stored in memory, a second-order Runge-Kutta method in this case.

at Figure 4.10. Instead of picking one `ps` of type `Parameters` and returning a value v, the same parametric record returns a **list** of values, with the **same** parametric dependency being applied to all state variables inside $[x, y, z]$.

However, this only addresses **how** the driver triggers the entire execution, but does **not** explain how the differential equations are actually being calculated with the `RK2` numerical method. This is done by the solver functions (*integEuler*, *integRK2* and *integRK4*) and those are all based on equation 3.1 regardless of the chosen method. The equation goes as the following:

$$y_{n+1} = y_n + hf(t_n, y_n) \rightarrow y_n = y_{n-1} + hf(t_{n-1}, y_{n-1})$$

The equation above makes the dependencies in the `RK2` example in Figure 4.10 clear:

- `d` $\Rightarrow$ Differential Equation that will be used to obtain the value of the previous iteration ($f(t_{n-1}, y_{n-1})$).
- `ps` $\Rightarrow$ Parametric record with solver information, such as the size of the time step (h).
- `i` and `c` $\Rightarrow$ The initial value of the system, as well as a solver step function, will be used to calculate the previous iteration result (y_{n-1}).

It is worth mentioning that the dependency `c` is a call of a **solver step**, meaning that it is capable of calculating the previous step y_{n-1}. This is accomplished in a **recursive** manner, since for every iteration the previous one is necessary. When the base case is achieved, by calculating the value at the first iteration using the `i` dependency, the recursion stops and the process folds, getting the final result for the iteration that has

started the chain. This is the same pattern across all the implemented solvers (`Euler`, `RungeKutta2` and `RungeKutta4`).

4.4 Lorenz's Butterfly

After all the explained theory behind the project, it remains to be seen if this can be converted into practical results. With certain constant values, the generated graph of the Lorenz's Attractor example used in the last chapter is known for oscillation and getting the shape of two fixed point attractors, meaning that the system evolves to an oscillating state even if slightly disturbed. As showed in Figure 4.11, the obtained graph from the Lorenz's Attractor model matches what was expected for a Lorenz's system. It is worth noting that changing the values of σ, ρ and β can produce completely different answers, destroying the resembled "butterfly" shape of the graph.

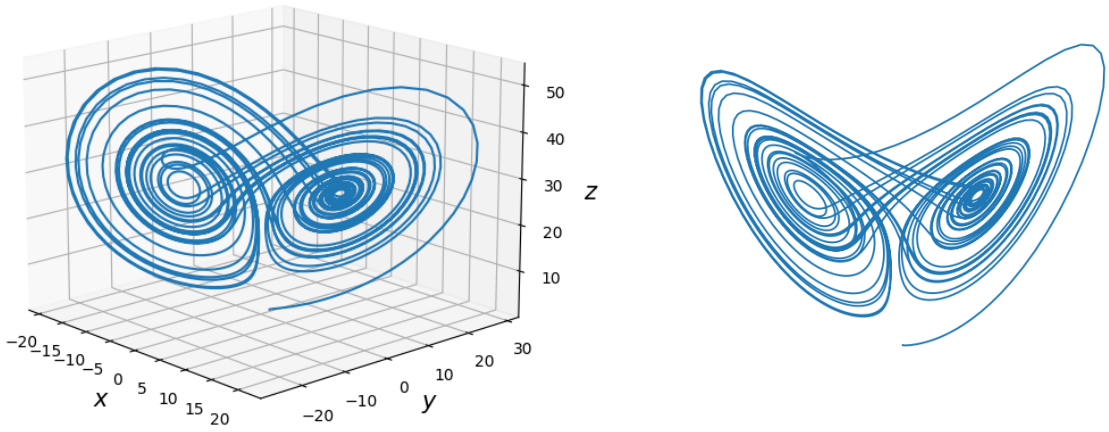


Figure 4.11: The Lorenz's Attractor example has a very famous butterfly shape from certain angles and constant values in the graph generated by the solution of the differential equations.

Although correct, the presented solution has a few drawbacks. The next two chapters will explain and address the two identified problems with the current implementation.

Chapter 5

Travelling across Domains

The previous chapter ended anouncing that drawbacks are present in the current implementation. This chapter will introduce the first concern: numerical methods do not reside in the continuous domain, the one we are actually interested in. After this chapter, this domain issue will be addressed via **interpolation**, with a few tweaks in the integrator and driver.

5.1 Time Domains

When dealing with continuous time, `Rivika` changes the domain in which **time** is being modeled. Figure 5.1 shows the domains that the implementation interact with during execution:

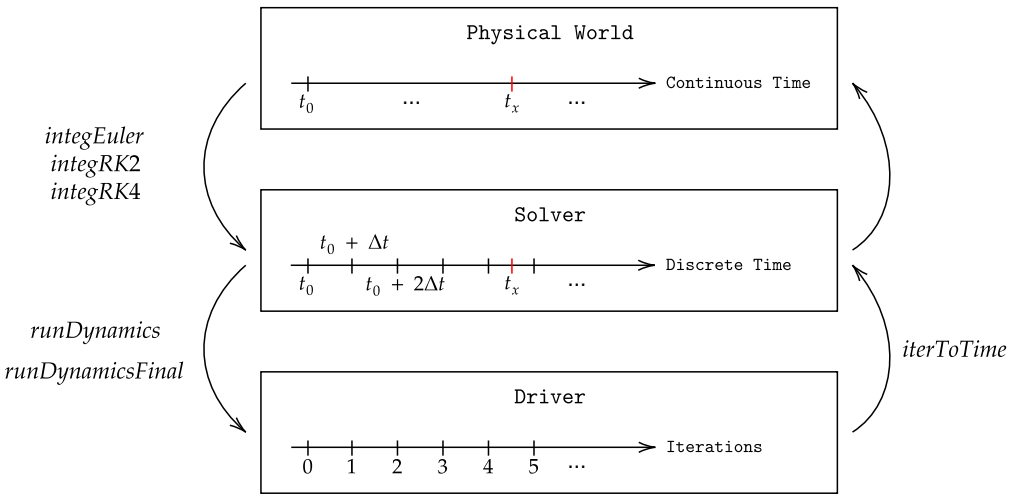


Figure 5.1: During simulation, functions change the time domain to the one that better fits certain entities, such as the `Solver` and the driver. The image is heavily inspired by a figure in [2].

The problems starts in the physical domain. The goal is to obtain a value of an unknown function $y(t)$ at time t_x. However, because the solution is based on **numerical methods** a sampling process occurs and the continuous time domain is transformed into a **discrete** time domain, where the solver methods reside — those are represented by the functions *integEuler*, *integRK2* and *integRK4*. A solver depends on the chosen time step to execute a numerical algorithm. Thus, time is modeled by the sum of t_0 with $n\Delta$, where n is a natural number. Hence, from the solver perspective, time is always dependent on the time step, i.e., only values that can be described as $t_0 + n\Delta$ can be properly visualized by the solver. Finally, there's the **iteration** domain, used by the driver functions, *runDynamics* and *runDynamicsFinal*. When executing the driver, one of its first steps is to call the function *iterationsBnds*, which converts the simulation time interval to a tuple of numbers that represent the amount of iterations based on the time step of the solver. This functions is presented bellow:

```
1 iterationBnds :: Interval -> Double -> (Int, Int)
2 iterationBnds interv dt = (0, round ((stopTime interv -
3                                 startTime interv) / dt))
```

To achieve the total number of iterations, the function *iterationBnds* does a **round** operation on the sampled result of iterations, based on the time interval (*startTime* and *stopTime*) and the time step (`dt`). The second member of the tuple is always the answer, given that it is assumed that the first member of the tuple is always zero.

