University of Brasília

Computer Science Department

FFACT: A Fix-based Domain-Specific Language based on a Functional Algebra for Continuous Time Modeling

Eduardo L. Rocha

Dissertation submitted in partial fullfilment of
the requirements to the Master's Degree in Informatics

Advisor
Prof. Eduardo Peixoto

Co-advisor
Prof. José Edil Guimarães

Brasília
2025

University of Brasília

Computer Science Department

FFACT: A Fix-based Domain-Specific Language based on a Functional Algebra for Continuous Time Modeling

Eduardo L. Rocha

Dissertation submitted in partial fullfilment of
the requirements to the Master's Degree in Informatics

Prof. Eduardo Peixoto (Advisor)
CIC/UnB

Prof. Rodrigo Bonifácio de Almeida
CIC/UnB

Prof. Denis Loubach
ITA

Prof. Rodrigo Bonifácio de Almeida
Coordinator of Graduate Program in Informatics

Brasília, April 28, 2025

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

— Ayn Rand, *Atlas Shrugged*

Dedicated to

First, I dedicate this milestone to my family. To my sister Alexya Lemos, for the memorable moments of joy and fun – it is always fun to discuss and brainstorm with you insights about whatever we happen to be watching together.

To my father Rodolfo Rocha, for sharing with me his wise and insightful perceptions about life. Your distinct perspectives bring me awareness about any subject that we end up discussing. Not everyone can have the luxury of having civil and calm conversations with whom they may fundamentally disagree. My dad gave me my first opportunities of this kind and I'm truly grateful to him for that.

And foremost, to my mother Dania Lemos. Since my infancy, my mom has been a fundamental pillar in my life; from my first pronounced word, to my reading and writing, and later in the first years of school; she has most if not all of the merit in regards to those accomplishments. The transition from school to university required way more effort from me and my mom came along for the ride: she became my golden guardian. My countless unhealthy decisions in the pursuit of doing the best in my academic journey were counter-balanced by her always-present joyful smiles and help in whatever she could offer me. Further, she kept celebrating any of my achievements, regardless of how minor or major, to make me believe in a self-esteem based on merit and high effort. Later on, after I started getting into the software industry as a Software Developer, all my complements about my english I redirected to my mom – and that is because she deserves all the attention and merit about this skill that today is a must-have in my life. Without her there to keep me pushing further on learning a foreign language, I would not be where I am today, far from it. Dania is the inspirational example that every person needs: an unstoppable warrior for us to follow; one based on kindness, effort, goodness, and ambition. Let your Will be charged by hers; let your Heart by melted by hers; your paramount goal should be to learn from her and consider what she has to share with you.

Second, I dedicate this work to my close friends – Edil Medeiros, Marcos Magueta, Mário Junior, Neil Mayhew, Elisama Honesko, Gabriela Lul and others. Thank you for all the great conversations, insights, and fun memories.

Acknowledgements

First, I must acknowledge my main advisor, Edil Medeiros. Since graduation, and now in my masters, he trusted that my effort could go on and beyond, surpassing my own expectations and limits, getting out of my comfort zone. All the endless meetings, including on the weekends, filled with thoughtful advice and helpful comments, will be the most remarkable memory of the best mentor I have encountered to this day – a title that Edil got back when I was doing graduation, and he continues to be worthy.

I'm thankful to my Computer Science study group, Dr.Nekoma, for the continued joyful programming practices leveraging functional programming principles, something that is still the core foundation of the present work, even though this paradigm remains uncommon both in the industry of software development industry and in academia.

I'm grateful for the company I'm currently working in, Tontine Trust, where I'm a member of a great team delivering a challenging product to the market. Some of these challenges are being solved in Haskell; a programming language I grew fond of it since my final graduation project was written in it. This work is a continuation of that.

Finally, a special thanks to everybody who took any amount of time to read any draft I had of this dissertation, providing honest feedback.

Abstract

Physical phenomena are difficult to properly model due to their continuous nature. Its paralellism and nuances were a challenge before the transistor, and even after the digital computer it 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) – FACT and its evolution FFACT – written in Haskell that resembles GPAC's concepts. The main goal is to take advantage of high level abtractions, both from the areas of programming and mathematics, 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, fixed-point, fixed-point combinator, monadic recursion

Resumo

Título: FFACT: Uma linguagem de Domínio Específico utilizando ponto fixo baseada em uma Álgebra Funcional para Modelagem de Tempo Contínuo.

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) – FACT e sua evolução FFACT – escrita em Haskell que assemelha-se aos conceitos do GPAC. O principal objetivo é aproveitar de abstrações de mais alto nível, tanto da área da programação quanto da matemática, 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, ponto fixo, recursão monádica

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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 continuing to be a challenge. Cyber-physical systems (CPS) — the integration of computers and physical processes [8, 9] — tackles this problem by attempting to include into the *semantics* of computing the physical notion of *time* [9, 10, 11, 12, 13, 14], i.e., treating time as a measurement of *correctness*, not *performance* [8] nor just an accident of implementation [9]. 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* [9].

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 [15]. 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 [16, 17, 15], 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 [9, 18]. 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 [18], encouraging ad-hoc design practices, thus adding inertia into a dangerous legacy we want to avoid [19]. With this in mind, Lee advocated

that leveraging better formal abstractions is the paramount goal to advance continuous time modeling [9, 18]. 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 [8] on how engineering artifacts should behave [10]. 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 [8, 18]. Also, Sangiovanni and Lee [20] 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 [8, 18]. 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* [21, 22, 23] — a relationship between a *tag*, generally used to order events, and an output value.

Ingo et al. went even further [24] by presenting 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 to be implemented later in a target programming language or hardware platform [24]. Afterward, Seyed-Hosein and Ingo [13] 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 [13]. Simulink being the standard tool for modeling means that, despite all the effort into utilizing a formal approach to model-based design, there is still room for improvement.

1.1 Contribution

The aforementioned works — the formal notion of MoCs, the ForSyDe framework and its interaction with modeling-related tools like Simulink — comprise the domain of model-based design or *model-based engineering*. Furthermore, the main goal of this work is to contribute to this sub-area of CPS by creating a domain-specific language tool (DSL) for

[1] Simulink documentation.

simulating continuous-time systems that addresses inspired by a mathematical model of computation. Thus, this tool will serve as the foundation to deal with the incompatibility of the mentioned sets of abstractions [9] — the discreteness of digital computers with the continuous nature of physical phenomena.