The function that allows us to go back to the discrete time domain being in the iteration axis is the *iterToTime* function. It uses the solver information, the current iteration and the interval to transition back to time, as depicted by the following code:

```
1  iterToTime :: Interval -> Solver -> Int -> Int -> Double
2  iterToTime interv solver n st =
3    if st < 0 then
4      error "Incorrect stage: iterToTime"
5    else
6      (startTime interv) + n' * (dt solver) + delta (method solver) st
7        where n' = fromInteger (toInteger n)
8              delta Euler       0 = 0
9              delta RungeKutta2 0 = 0
10             delta RungeKutta2 1 = dt solver
11             delta RungeKutta4 0 = 0
12             delta RungeKutta4 1 = dt solver / 2
13             delta RungeKutta4 2 = dt solver / 2
14             delta RungeKutta4 3 = dt solver
```

A transformation from iteration to time depends on the chosen solver method due to their next step functions. For instance, the second and forth order Runge-Kutta methods

have more stages, and it uses fractions of the time step for more granular use of the derivative function. This is why lines 11 and 12 are using half of the time step. Moreover, all discrete time calculations assume that the value starts from the beginning of the simulation (*startTime*). The result is obtained by the sum of the initial value, the solver-dependent *delta* function and the iteration times the solver time step (line 6).

There is, however, a missing transition: from the discrete time domain to the domain of interest in CPS — the continuous time axis. This means that if the time value t_x is not present from the solver point of view, it is not possible to obtain $y(t_x)$. The proposed solution is to add an **interpolation** function into the pipeline, which addresses this transition. Thus, values in between solver steps will be transfered back to the continuous domain.

5.2 Tweak I: Interpolation

This tweak in the current implementation is divided into two parts: the driver and the integrator. These entities will communicate with each other to properly adapt the outcome. As mentioned previously, we will add an interpolation function to change from the discrete domain to the continuous one. However, this interpolation procedure needs to occur only in special situations: when it is not possible to model that specific point in time in the discrete time domain. Otherwise, the execution should continue as it is.

So, the proposed mechanism is the following: the driver will identify these corner cases and communicate to the integrator — via the `stage` field in the `Solver` data type — that the interpolation needs to be added into the pipeline of execution. When this flag is not on, i.e., the `stage` informs to continue execution normally, the implementation goes as the previous chapters detailed. This behaviour is altered **only** in particular scenarios, which the driver will be responsible for identifying.

Hence, it remains to re-implement the driver functions. The driver will notify the integrator that an interpolation needs to take place. Furthermore, the function *iterationBnds* will also be modified to use *ceiling* instead of *round*. The reason will be explained further on the line. The code below shows these changes:

```
1  iterationBnds :: Interval -> Double -> (Int, Int)
2  iterationBnds interv dt = (0, ceiling ((stopTime interv -
3                                startTime interv) / dt))

4  epslon = 0.00001

5  runDynamics :: Model a -> Interval -> Solver -> IO [a]
6  runDynamics (Dynamics m) iv sl =
```

```
7    do let (nl, nu) = basicIterationBnds iv (dt sl)
8           parameterise n = Parameters { interval = iv,
9                                         time = iterToTime iv sl n 0,
10                                        iteration = n,
11                                        solver = sl { stage = 0 }}
12          ps = Parameters { interval = iv,
13                            time = stopTime iv,
14                            iteration = nu,
15                            solver = sl { stage = -1}}
16      if (iterToTime iv sl nu 0) - (stopTime iv) < epslon
17      then sequence $ map (m . parameterise) [nl .. nu]
18      else sequence $ ((init $ map (m . parameterise) [nl .. nu]) ++ [m ps])
```

The new implementation of *iterationBnds* is pretty similar to the previous one, with the difference being the replacement of the *round* function for the *ceiling* function. As explained in the previous section, the rounding is used to go to the iteration domain. However, because the interpolation **requires** both solver steps — the one that came before t_x and the one immediately afterwards — the number of iterations needs always to surpass the requested time. For instance, the time 5.3 seconds will demand the fifth and sixth iterations with a time step of 1 second. When using *ceiling*, it is assured that the value of interest will be in the interval of computed values. So, when dealing with 5.3, the integrator will calculate all values up to 6 seconds.

Lines 5 to 11 are equal to the previous implementation of the *runDynamics* function. On line 12, a new record of type `Parameters` is being created, especifically to these special cases of mismatch between discrete and continuous time. The differences within this special record are relevant: the time field is being fulfilled with the actual stop time and the stage field of the solver is being set to **-1**. The latter is how the driver tells the integrator to apply the interpolation function. Later, as we will see, the integrator will check for negative values in the `stage` field and it will execute a slightly different pipeline.

This parametric record, however, will only be used if, and **only** if, we detect that a divergence took place, based on the configuration of the simulation. This is being checked in line 16, where it is being compared the conversion of the last iteration to time with the provided stop time in the `Interval` type record. If they are not discrepant by an `epslon` value, it means that the simulation can proceed normally. Otherwise, the stop time is between the two last iterations, and cannot be represented discreetly. So, the solution is to cut the last iteration, given that we know that it surpasses the end of the simulation, and add a new member to the list of outcomes, a computation where the altered `ps` record is being applied. This major step is happening on line 18.

Next, the integrator needs to be modified in order to cope with negative value in solver stages. The following *interpolate* function will be an added to the integrator:

```
1  interpolate :: Dynamics Double -> Dynamics Double
2  interpolate (Dynamics m) =
3    Dynamics $ \ps ->
4    if stage (solver ps) >= 0 then
5      m ps
6    else
7      let iv = interval ps
8          sl = solver ps
9          t  = time ps
10         st = dt sl
11         x  = (t - startTime iv) / st
12         n1 = max (floor x) (iterationLoBnd iv st)
13         n2 = min (ceiling x) (iterationHiBnd iv st)
14         t1 = iterToTime iv sl n1 0
15         t2 = iterToTime iv sl n2 0
16         z1 = m $ ps { time = t1,
17                       iteration = n1,
18                       solver = sl { stage = 0 } }
19         z2 = m $ ps { time = t2,
20                       iteration = n2,
21                       solver = sl { stage = 0 } }
22     in do y1 <- z1
23           y2 <- z2
24           return $ y1 + (y2 - y1) * (t - t1) / (t2 - t1)
```

Lines 4 to 6 are the normal workflow for positive values in the `stage` field. If a corner case comes in, the reminaing code applies **linear interpolation** to it. It accomplishes this by first comparing the next and previous discrete times (lines 14 and 15) relative to `x` (line 11) — the discrete counterpart of the time of interest `t` (line 9). These time points are calculated by their correspondent iterations (lines 12 and 13). Then, the integrator calculates the outcomes at these two points, i.e., do applications of the previous and next modeled times points with their respective parametric records (lines 16 to 21). Finally, lines 22 to 24 execute the linear interpolation with the obtained values that surround the non-discrete time point. Figure 5.2 illustrates the effect of the *interpolate* function when converting domains.

```
1  diffInteg :: Integrator -> Dynamics Double -> Dynamics ()
2  diffInteg integ diff =
3    do let z = Dynamics $ \ps ->
4             do whatToDo <- readIORef (computation integ)
5                let i = initial integ
6                case method (solver ps) of
7                  Euler -> integEuler diff i whatToDo ps
8                  RungeKutta2 -> integRK2 diff i whatToDo ps
```

```
9                RungeKutta4 -> integRK4 diff i whatToDo ps
10       liftIO $ writeIORef (computation integ) (interpolate z)
```

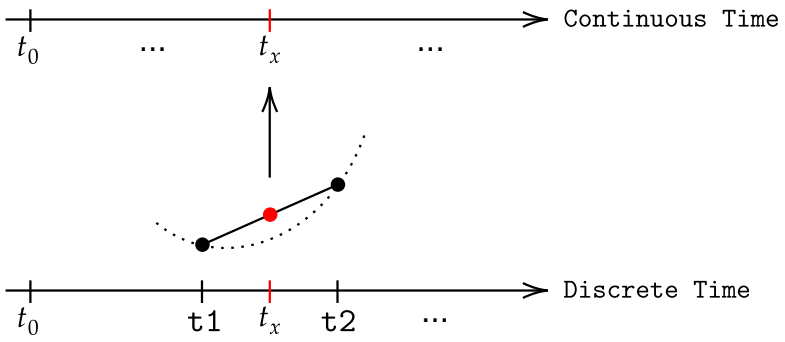


Figure 5.2: Linear interpolation is a transformation that transition us back to the continuous domain.