The proposed DSL has three special properties of interest:

1. it needs to have well-defined *operational* semantics, as well as being a piece of *executable* software;
2. it needs to be related or inspired by a *formal* model, moving past *ad-hoc* implementations;
3. it should be *concise*; its lack of noise will bring familiarity to the *system's designer* — the pilot of the DSL which strives to execute a given specification or golden model.

1.1.1 Executable Simulation

By making an executable software capable of running continuous time simulations, *verification via simulation* will be available — 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.

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, alongside code refactoring using other Haskell's abstractions. So, this reduced and refactored version of `Aivika`, so-called `FACT` [3], will be a Haskell Embedded Domain-Specific Language (HEDSL) within the model-based engineering domain. The built DSL will explore Haskell's specific features and details, such as

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

the type system and typeclasses, to solve differential equations. Figure 1.1 shows a side-by-side comparison between the original implementation of Lorenz Attractor in FACT, presented in [2], and its final form, so-called FFACT, for the same physical system.

```
-- FACT
lorenzModel = do
  integX <- createInteg 1.0
  integY <- createInteg 1.0
  integZ <- createInteg 1.0
  let x = readInteg integX
      y = readInteg integY
      z = readInteg integZ
      sigma = 10.0
      rho = 28.0
      beta = 8.0 / 3.0
  updateInteg integX (sigma * (y - x))
  updateInteg integY (x * (rho - z) - y)
  updateInteg integZ (x * y - beta * z)
  return $ sequence [x, y, z]
```

```
-- FFACT
lorenzModel =
  mdo x <- integ (sigma * (y - x)) 1.0
      y <- integ (x * (rho - z) - y) 1.0
      z <- integ (x * y - beta * z) 1.0
      let sigma = 10.0
          rho = 28.0
          beta = 8.0 / 3.0
      return $ sequence [x, y, z]
```

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

1.1.2 GPAC: inspiration for a Formal Model

This work and its artifact (a functional DSL to execute simulations) is a direct continuation of the work made by Edil Medeiros et al. [1]. Their work tackled CPS-modeling via a DSL, which used the general-purpose analog computer (GPAC), proposed by Shannon [16] in 1941, as a guideline for a solid and formal foundation.

Hence, the tool we propose is also inspired by GPAC. 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 [25]. 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 [16]. 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 [15], 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 [15] all the way to make GPAC capable of expressing generable functions, Turing universality and hypertranscendental functions [25, 26]. Furthermore, although the analog computer has been forgotten in favor of its digital counterpart [15], recent

studies in the development of hybrid systems [1] brought GPAC back to the spotlight in the CPS domain.

During the design of the DSL, parallels will establish some resemblance between GPAC's constructs and the implementation. With this strategy, all the mathematical formalism leveraged for analog computers will drive the implementation in the digital computer. However, it is worth noting that we do not formally prove this mapping holds using dedicated tools, such as proof assistants or dependently-typed prograamming languages. GPAC serves as an initial specification in which the generated artifact (executable models for simulation) attempts to follow. Although outside of the scope of this work, this pursue for a formal foundation can be developed in the future.

With that in mind, the HEDSL will strive to translate GPAC's original set of black boxes to some executable software leveraging mathematical constructs to simplify its usability. The programming language of choice was *Haskell* — a well known language in the functional paradigm (FP). The recognition that such paradigm provides better well-defined, mathematical and rigourous abstractions has been proposed by Backus [27] in his Turing Award lecture; where he argued that FP is the path to liberate computing from the limitations of the *von Neumann style* when thinking about systems. Thus, continuous time being specified in mathematical terms, 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 or better described in FP. Lee describes a lot of properties [8] that matches this programming paradigm almost perfectly:

1. Prevent misconnected MoCs by using great interfaces in between ⇒ Such interfaces can be built using Haskell's *strong type system*
2. Enable composition of MoCs ⇒ Composition is a first-class feature in functional programming languages
3. 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
4. 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
5. 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

In terms of the DSL being *embedded* in Haskell, this approach of making specialized programming languages, or *vocabularies*, within consistent and well-defined host programming languages, has already proven to be valuable, as noted by Landin [28]. Further, this strategy is already being used in the CPS domain in some degree, as showed by the ForSyDe framework [24, 13].

1.1.3 Expressiveness and Conciseness

This dissertation being a step in a broader story, started in 2018 by Medeiros et al. [1], one of the goals is to improve on the identified limitations in the first proposed DSL, such as the lack or high difficulty on expressing systems via explicit signal manipulation for arbitrary closed feedback loops. Later publications addressed this issue [2, 3] whilst introducing or keeping known problems, such as noisy and overloaded notation when using the DSL (Figure 1.2)– a consequence of an *abstraction leaking* – and not being able to model hybrid systems; systems with changes in continuous behavior based on discrete events.

```
sineModel = intCT rk4 0 0 p1
  where
    p1 = (constCT (-1) *** idCT) >>> multCT >>> integ
    integ = intCT rk4 0 1 loopBreaker
    loopBreaker = (idCT *** constCT 0) >>> adderCT
```

$$\begin{cases} \dot{y}(t) = z(t) \\ \dot{z}(t) = -y(t) \end{cases}$$

```
sineModel =
  do integY <- createInteg 1
     integZ <- createInteg 0
     let y = readInteg integY
         z = readInteg integZ
     updateInteg integY z
     updateInteg integZ (-y)
     return $ sequence [y, z]
```

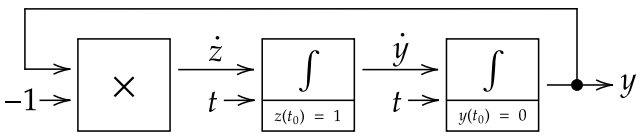


Figure 1.2: Comparison between the original proposed DSL [1] and the first version of `FACT` [2, 3] using the same sine model, alongside its mathematical and GPAC descriptions.

So, to address the aforementioned abstraction leaking and improve the DSL's conciseness, this work uses the *fixed-point combinator*; a mathematical construct. The goal is to make the DSL's machinery hide implementation details noise from the user's perspective, keeping on the surface only the constructs that matter from the designer's point of view. Once the leak is solved, it is expected that the *target audience* — system's designers with less programming experience but familiar with the system's mathematical description —

will be able to leverage the DSL either when improving the system's description, using the DSL as a refinement tool, or as a way to execute an already specified system. Given that the present work, FFACT, being a direct continuation of FACT [2], it is important to highlight that this final property is the main differentiating factor between the two pieces.

When comparing models in `FFACT` to other implementations in other ecosystems and programming languages, `FFACT`'s conciseness brings more familiarity, i.e., one using the HEDSL needs less knowledge about the host programming language, Haskell in our case, *and* one can more easily bridge the gap between a mathematical description of the problem and its analogous written in `FFACT`, due to less syntatical burden and noise from a user's perpective. Examples and comparisons will be depicted in Chapter 7, *Fixing Recursion*, Section 7.5.

1.2 Outline

Chapter 2, *Design Philosophy*, presents the foundation of this work, started in 2018 [1]. Although the artifacts presented in the original work and this work are far apart, the mathematical base is the same. Chapters 3 to 6 describe future improvements made in 2022 [2] and 2023 [3]. These chapters go in detail about the DSL's implementation details, such as the used abstractions, going through executable examples, pointing out and addressing problems in its usability and design. Issues like performance, and continuous time implementation are explained and then addressed. Whilst the implementation of Chapters 2 to 6 were vastly improved during the making of this dissertation, alongside improvements on the writing of their respective chapters, the latest inclusion to this research is concentrated in Chapter 7, *Fixing Recursion*, which dedicates itself to improving an abstraction leak in the most recent published version of the DSL [3]. Those improvements leverage the *fixed point combinator* to eliminate abstraction leaks, thus making the DSL more concise and familiar to a system's designer. These enhacements were submitted and are waiting approval in a related journal [4]. Finally, limitations, future improvements and final thoughts are drawn in Chapter 8, *Conclusion*.

[4] Journal of Functional Programming.

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 and typeclass systems 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 [26]. 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 [16, 25]. 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 [16]. 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 [16]. 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, 16]. 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.

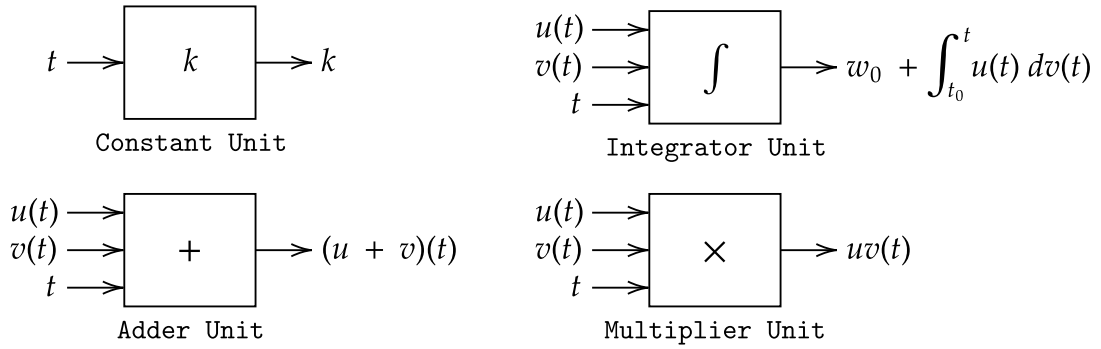


Figure 2.1: The combination of these four basic units compose any GPAC circuit (taken from [1] with permission).

- 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 connected 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, 15, 16]. Daniel's GPAC extension, FF-GPAC [15], added new constraints related to no-feedback GPAC configurations while still using the same four basic units. These structures, so-called *polynomial circuits* [1, 25], 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:

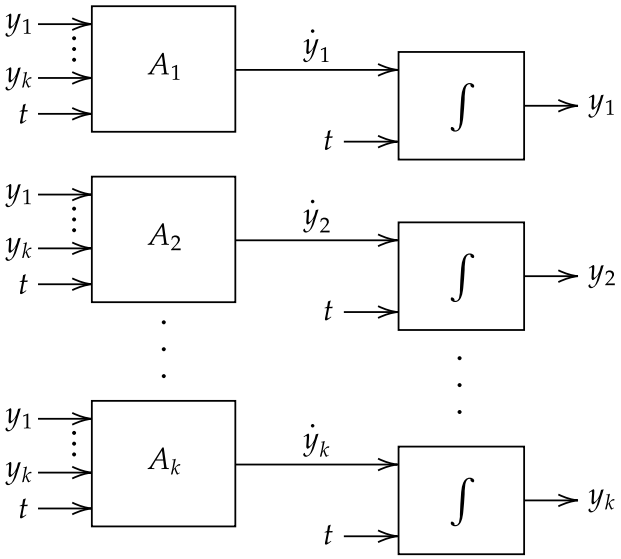


Figure 2.2: Polynomial circuits resembles combinational circuits, in which the circuit respond instantly to changes on its inputs (taken from [1] with permission).

- 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.
- 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.

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*.

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,

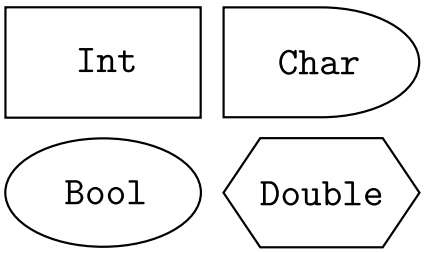


Figure 2.3: Types are not just labels; they enhance the manipulated data with new information. Their difference in shape can work as the interface for the data.

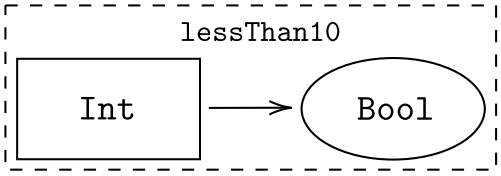


Figure 2.4: Functions' signatures are contracts; they purespecify which shape the input information has as well as which shape the output information will have.