The last step in this tweak is to add this function into the integrator function *diffInteg*. The code is almost identical to the one presented in chapter 3, *Effectful Integrals*. The main difference is in line 10, where the interpolation function is being applied to `z`. Figure 5.3 shows the same visual representation for the *diffInteg* function used in chapter 4, but with the aforementioned modifications.

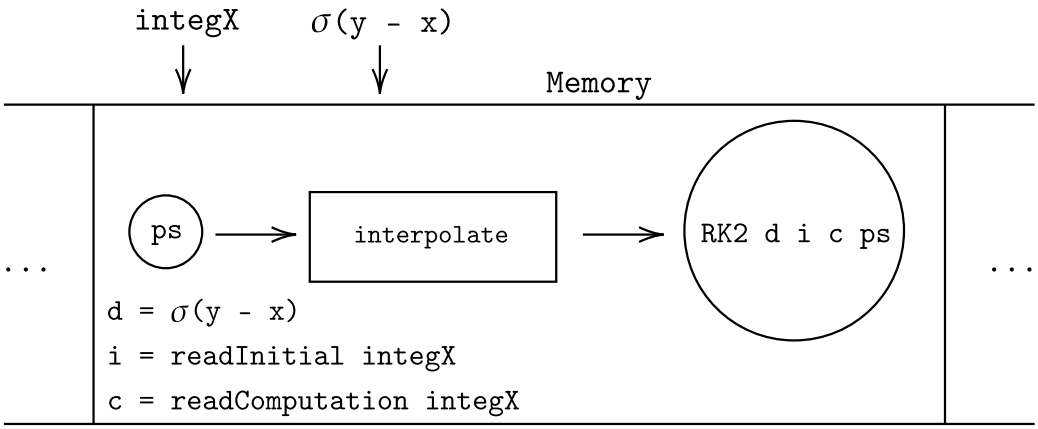


Figure 5.3: The new *diffInteg* function add linear interpolation to the pipeline when receiving a parametric record.

This concludes the first tweak in `Rivika`. Now, the mismatches between the stop time of the simulation and the time step are being treated differently, going back to the continuous domain thanks to the added linear interpolation. The next chapter, *Caching the Speed Pill*, goes deep into the program's performance and how this can be fixed with a caching strategy.

Chapter 6

Caching the Speed Pill

Chapter 5, *Travelling across Domains*, leveraged a major concern with the proposed software: the solvers don't work in the domain of interest, continuous time. This chapter, *Caching the Speed Pill*, addresses a second problem: the performance in `Rivika`. At the end of it, the simulation will be orders of magnitude faster by using a common modern caching strategy to speed up computing processes: memoization.

6.1 Performance

The simulations executed in `Rivika` take too long to run. For instance, to execute the Lorenz's Attractor example using the second-order Runge-Kutta method with an unrealistic time step size for real simulations (time step of 1 second), the simulator can take around **10 seconds** to compute 0 to 5 seconds of the physical system with a testbench using a 6-th generation quad-core (`i5`) Intel processor and 16GB of RAM. Increasing this interval shows an exponential growth in execution time, as depicted by Table 6.1 and by Figure 6.1 (values obtained after the interpolation tweak). Although the memory use is also problematic, it is hard to reason about those numbers due to Haskell's **garbage collector** [1], a memory manager that deals with Haskell's **immutability**. Thus, the memory values serve just to solidify the notion that `Rivika` is inneficient, showing an exponentinal growth in resource use, which makes it impractical to execute longer simulations and diminishes the usability of the proposed software.

[1] Garbage Collector wiki page.

Total of Iterations	Execution Time (seconds)	Consumed Memory (MB)
1	0.01	0.5
2	0.01	1.8
3	0.08	19.1
4	0.79	244.7
5	10.06	3198.7
6	140.95	41867.3
7	1798.16	548045.8
8	23801.51	7174008.0

Table 6.1: Small increases in the number of the iterations within the simulation provoke exponential penalties in performance.

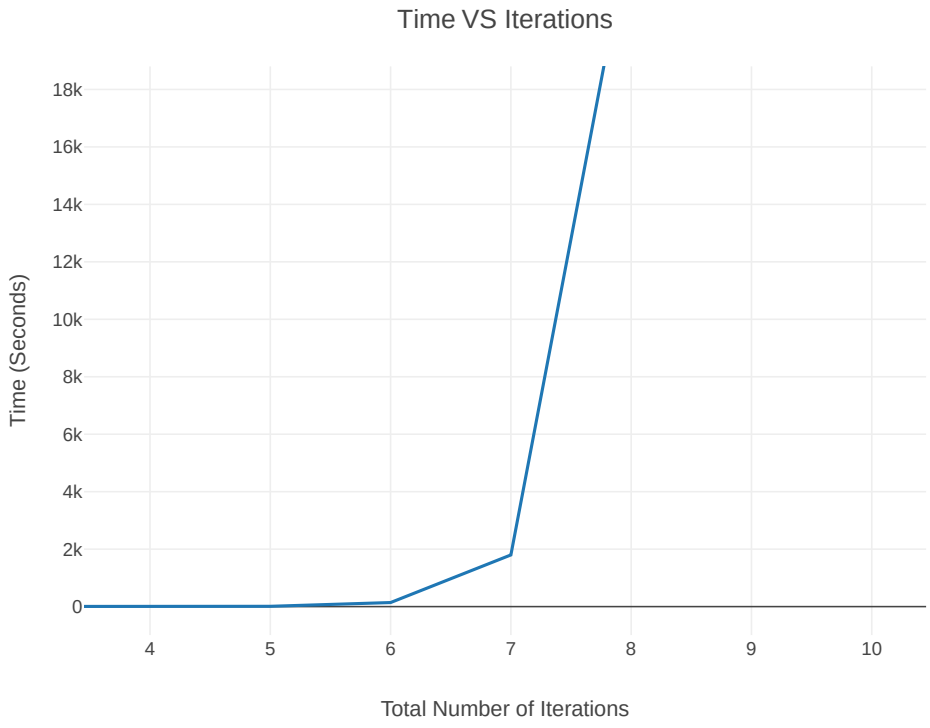


Figure 6.1: With just a few iterations, the exponential behaviour of the implementation is already noticeable.

6.2 The Saving Strategy

Before explaining the solution, it is worth describing **why** and **where** this problem arises. First, we need to take a look back onto the solvers' functions, such as the *integEuler*

function, introduced in chapter 3, *Effectful Integrals*:

```
1  integEuler :: Dynamics Double
2              -> Dynamics Double
3              -> Dynamics Double
4              -> Parameters -> IO Double
5  integEuler (Dynamics diff) (Dynamics init) (Dynamics compute) ps =
6    case iteration ps of
7      0 ->
8        init ps
9      n -> do
10       let iv  = interval ps
11           sl  = solver ps
12           ty  = iterToTime iv sl (n - 1) 0
13           prevPS = ps { time = ty, iteration = n - 1, solver = sl { stage = 0} }
14       a <- compute prevPS
15       b <- diff prevPS
16       let !v = a + dt (solver ps) * b
17       return v
```

From chapter 3, we know that lines 10 to 13 serve the purpose of creating a new parametric record to execute a new solver step for the **previous** iteration, in order to calculate the current one. From chapter 4, this code section turned out to be where the implicit recursion came in, because the current iteration needs to calculate the previous one. Effectively, this means that for **all** iterations, **all** previous steps from each one needs to be calculated. The problem is now clear: unnecessary computations are being made for all iterations, because the same solvers steps are not being saved for future steps, although these values do **not** change. In other words, to calculate step 3 of the solver, steps 1 and 2 are the same to calculate step 4 as well, but these values are being lost during the simulation.

To estimate how this lack of optimization affects performance, we can calculate how many solver steps will be executed to simulate theLorenz's Attractor example used in chapter 4, *Execution Walkthrough*. The Table 6.2 shows the total number of solver steps needed per iteration simulating the Lorenz example with the Euler method. In addition, the amount of steps also increase depending on which solver method is being used, given that in the higher order Runge-Kutta methods, multiple stages count as a new step as well.

Iteration	Total Solver Steps
1	1
2	3
3	6
4	10
5	15
6	21

Table 6.2: Because the previous solver steps are not saved, the total number of steps **per iteration** starts to accumullate following the numerical sequence of **triangular numbers** when using the Euler method.

This is the cause of the imense hit in performance. However, it also clarifies the solution: if the previous solver steps are saved, the next iterations don't need to re-compute them in order to continue. In the computer domain, the act of saving previous steps that do not change is called **memoization** and it is one form to execute **caching**. This optimization technique stores the values in a register or memory region and, instead of the process starts calculating the result again, it consults this region to quickly obtain the answer.

6.3 Tweak II: Memoization

The first tweak, *Memoization*, alters the `Integrator` type. The integrator will now have a pointer to the memory region that stores the previous computed values, meaning that before executing a new computation, it will consult this region first. Because the process is executed in a **sequential** manner, it is guaranteed that the previous result will be used. Thus, the accumulation of the solver steps will be addressed, and the amount of steps will be equal to the amount of iterations times how many stages the solver method uses.

The *memo* function creates this memory region for storing values, as well as providing read access to it. This is the only function in `Rivika` that uses a *constraint*, i.e., it restricts the parametric types to the ones that have implemented the requirement. In our case, this function requires that the internal type `Dynamics` dependency has implemented the `UMemo` typeclass. Because this typeclass is too complicated to be in the scope of this project, we will settle with the following explanation: it is required that the parametric values are capable of being contained inside an **mutable** array, which is the case for our `Double` values. As dependencies, the *memo* function receives the dynamic computation, as well as the interpolation function that is assumed to be used, in order to attenuate the domain problem described in the previous chapter. This means that at the end, the final result will be piped to the interpolation function.

```
1  memo :: UMemo e => (Dynamics e -> Dynamics e) -> Dynamics e
2          -> Dynamics (Dynamics e)
3  memo tr (Dynamics m) =
4    Dynamics $ \ps ->
5    do let sl = solver ps
6           iv = interval ps
7           (stl, stu) = stageBnds sl
8           (nl, nu)   = iterationBnds iv (dt sl)
9       arr   <- newMemoUArray_ ((stl, nl), (stu, nu))
10      nref  <- newIORef 0
11      stref <- newIORef 0
12      let r ps =
13            do let sl  = solver ps
14                   iv  = interval ps
15                   n   = iteration ps
16                   st  = stage sl
17                   stu = stageHiBnd sl
18                   loop n' st' =
19                     if (n' > n) || ((n' == n) && (st' > st))
20                     then
21                       readArray arr (st, n)
22                     else
23                       let ps' = ps { time = iterToTime iv sl n' st',
24                                      iteration = n',
25                                      solver = sl { stage = st' }}
26                       in do a <- m ps'
27                             a `seq` writeArray arr (st', n') a
28                             if st' >= stu
29                               then do writeIORef stref 0
30                                       writeIORef nref (n' + 1)
31                                       loop (n' + 1) 0
32                               else do writeIORef stref (st' + 1)
33                                       loop n' (st' + 1)
34               n'  <- readIORef nref
35               st' <- readIORef stref
36               loop n' st'
37      return $ tr $ Dynamics r
```

The function starts by getting how many iterations will occur in the simulation, as well as how many stages the chosen method uses (lines 5 to 8). This is used to pre-allocate the minimum amount of memory required for the execution (line 9). This mutable array is two-dimensional and can be viewed as a table in which the number of iterations and stages determine the number of rows and columns. Pointers to iterate accross the table are declared as *nref* and *stref* (lines 10 and 11), to read iteration and stage values respectively.

The code block from line 12 to line 36 delimit a procedure or computation that will only be used when needed, and it is being called at the end of the *memo* function (line 37).

The next step is to follow the exection of this internal function. From line 13 to line 17, auxiliar "variables", i.e., labels to read information, are created to facilitate manipulation of the solver (`sl`), interval (`iv`), current iteration (`n`), current stage (`st`) and the final stage used in a solver step (`stu`). The definition of *loop*, which starts at line 18 and closes at line 33, uses all the previously created labels. The conditional block (line 19 to 33) will store in the pre-allocated memory region the computed values and, because they are stored in a **sequential** way, the stop condition of the loop is one of the following: the iteration counter of the loop (`n'`) surpassed the current iteration **or** the iteration counter matches the current iteration **and** the stage counter (`st'`) reached the ceiling of stages of used solver method (line 19). When the loop stops, it **reads** from the allocated array the value of interest (line 21), given that it is guaranteed that is already in memory. If this condition is not true, it means that further iterations in the loop need to occur in one of the two axis, iteration or stage.

The first step towards that goal is to save the value of the current iteration and stage into memory. The dynamic computation `m`, received as a dependency in line 3, is used to compute a new result with the current counters for iteration and stage (lines 23 to 26). Then, this new value is written into the array (line 27). The condition in line 28 checks if the current stage already achieved its maximum possible value. In that case, the counters for stage and iteration counters will be refreshed to the first stage (line 29) of the next iteration (line 30) respectively, and the loop should continue (line 31). Otherwise, we need to advance to the next stage within the same iteration and an updated stage (line 32). The loop should continue with the same iteration counter but with the stage counter incremented (lines 32 and 33).

Lines 34 to 36 are the trigger to the beginning of the loop, with *nref* and *stref* being read. These values set the initial values for the counters used in the *loop* function, and both of their values start at zero (lines 10 and 11). All computations related to the *loop* function will only be called when the r function is called. Further, all of these impure computations (lines 12 to 36) compose the definition of r (line 12), which is being returned in line 37 combined with the interpolation function tr and being wrapped with an extra `Dynamics` shell via the *return* function (provided by the `Monad` typeclass).

With this function on-hand, it remains to couple it to the `Integrator` type, meaning that **all** integrator functions need to be aware of this new caching strategy. First and foremost, a pointer to this memory region needs to be added to the integrator type itself:

```
1 data Integrator = Integrator { initial     :: Dynamics Double,
2                                cache       :: IORef (Dynamics Double),
3                                computation :: IORef (Dynamics Double)
4                              }
```

Next, two other functions need to be adapted: *newInteg* and *readInteg*. In the former function, the new pointer will be used, and it points to the region where the mutable array will be allocated. In the latter, instead of reading from the computation itself, the read-only pointer will be looking at the **cached** version. These differences will be illustrated by using the same integrator and state variables used in the Lorenz's Attractor example, detailed in chapter 4, *Execution Walkthrough*.

The main difference in the updated version of the *newInteg* function is the inclusion of the new pointer that reads the cached memory (lines 4 to 7). The pointer `computation`, which will be changed by *diffInteg* in a model to the differential equation, is being read in lines 8 to 10 and piped with interpolation and memoization in line 11. This approach maintains the interpolation, justified in the previous chapter, and adds the aforementioned caching strategy. Finally, the final result is written in the memory region pointed by the caching pointer (line 12).

Figure 6.2 shows that the updated version of the *newInteg* function is similar to the previous implementation. The new field, `cached`, is a pointer that refers to `readComp` — the result of memoization (`memo`), interpolation (`interpolate`) and the value obtained by the region pointed by the `computation` pointer. Given a parametric record `ps`, `readComp` gives this record to the dynamic value stored in the region pointed by `computation`. This result is then interpolated via the `interpolate` block and it is used as a dependency for the `memo` block.