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
```

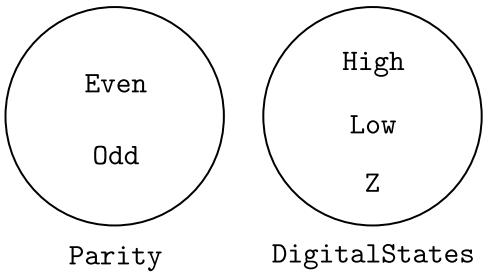


Figure 2.5: Sum types can be understood in terms of sets, in which the members of the set are available candidates for the outer shell type. Parity and possible values in digital states are examples.

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 }
```

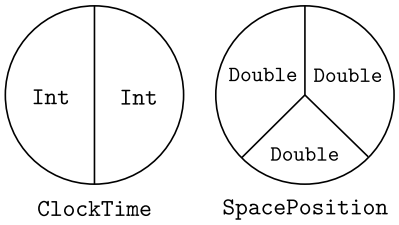


Figure 2.6: Product types are a combination of different sets, where you pick a representative from each one. Digital clocks' time and objects' coordinates in space are common use cases. In Haskell, a product type can be defined using a *record* alongside with the constructor, where the labels for each 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* [29]. 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, although the implementer is not obligated to prove its laws on a language level. 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` typeclass for the presented `ClockTime`, giving it capabilities for sorting instances of such 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 trans-

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

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

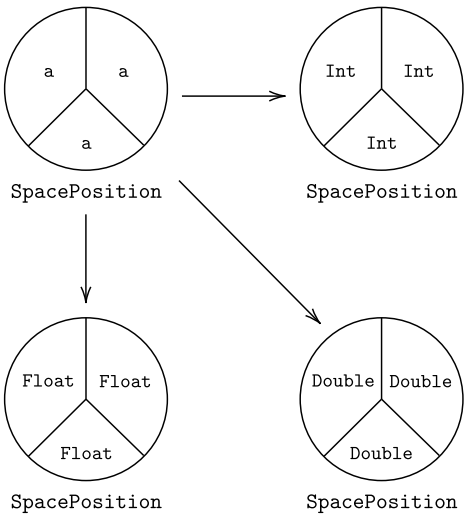


Figure 2.7: Depending on the application, different representations of the same structure need to used due to the domain of interest and/or memory constraints.

```haskell
data ClockTime = Time Int Int

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

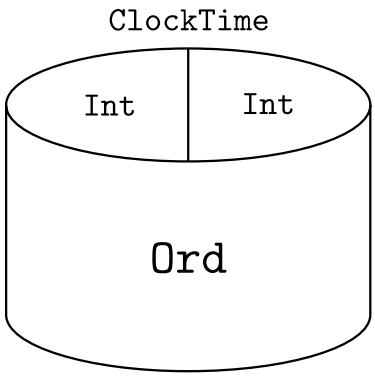


Figure 2.8: The minimum requirement for the `Ord` typeclass is the <= operator, meaning that the functions <, <=, >, >=, `max` and `min` are now unlocked for the type `ClockTime` after the implementation. Typeclasses can be viewed as a third dimension in a type.

formation 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.

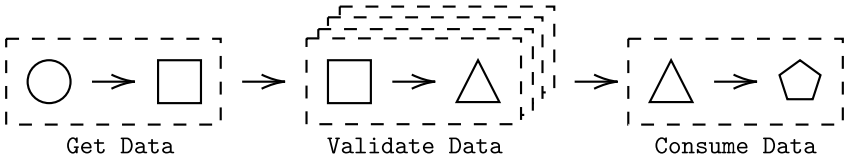


Figure 2.9: Replacements for the validation function within a pipeline like the above are common.

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 instance, 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 `CT` (stands for *continuous machine*) in Figure 2.11 is a first draft of representing the continuous physical dynamics [8] — the evolution of a system state in time:

```
type Time = Double
type Outcome = Double
data CT =
     CT (Time -> Outcome)
```

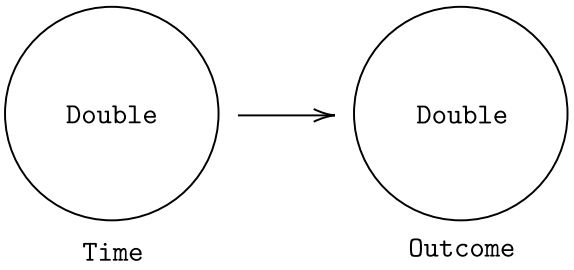


Figure 2.11: In Haskell, the `type` keyword works for alias. The first draft of the `CT` type is a *function*, in which providing a floating point value as time returns another value as outcome.

This type seems to capture the concept, whilst being compatible with the definition of a tagged system presented by Lee and Sangiovanni [20]. 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 `CT` 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 (more about the stage field on a later Chapter). Finally, the `Parameters` type combines everything together, alongside with the current time value as well as its discrete counterpart, iteration.

Further, the new `CT` type can also be parametrically polymorphic, removing the limitation of only using `Double` values as the outcome. Figure 2.14 depicts the final type 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. Below, we have the definition for the `CT` type used in previous work [2]:

```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)
```

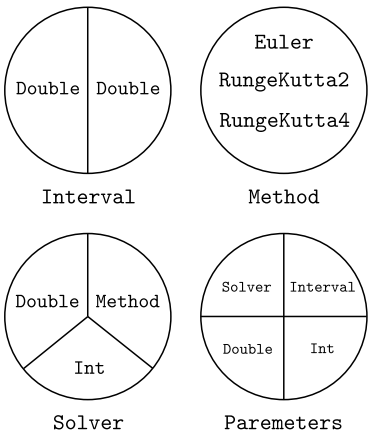


Figure 2.12: The `Parameters` type represents a given moment in time, carrying over all the necessary information to execute a solver step until the time limit is reached. Some useful typeclasses are being derived to these types, given that Haskell is capable of inferring the implementation of typeclasses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

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

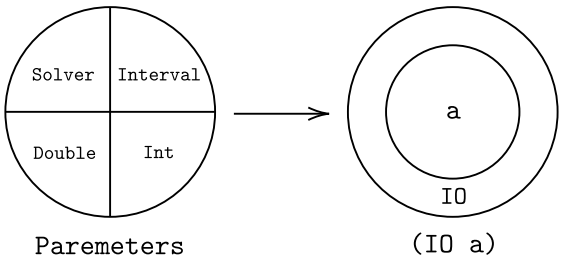


Figure 2.13: The `CT` type is a function of from time related information to an arbitrary potentially effectful outcome value.

In Haskell, however, function types — types that are carrying a function inside — are well-known and identified as an instance of a *reader pattern*. The argument in the function type, in our case `Parameters`, is called a shared *environment* for the computation of the `Reader` [1]. Moreover, because the output of our function type is wrapped inside `IO`, we can leverage another common abstraction in Haskell: *monad transformers*. More about Monads will be explained in later chapters, but for now, it suffices to show that our type `CT` is just a type alias for a combination of the *reader transformer* – a combination of the monads `Reader` with an underlying `IO` monad within:

[1] `Reader` Hackage reference.

```haskell
type CT a = ReaderT Parameters IO a
```

Figure 2.14: The `CT` type can leverage monad transformers in Haskell via `Reader` in combination with `IO`.

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 `CT` type, and will later describe the last core type before explaining the solver execution: the `Integrator` type. These improvements for the `CT` 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 using more advanced Haskell concepts, like typeclasses [1]. It starts by enhancing the previously defined `CT` 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 CT Type

The `CT` 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 `CT` 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 `CT` 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

[1] `Classes in Haskell:` reference.

signature, they share the intuition that these functions will be interacting with the `CT` 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.

Given that the `CT` type is just a type alias with `ReaderT` as the under the hood type, all of these lift operations are already provided in Haskell's libraries. However, it is still valuable to present their implementation to completely understand how the final look for the DSL will look like. Hence, the following implementations will assume we *aren't* using CT as the type alias and instead we will be showing the implementations as if we are using the definition used previously [2] for the `CT` type:

```
data CT a = CT (Parameters -> IO a)
```

With this in mind, the `Functor` typeclass, when considering this version of the `CT` type, let the lifting of functions to be enclosed by the `CT` 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 `CT` type, i.e., the outcome is a function with the signature `CT a -> CT b`. The code below shows the implementation of the *fmap* function — the minimum requirement to the `Functor` typeclass. 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.