The modifications in the *readInteg* function are being portrayed in Figure 6.3. As described earlier, the change is minor: instead of reading from the region pointed by the `computation` pointer, this function will read the value contained in the region pointed by the `cache` pointer (line 4). This means that the same `readComp`, described in the new *newInteg* function, will receive a given `ps`. It is worth noticing that, just like with the *newInteg* function, this cache pointer indirectly interacts with the same memory location pointed by the `computation` pointer in the integrator (Figure 6.3).

```haskell
newInteg :: Dynamics Double -> Dynamics Integrator
newInteg i =
  do comp <- liftIO $ newIORef $ initialize i
     cachedComp <- liftIO $ newIORef $ initialize i
     let integ = Integrator { initial      = i,
                              cache        = cachedComp,
                              computation  = comp }
         readComp = Dynamics $ \ps ->
                  do (Dynamics m) <- readIORef (computation integ)
                     m ps
     interpCached <- memo interpolate readComp
     liftIO $ writeIORef (cache integ) interpCached
     return integ
```

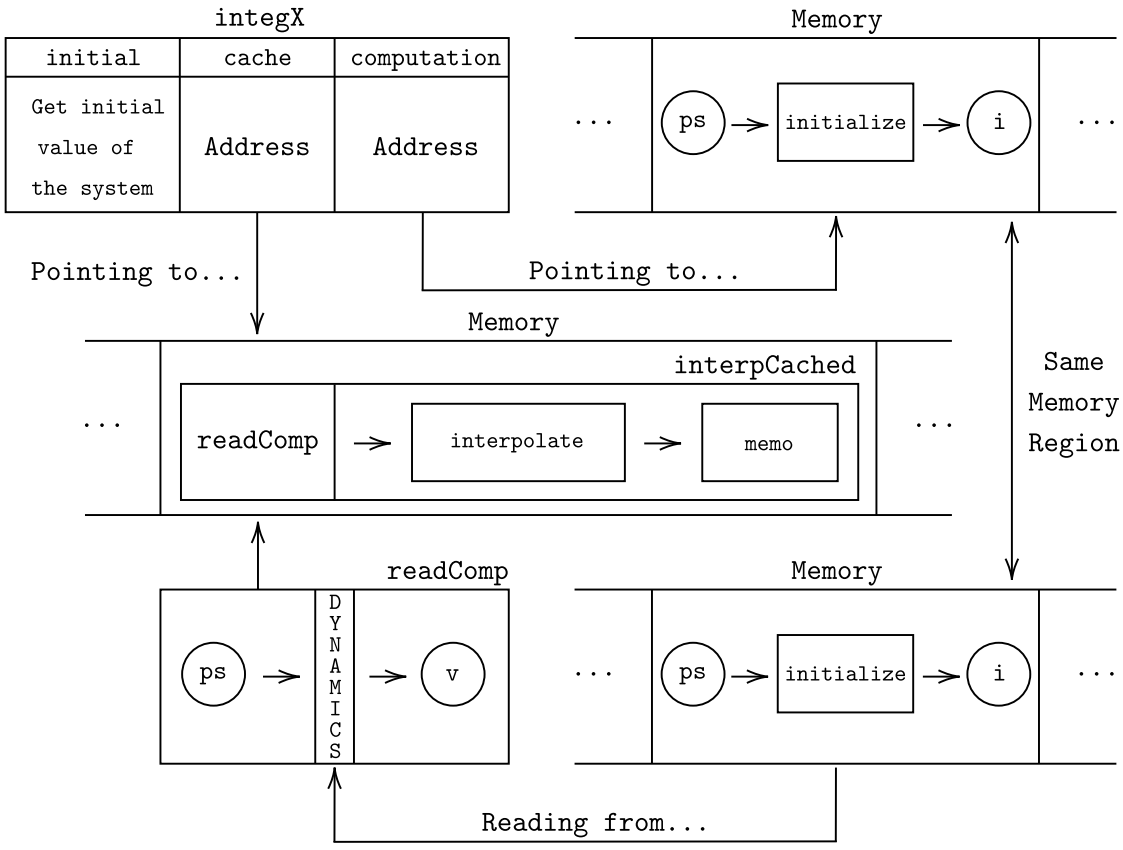


Figure 6.2: The new *newInteg* function relies on interpolation composed with memoization. Also, this combination **produces** results from the computation located in a different memory region, the one pointed by the `computation` pointer in the integrator.

```haskell
readInteg :: Integrator -> Dynamics Double
readInteg integ =
  Dynamics $ \ps ->
  do (Dynamics m) <- readIORef (cache integ)
     m ps
```

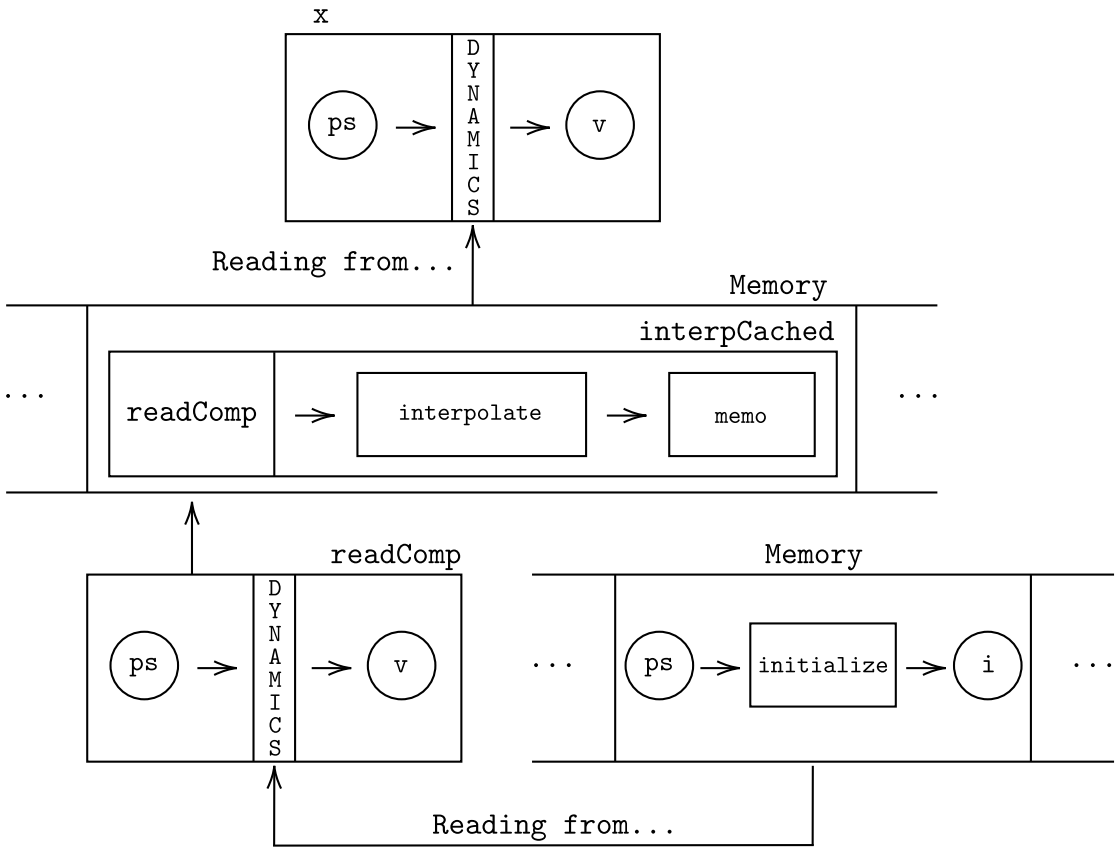


Figure 6.3: The function **reads** information from the caching pointer, rather than the pointer where the solvers compute the results.