```
instance Functor CT where
  fmap f (CT da) = CT $ \ps -> fmap f (da ps)
```

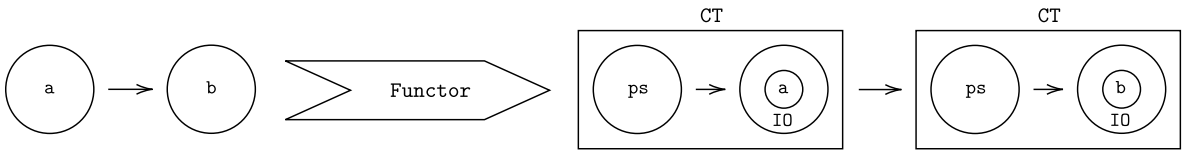


Figure 3.1: Given a parametric record `ps` and a dynamic value `da`, the *fmap* functor of the `CT` type applies the former to the latter. Because the final result is wrapped inside the `IO` shell, a second *fmap* is necessary.

The next typeclass, `Applicative`, deals with functions that are inside the `CT` type. When implemented (again, referring to the non-type-alias version), 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 `CT a -> CT b`. The minimum requirements for this typeclass is the function *pure*, a function responsible for wrapping any value with the `CT` wrapper, and the `<*>` operator, which does the aforementioned interaction between the internal values with the outer shell. The

implementation of this typeclass has the dependency `df` has the signature `CT (a -> b)` and its internal function `a -> b` is being lifted to the `CT` type. Figure 3.2 illustrates the described lifting with `Applicative`.

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

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

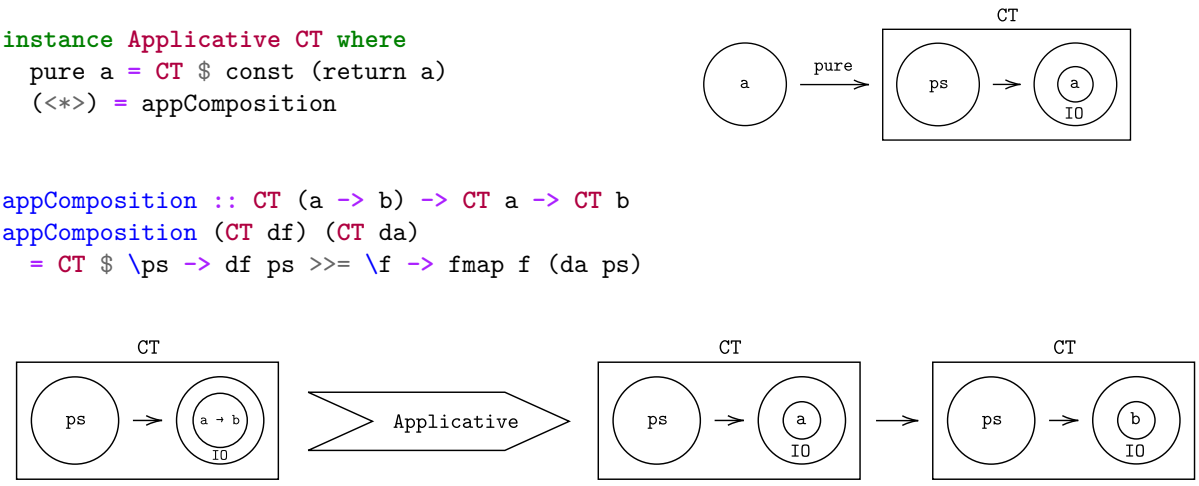


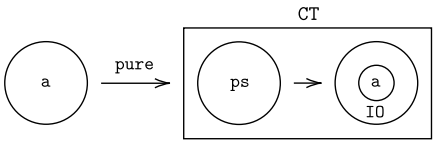
Figure 3.2: With the `Applicative` typeclass, it is possible to cope with functions inside the `CT` type. Again, the *fmap* from `IO` is being used in the implementation.

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 -> CT b` can be lifted to the signature `CT a -> CT 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 represent the minimum requirements of the `Monad` typeclass. Figure 3.3 illustrates the aforementioned scenario.

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, `CT` in this case. The usefulness of this typeclass will be more clear in the next topic, Section 3.3. The implementation follows, alongside its visual representation in Figure 3.4. Once again, consider the explicit definition for the `CT` type instead of the type alias.

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 `CT` 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 following code depicts the implementation for unary and binary operations,

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




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

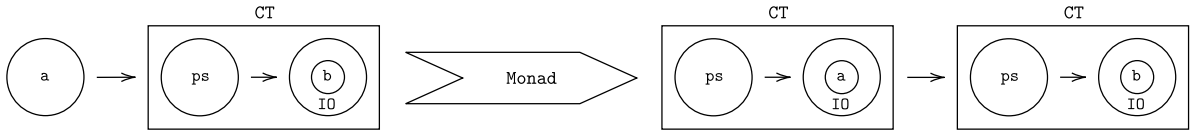


Figure 3.3: The >>= operator used in the implementation is the *bind* from the `IO` shell. This indicates that when dealing with monads within monads, it is frequent to use the implementation of the internal members.

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

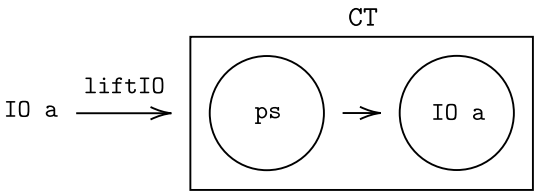


Figure 3.4: The typeclass `MonadIO` transforms a given value wrapped in `IO` into a different monad. In this case, the parameter `m` of the function is the output of the `CT` type.

which are used in the requirements for those typeclasses. As a side note, to make these implementations possible for the type-aliased version of the `CT` type, it is required to use a compiler extension `FlexibleInstances`. Further, the same operations can be used as internal helpers for both versions of the type:

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

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

3.2 GPAC Bind I: CT

After these improvements in the `CT` 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 3 out of 4 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 `CT` type represents continuous physical dynamics [8], 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 granted by the `CT` type. Next, it needs to return a constant value k for all moments in time. The `Num` given the `CT` 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 `FACT` counterparts. The forth unit and the composition rules will be mapped after describing the second main type of `FACT`: the `Integrator` type.

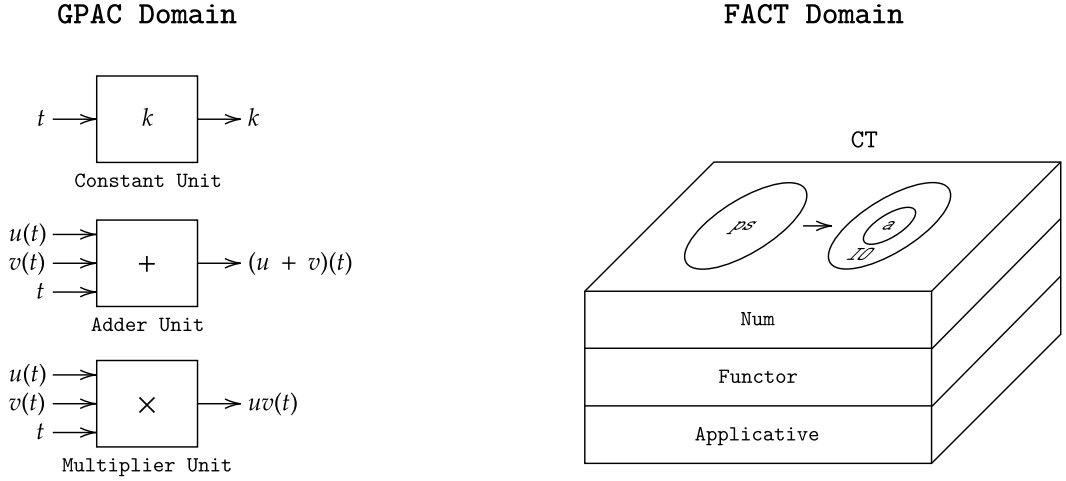


Figure 3.5: The ability of lifting numerical values to the `CT` type resembles three FF-GPAC analog circuits: `Constant`, `Adder` and `Multiplier`.

3.3 Exploiting Impurity

The `CT` 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.

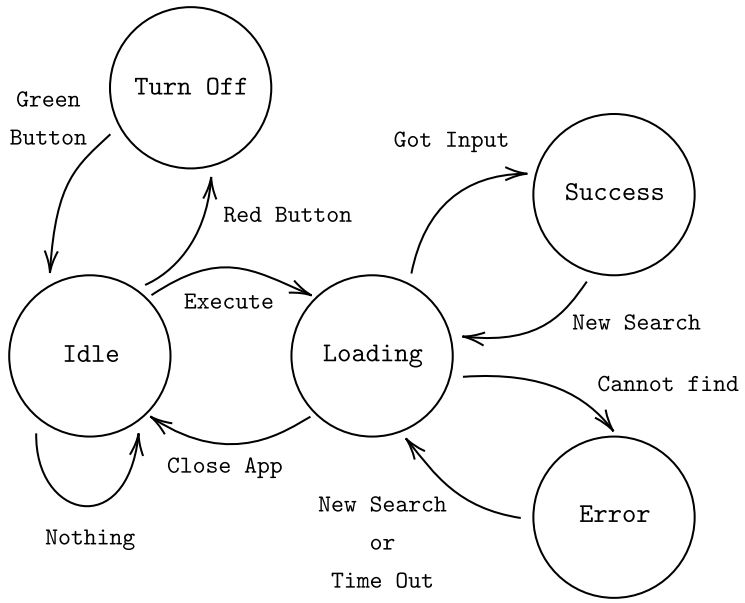


Figure 3.6: State Machines are a common abstraction in computer science due to its easy mapping between function calls and states. Memory regions and peripherals are embedded with the idea of a state, not only pure functions. Further, side effects can even act as the trigger to move from one state to 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, Fortran, Zig, Rust, 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 feature is still available but 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 `CT Double`. Also, the initial value is also represented by `CT 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.

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

There are three functions that involve the `Integrator` and the `CT` types together: the function *createInteg*, responsible for allocating the memory that the pointer will point to, *readInteg*, letting us to read from the pointer, and *updateInteg*, 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 following code is the implementation of the *createInteg* function, which creates an integrator:

```
1 createInteg :: CT Double -> CT Integrator
2 createInteg i = do
3   comp <- liftIO . newIORef $ initialize i
4   let integ = Integrator { initial = i,
5                            computation  = comp }
6   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 `CT` via the `ReaderT`. 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 (line 3) 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 (CT Double)`. At first glance, this seems to be an issue because the result of the *newIORef* function is wrapped with the `IO` monad [2]. 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` (liine 3) is capable of removing the `IO` wrapper and adding an arbitrary monad in its place, `CT` in this case. So, after line 3 the `comp` value has the desired `CT` type. The remaining step of this creation process is to construct the integrator itself by building up the record with the correct fields, e.g., the CT version of the initial value and the pointer to the constructed computation written in memory (lines 4 and 5).

```
readInteg :: Integrator -> CT Double
readInteg = join . liftIO . readIORef . computation
```

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 `CT` that applies the given record of `Parameters` to what's being stored in the region. This is accomplished by using the function `join` with the *readIORef* function [2].

Finally, the function *updateInteg* 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 `CT Double`. To update this behaviour, the *updateInteg* change the content being pointed by the integrator's pointer:

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

[2] `IORef` hackage documentation.

In the beginning of the function (line 3), we extract the initial value from the integrator, so-called `i`. Next (line 4 onward), we create a new computation, so-called `z` — a function wrapped in the `CT` type that receives a `Parameters` record and computes the result based on the solving method. Because this computation needs to do lookups on some configuration values, we use the function `ask` (line 5) from `ReaderT` to get our environment values; in this case a value of type `Parameters`. Later on, the follow-up step is to build a copy of the *same process* being pointed by the `computation` pointer (line 6). Finally, after checking the chosen solver (line 7), 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 [2]. It may seem confusing that inside `z` we are *reading* what is being pointed and later, on the last line of *updateInteg*, 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 *updateInteg* 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 6), 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 [25]. In contrast, the integrator modeled in FACT 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 `FACT` by the `CT` type with a set of typeclasses, as explained in the previous Section about GPAC. The *integrator functions*, e.g., *readInteg* and *updateInteg*, represent the composition rules.

Going back to the type signature of the *updateInteg*, `Integrator -> CT Double -> CT ()`, 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 `CT Double` type. Because the operation is just setting up the connections between the two, the functions ends with the type `CT ()`.

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 `CT` 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 `FACT` 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 *updateInteg* function. The fourth rule is also being attended it, given that the solver methods inside the *updateInteg* 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.

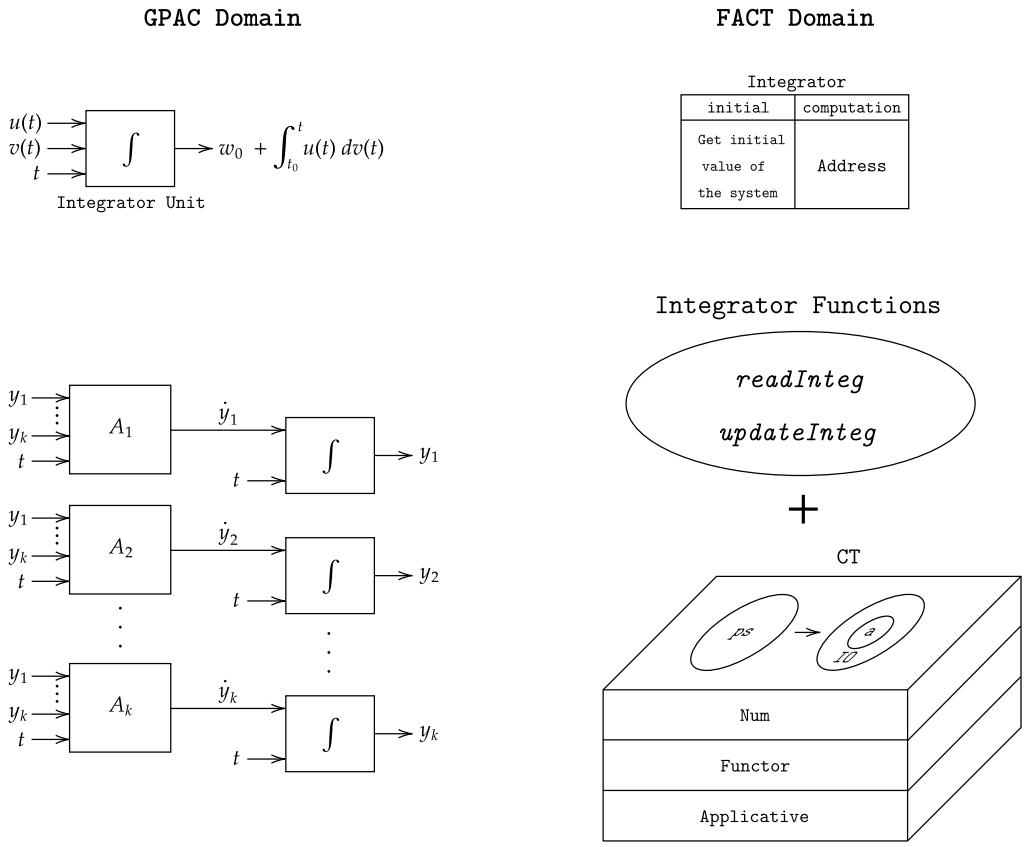


Figure 3.7: The integrator functions attend the rules of composition of FF-GPAC, whilst the `CT` and `Integrator` types match the four basic units.

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
- Fourth-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 :: CT Double
2             -> CT Double
3             -> CT Double
4             -> CT Double
5  integEuler diff init compute = do
6    ps <- ask
7    case iteration ps of
8      0 -> init
9      n -> do
10       let iv  = interval ps
11           sl  = solver ps
12           ty  = iterToTime iv sl (n - 1) 0
13           psy =
14             ps { time = ty, iteration = n - 1, solver = sl { stage = 0} }
15       a <- local (const psy) compute
16       b <- local (const psy) diff
17       let !v = a + dt (solver ps) * b
18       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 *updateInteg*; 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, originally obtained via the `ReaderT` with the `ask` function, 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 outcome is obtained by just returning the continuous machine with the initial value. 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. These machines, based on `compute` and `diff`, need to be modified with a value of type `Parameters` containing the previous iteration (so-called `psy` in the code). Hence, the function `local` is used to alterate the existing parameters value in those readers. Finally, we compute 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 `CT` 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. Later on, we will improve the DSL to completely remove all the noise introduced in its use because of such implicit recursion.

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 `CT` type. This Chapter is a follow-up, and its objectives are threefold: to describe how to map a set of differential equations to an executable model, to reveal which functions execute a given example and to present a guided-example as a proof-of-concept. For a simplified guide on how to use the DSL, check the Appendix 9.2.

4.1 From Models to Models

Systems of differential equations reside in the mathematical domain. In order to *execute* using the `FACT` 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, the mappings between these worlds need to be established. Chapters 2 and 3 explained the mapping between `FACT` and FF-GPAC. It remains to map the *semantics* of the mathematical world to the *operational* world of `FACT`. 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 `CT` type.