Lastly, Figure 6.4 depicts the new version of the *diffInteg* function. Further, the tweaks in this function are minor, just as with the *readInteg* function. Previously, the `whatToDo` label, used as a dependency in the solver methods, was being made by reading the content in the region pointed by the `computation` pointer. Now, this dependency reads the region related to the caching methodology via reading the `cache` pointer.

```haskell
diffInteg :: Integrator -> Dynamics Double -> Dynamics ()
diffInteg integ diff =
  do let z = Dynamics $ \ps ->
           do whatToDo <- readIORef (cache integ)
              let i = initial integ
              case method (solver ps) of
                Euler -> integEuler diff i whatToDo ps
                RungeKutta2 -> integRK2 diff i whatToDo ps
                RungeKutta4 -> integRK4 diff i whatToDo ps
     liftIO $ writeIORef (computation integ) z
```

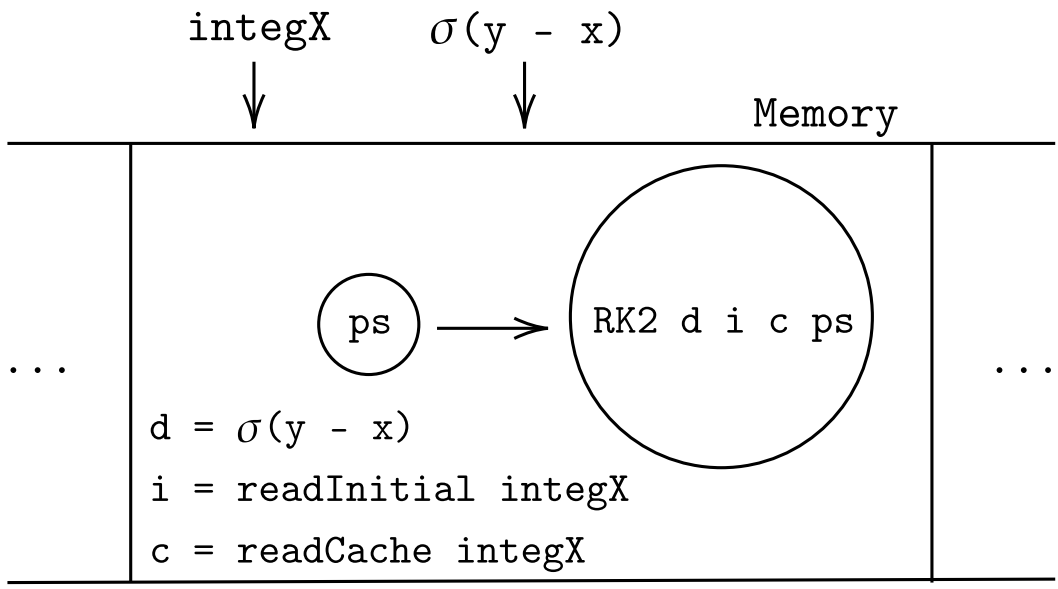


Figure 6.4: The new *diffInteg* function gives to the solver functions access to the region with the cached data.

The solver functions, *integEuler*, *integRK2* and *integRK4*, always need to calculate the value of the previous iteration. By giving them access to the cached region of the simulation, instead of starting a recursive chain of stack calls, the previous computation will be handled immediately. This is the key to cut orders of magnitude in execution time during simulation.

6.4 A change in Perspective

Before the implementation of the described caching strategy, **all** the solver methods rely on implicit recursion to get the previous iteration value. Thus, performance was degraded due to this potentially long stack call. After caching, this mechanism is not only faster, but it **completely** changes how the solvers will get these past values.

For instance, when using the function *runDynamicsFinal* as the driver, the simulation will start by the last iteration. Without caching, the solver would go from the current iteration to the previous ones, until it reaches the base case with the initial condition and starts backtracking the recursive calls to compute the result of the final iteration. On the other hand, with the caching strategy, the *memo* function goes in the **opposite** direction: it starts from the beginning, with the counters at zero, and then incrementally proceeds until it reaches the desired iteration.

Figure 6.5 depicts this stark difference in approach when using memoization in `Rivika`. Instead of iterating through all iterations two times, one backtracking until the base case and another one to accumulate all computed values, the new version starts from the base case, i.e., at iteration 0, and stops when achieves the desired iteration, saving all the values along the way.

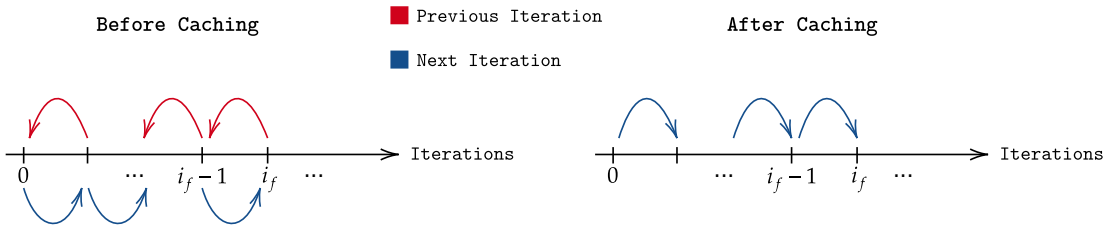


Figure 6.5: Caching changes the direction of walking through the iteration axis. It also removes an entire pass through the previous iterations.

6.5 Tweak III: Model and Driver

The memoization added to `Rivika` needs a second tweak, related to the executable models established in chapter 4. The code bellow is the same example model used in that chapter:

```
1 exampleModel :: Model Vector
2 exampleModel =
3   do integX <- newInteg 1
4      integY <- newInteg 1
5      let x = readInteg integX
6          y = readInteg integY
7      diffInteg integX (x * y)
8      diffInteg integY (y + t)
9      sequence [x, y]
```

The caching strategy assumes that the created mutable array will be available for the entire simulation. However, the proposed models will **always** discard the table created

by the *newInteg* function due to the garbage collector [2], after the *sequence* function. Even worse, the table will be created again each time the model is being called and a parametric record is being provided, which happens when using the driver. Thus, the proposed solution to address this problem is to update the `Model` alias to a **function** of the model. This can be achieved by **wrapping** the state vector with a the `Dynamics` type, i.e., wrapping the model using the function *pure* or *return*. In this manner, the computation will be "placed" as a side effect of the `IO` monad and Haskell's memory management system will not remove the table used for caching, in the first computation. So, the following code is the new type alias, alongside the previous example model using the *return* function:

```
type Model a = Dynamics (Dynamics a)

exampleModel :: Model Vector
exampleModel =
  do integX <- newInteg 1
     integY <- newInteg 1
     let x = readInteg integX
         y = readInteg integY
     diffInteg integX (x * y)
     diffInteg integY (y + t)
     return $ sequence [x, y]
```

Due to the new type signature, this change implies changing the driver, i.e., modify the function *runDynamics* (the changes are analogus to the *runDynamicsFinal* function variant). Further, a new auxiliary function was created, *subRunDynamics*, to separate the environment into two functions. The *runDynamics* will execute the mapping with the function *parameterise* and the auxiliary function will address the need for interpolation.

```
runDynamics :: Model a -> Interval -> Solver -> IO [a]
runDynamics (Dynamics m) iv sl =
  do d <- m Parameters { interval = iv,
                         time = startTime iv,
                         iteration = 0,
                         solver = sl { stage = 0 }}
     sequence $ subRunDynamics d iv sl

subRunDynamics :: Dynamics a -> Interval -> Solver -> [IO a]
subRunDynamics (Dynamics m) iv sl =
  do let (nl, nu) = iterationBnds iv (dt sl)
         parameterise n = Parameters { interval = iv,
```

[2] Garbage Collector wiki page.

```
12                                      time = iterToTime iv sl n 0,
13                                      iteration = n,
14                                      solver = sl { stage = 0 }}
15          ps = Parameters { interval = iv,
16                            time = stopTime iv,
17                            iteration = nu,
18                            solver = sl { stage = -1}}
19      if (iterToTime iv sl nu 0) - (stopTime iv) < 0.00001
20      then map (m . parameterise) [nl .. nu]
21      else (init $ map (m . parameterise) [nl .. nu]) ++ [m ps]
```

The main change is the division of the driver into two: one dedicated to "initiate" the simulation environment providing an initial record of the type `Parameters` (lines 3 to 6), and an auxiliary function doing the mapping to the iteration axis (lines 10 to 14, 20 and 21), as well as checking for interpolation (lines 15 to 19). Thus, this is the final implementation of the driver in `Rivika`.

6.6 Results with Caching

The following table (Table 6.3) shows the same Lorenz's Attractor example used in the first section, but with the preceding tweaks in the `Integrator` type and the integrator functions. These modifications allows better and more complicated models to be simulated. For instance, the Lorenz example with a variety of total number of iterations can be checked in Table 6.4 and in Figure 6.6.

Total of Iterations	Previous Execution Time (seconds)	Execution Time (seconds)	Consumed Memory (MB)
1	0.01	0.00	0.5
2	0.01	0.00	0.6
3	0.08	0.00	0.7
4	0.79	0.00	0.8
5	10.06	0.00	0.9
6	140.95	0.01	1.1
7	1798.16	0.01	1.2
8	23801.51	0.00	1.3

Table 6.3: These values were obtained using the same hardware. It shows that the caching strategy drastically improves `Rivika`'s performance. Again, the concrete memory values obtained from GHC should be considered as just an indicative of improvement due to the garbage collector interference.

Total of Iterations	Execution Time (seconds)	Consumed Memory (MB)
100	0.02	1.5
1K	0.04	11.8
10K	0.30	114.7
100K	3.28	1143.3
1M	29.91	11429.3
10M	307.66	114289.7
100M	3205.06	1142893.0

Table 6.4: These values were obtained using the same hardware. More complicated simulations can be done with `Rivika` after adding memoization.

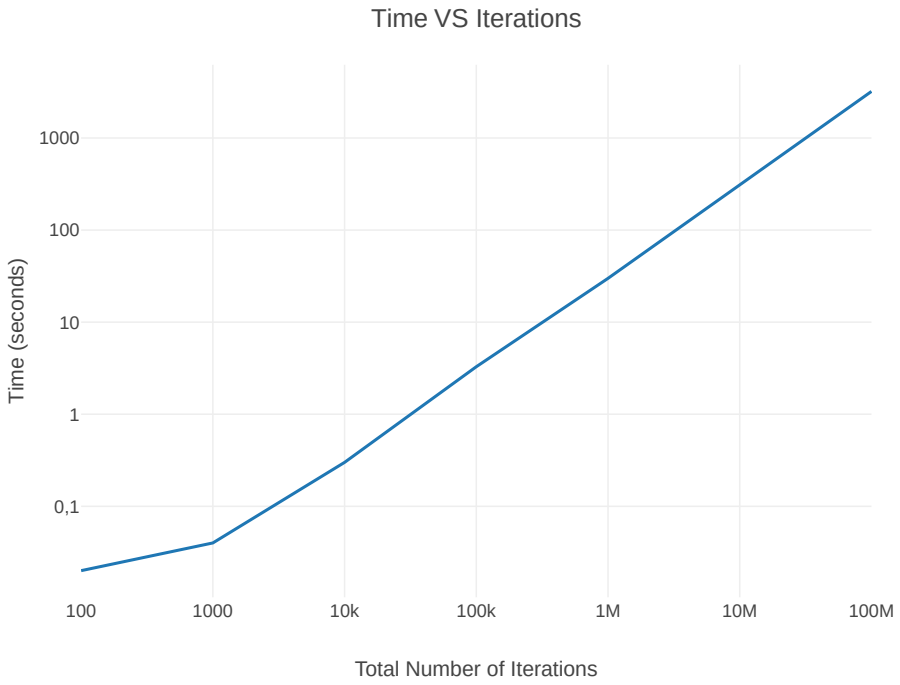


Figure 6.6: By using a logarithmic scale, we can see that the final implementation is performant with more than 100 million iterations in the simulation.

This summarizes the chapter, closing the arc about addressing drawbacks. The project is currently capable of executing interpolation as well as applying memoization to speed up results. These two solutions, detailed in chapter 5 and 6, adds practicality to `Rivika` as well as makes it more competitive. The final chapter, *Conclusion*, will conclude this work, pointing out limitations of the project, as well as future improvements and final thoughts about the project.

Chapter 7

Conclusion

Chapters 2 and 3 explained the relationship between software, FF-GPAC and the mathematical world of differential equations. As a follow-up, chapter 4 raised intuition and practical understanding of `Rivika` via a detailed walkthrough of an example. Chapters 5 and 6 identified some problems with the current implementation, such as lack of performance and the discrete time issue, and addressed both problems via caching and interpolation. This chapter, *Conclusion*, draws limitations, future improvements that can bring `Rivika` to a higher level of abstraction and some final conclusions about the project.

7.1 Limitations

One of the main concerns is the **correctness** of `Rivika` between its specification and its final implementation, i.e., refinement. Shannon's GPAC concept acted as the specification of the project, whilst the proposed software attempted to implement it. The criteria used to verify that the software fulfilled its goal were by using it for simulation and via code inspection, both of which are based on human analysis. This connection, however, was **not** formally verified. Thus, `Rivika` can be a threat to validity if a future formal verification comes up and checks that the parallel between those two can't be guaranteed.

Further, there is also an issue to regards to **validation**. In order to know that the mathematical description of the problem is being correctly mapped onto a model representation some formal work needs to be done. This was not explored, and it was considered out of the scope of the project. However, such aspect dictates if the specification for further implementation is actually correct and describes its mathematical counterpart. So, checking for validation is just as important as verifying refinement.

This lack of formalism extends to the typeclasses as well. The programming language of choice, Haskell, does **not** provide any proofs that the created types actually follow the

typeclasses' properties, even if the requested functions type check. This burden is on the developer to manually write down such proofs, a non-explored aspect of this work.

As explained in chapters 1 and 2, there are some extensions that increase the capabilities of Shannon's original GPAC model. One of these extensions, FF-GPAC, was the one chosen to be modeled via software. However, there are other extensions that not only expand the types of functions that can be modeled, e.g., hypertranscendental functions, but also explore new properties, such as Turing universitality [11, 12]. The proposed software didn't touch on those enhancements and restricted the set of functions to only algebraic functions.

Finally, there is the language itself, Haskell. Although Haskell's type system allowed a great mapping between the numerical methods and its nuances to created types, its simplicity started to fall apart when impurity came into picture. The side effect overhead makes `Rivika` hard to reason about, especially for newcomers that intent to expand the software's functionalities.

7.2 Future Improvements

There are solutions to mitigate the problems presented in the previous section. First, to address refinement, the simulation could be assessed by continuous domain specialists. Also, proof-assistant tools, such as COQ and PVS, could be used to re-write `Rivika` with a proper formal basis, hence establishing a solid map between the mathematical description, specification and implementation. Further, the same tools can leverage the correctness of the typeclasses' implementation, via demonstrating that it assures the axioms and properties demanded by each typeclass. More recent extensions of GPAC should also be explored to simulate an even broader set of functions present in the continuous time domain.

In regards to numerical methods, one of the immediate improvements would be to use **adaptive** size for the solver time step that **change dynamically** in run time. This strategy controls the errors accumulated when using the derivative by adapting the size of the time step. Hence, it starts backtracking previous steps with smaller time steps until some error threshold is satisfied, thus providing finer and granular control to the numerical methods, coping with approximation errors due to larger time steps.

In terms of the used technology, some ideas come to mind related to abstracting out duplicated **patterns** across the code base. The proposed software used a mix of high level abstractions, such as algebraic types and typeclasses, with some low level abstractions, e.g., explicit memory manipulation. An immediate improvement related to this topic would be to abstract the **stage** information inside the solver using a sum type, `Stage`,

thus removing the use of negative and positive numbers as the trigger for interpolation. On the same line of leveraging abstractions, another major improvement would be to make it entirely **pure**, meaning that all the necessary side effects would be handled **only** by high-level concepts internally, hence decreasing complexity of the software. For instance, the memory allocated via `IORef` [1] acts as a **state** of the numerical solver; this could be refactored to use the `ST` monad [2] This monad deals with state management by itself, removing this weight from the developer.

Further, with the removal of `IORef` type from the project, the next step would be to change the `Dynamics` type to not include in its definition the `IO` monad. As we saw in chapters 2 and 3, this type is heavily coupled to functions that deal with `IORef` type, such as providing a pointer to a memory region. Moreover, because `IO` was involved, the typeclass `MonadIO` became a requirement, given that we need to transition from it to the `Dynamics` monad in a few situations, like in the *newInteg* function. As a middle step before achieving an implementation based on the `ST` monad, **monad transformers** [3] provides a more elegant alternative to go back and forth between monads, removing the need for the `MonadIO` typeclass.

The `Dynamics` type, which is a function with the signature `Parameters -> IO a`, resembles the `Reader` monad [4], a monad that captures the notion of functions. Across the implementation, a lot of intermediate dynamic computations are being created and in the majority of these steps the same record of `Parameters` is being applied in sequence, creating a chain of functions that are passing the same parameter to one another. By using the `Reader` monad, this pattern could be abstracted out from the program. This idea, when combined with the `ST` monad initiative, indicates that the `RWS` monad [5], a monad that combines the monads `Reader`, `Writer` and `ST`, may be the final goal for a completely pure but effective solution.

Also, there's GPAC and its mapping to Haskell features. As explained previously, some basic units of GPAC are being modeled by the `Num` typeclass, present in Haskell's `Prelude` module. By using more specific and customized numerical typeclasses [6], it might be possible to better express these basic units and take advantage of better performance and convenience that these alternatives provide.

Finally, there's the `MonadFix` typeclass [13, 14, 15] [7] [8]; an implemented typeclass

[1] `IORef` hackage documentation.
[2] `ST` Monad wiki page.
[3] Monad Transformers wiki page.
[4] `Reader` Monad hackage documentation.
[5] `RWS` Monad hackage documentation.
[6] Examples of alternative preludes.
[7] `MonadFix` Monad hackage documentation.
[8] `MonadFix` Monad wiki page.

used in more recent versions of `Aivika`. This typeclass uses the mathematical definition of the *fixed-point* concept to compute monadic operations, i.e., it makes it possible to compute the **fix point** of a computation while being wrapped in a monad, thus being useful for creating loopbacks [9] within the monad. As the final result, this typeclass abstracts out the `Integrator` type, meaning that the manipulation of the integrator is no longer maintained by the developer. This shrink in the DSL removes the similarities of the implementation with the GPAC model in some degree, given that the integrator is now implicit. The code below is the same Lorenz Attractor example previously used, but written with this improved version. The main differences are: the absence of the integrator explicitly, the existence of another type that encapsulates the `Dynamics` type, so-called `Simulation`, and the use of `mdo-notation`, also known as *recursive do-notation* [10], rather than `do-notation`:

```
1 lorenzModel :: Simulation [IO Vector]
2 lorenzModel =
3   mdo x <- integ (sigma * (y - x)) 1.0
4       y <- integ (x * (rho - z) - y) 1.0
5       z <- integ (x * y - beta * z) 1.0
6       let sigma = 10.0
7           rho = 28.0
8           beta = 8.0 / 3.0
9       runDynamicsInIntegTimes $ sequence [x, y, z]
```

[9] `MonadFix` Monad example of use case.
[10] Recursive `do-notation` GHC documentation.

7.3 Final Thoughts

When Shannon proposed a formal foundation for the Differential Analyzer [6], mathematical abstractions were leveraged to model continuous time. However, after the transistor era, a new set of concepts that lack this formal basis was developed, and some of which crippled our capacity of simulating reality. Later, the need for some formalism made a comeback for modeling physical phenomena with abstractions that take *time* into consideration. Models of computation [3, 4, 7, 8] and the ForSyDe framework [9, 10] are examples of this change in direction. Nevertheless, Shannon's original idea is now being discussed again with some improvements [5, 11, 12] and being transposed to high level programming languages in the hybrid system domain [1].

The `Rivika` EDSL [11] follows this path of bringing CPS simulation to the highest level of abstraction, via the Haskell programming language, but still taking into account a formal background inspired by the GPAC model. The software uses advanced functional programming techniques to solve differential equations, mapping the abstractions to FF-GPAC's analog units. Although still limited by the discrete nature of numerical methods, the solution is performant and accurate enough for studies in the cyber-physical domain.

[11] `Rivika` source code.

References

[1] Medeiros, José E. G. de, George Ungureanu, and Ingo Sander: *An algebra for modeling continuous time systems*. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 861–864, 2018. x, 3, 7, 8, 64

[2] Medeiros, José E. G.: *Unscented transform framework for quantization modeling in data conversion systems*. 2017. xii, 40

[3] Derler, Patricia, Edward A. Lee, and Albert Sangiovanni Vincentelli: *Modeling cyber-physical systems*. Proceedings of the IEEE, 100(1):13–28, 2012. 1, 2, 4, 14, 19, 64

[4] Lee, Edward A.: *Cyber physical systems: Design challenges*. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008. 1, 2, 3, 64

[5] Graça, Daniel and José Costa: *Analog computers and recursive functions over the reals*. Journal of Complexity, 19:644–664, October 2003. 1, 3, 7, 29, 64

[6] Shannon, Claude E: *Mathematical theory of the differential analyzer*. Journal of Mathematics and Physics, 20(1-4):337–354, 1941. 1, 3, 6, 7, 64

[7] Lee, Edward A. and Alberto L. Sangiovanni-Vincentelli: *Component-based design for the future*. In *2011 Design, Automation Test in Europe*, pages 1–5, 2011. 1, 2, 64

[8] Lee, E.A. and A. Sangiovanni-Vincentelli: *A framework for comparing models of computation*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 17(12):1217–1229, 1998. 2, 14, 64

[9] Sander, Ingo, Axel Jantsch, and Seyed Hosein Attarzadeh-Niaki: *ForSyDe: System Design Using a Functional Language and Models of Computation*, pages 1–42. Springer Netherlands, Dordrecht, 2017, ISBN 978-94-017-7358-4. `https://doi.org/10.1007/978-94-017-7358-4_5-1`. 2, 4, 64

[10] Attarzadeh-Niaki, Seyed Hosein and Ingo Sander: *Heterogeneous co-simulation for embedded and cyber-physical systems design*. SIMULATION, 96(9):753–765, 2020. `https://doi.org/10.1177/0037549720921945`. 2, 4, 64

[11] Graça, Daniel: *Some recent developments on shannon's general purpose analog computer*. Math. Log. Q., 50:473–485, September 2004. 3, 6, 7, 24, 61, 64

[12] Bournez, Olivier, Daniel Graça, and Amaury Pouly: *On the functions generated by the general purpose analog computer*. Information and Computation, 257, January 2016. 3, 6, 61, 64

[13] Erkök, Levent and John Launchbury: *A recursive do for haskell*. Proceedings of the 2002 ACM SIGPLAN Haskell Workshop, September 2002. 62

[14] Erkök, Levent and John Launchbury: *Recursive monadic bindings*. Volume 35, pages 174–185, September 2000. 62

[15] Erkök, Levent and John Launchbury: *A recursive do for haskell: Design and implementation*. January 2000. 62