With that in mind, Figure 4.1 illustrates an example of a model in `FACT`, alongside its mathematical counterpart. Further, Figure 4.2 shows which FF-GPAC circuit each line

```
1  t :: CT Double
2  t = CT $ \ps -> return (time ps)

3  exampleModel :: CT Double
4  exampleModel =
5    do integ <- createInteg 1
6       let y = readInteg integ
7       updateInteg integ (y + t)
8       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.

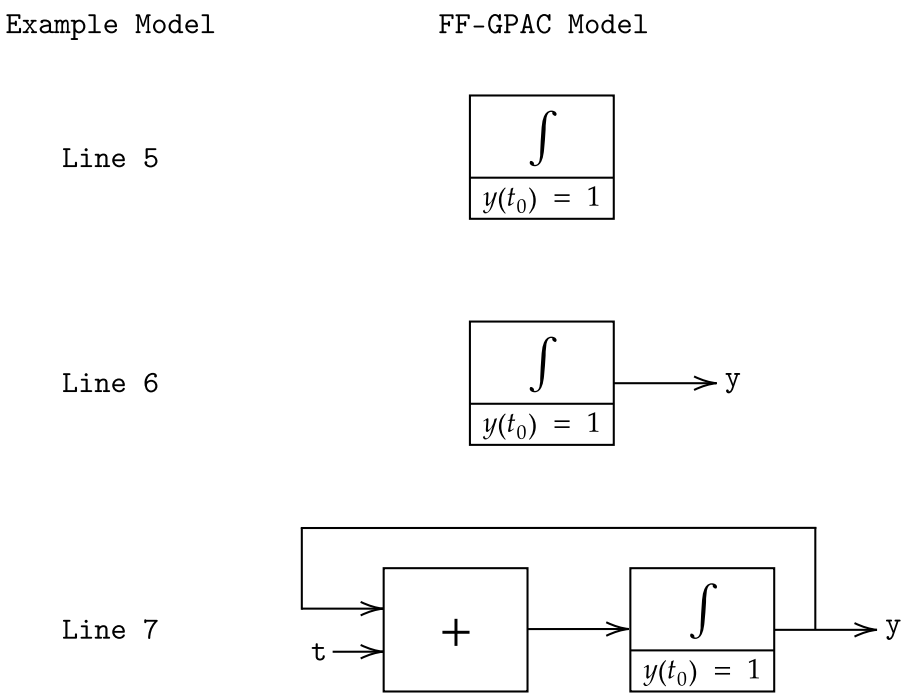


Figure 4.2: The developed DSL translates a system described by differential equations to an executable model that resembles FF-GPAC's description.

is modeling. This pipeline effectively makes `FACT` a bridge between a physical system, modeled by differential equations, and the FF-GPAC model proposed by Graça [15].

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 *updateInteg* 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.

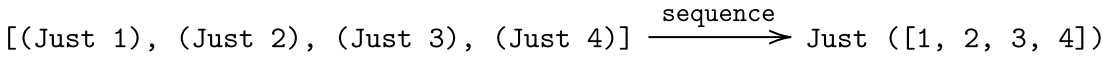


Figure 4.3: Because the list implements the `Traversable` typeclass, it allows this type to use the *traverse* and *sequence* functions, in which both are related to changing the internal behaviour of the nested structures.

Similarly to the preceding example, the list structure will be used to involve all the state variables with type `CT 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 `CT [Double]` or, by using a type aliases for `[Double]` as `Vector`, `CT Vector`. A second alias can be created to make it more descriptive, as exemplified in Figure 4.4:

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:

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

exampleModel :: Model Vector
exampleModel =
  do integX <- createInteg 1
     integY <- createInteg 1
     let x = readInteg integX
         y = readInteg integY
     updateInteg integX (x * y)
     updateInteg 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.

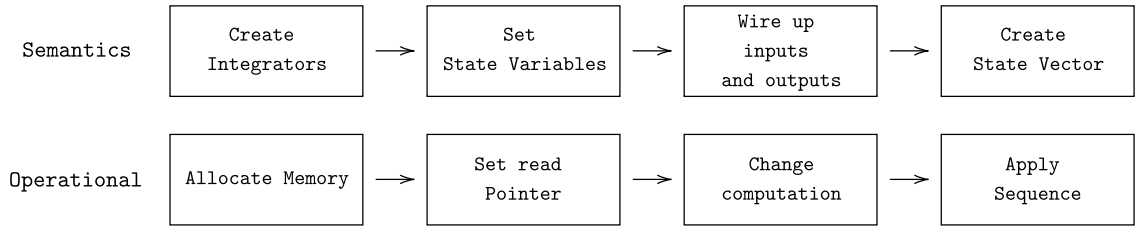


Figure 4.5: When building a model for simulation, the above pipeline is always used, from both points of view. The operations with meaning, i.e., the ones in the `Semantics` pipeline, are mapped to executable operations in the `Operational` pipeline, and vice-versa.

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 *runCT* fulfills this role:

```haskell
runCT :: Model a -> Double -> Solver -> IO [a]
runCT m t sl =
  let iv = Interval 0 t
      (nl, nu) = iterationBnds iv (dt sl)
      parameterize n =
          let time = iterToTime iv sl n 0
              solver = sl {stage = 0}
          in Parameters { interval = iv,
                          time = time,
                          iteration = n,
                          solver = solver }
  in sequence $ map (runReaderT m . parameterize) [nl .. nu]
```

On line 3, we convert the final *time value* for the simulation into an interval value for the simulation (`iv`) — the simulation always starts at 0 and goes all the way up to the requested time. Next up, on line 4, we convert the interval 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 5 to line 11, we are defining an auxiliary function *parameterize*. 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 7), 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 13 produces the outcome of the *runCT* 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 continuous machine `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 `CT` 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 *runCTFinal*, that return only the final result of the simulation instead of the outputs at the time step samples. The next section will provide an example of this in a step-by-step manner.

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 simpler system for demonstration purposes, 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:

```
1  lorenzSolver = Solver { dt = 0.01,
2                          method = RungeKutta2,
3                          stage = 0
4                        }

5  sigma = 10.0
6  rho = 28.0
7  beta = 8.0 / 3.0

8  lorenzModel :: Model Vector
9  lorenzModel =
10   do integX <- createInteg 1.0
11      integY <- createInteg 1.0
12      integZ <- createInteg 1.0
13      let x = readInteg integX
14          y = readInteg integY
15          z = readInteg integZ
16      updateInteg integX (sigma * (y - x))
17      updateInteg integY (x * (rho - z) - y)
18      updateInteg integZ (x * y - beta * z)
19      return $ sequence [x, y, z]

20 lorenzSystem = runCT lorenzModel 100 lorenzSolver
```

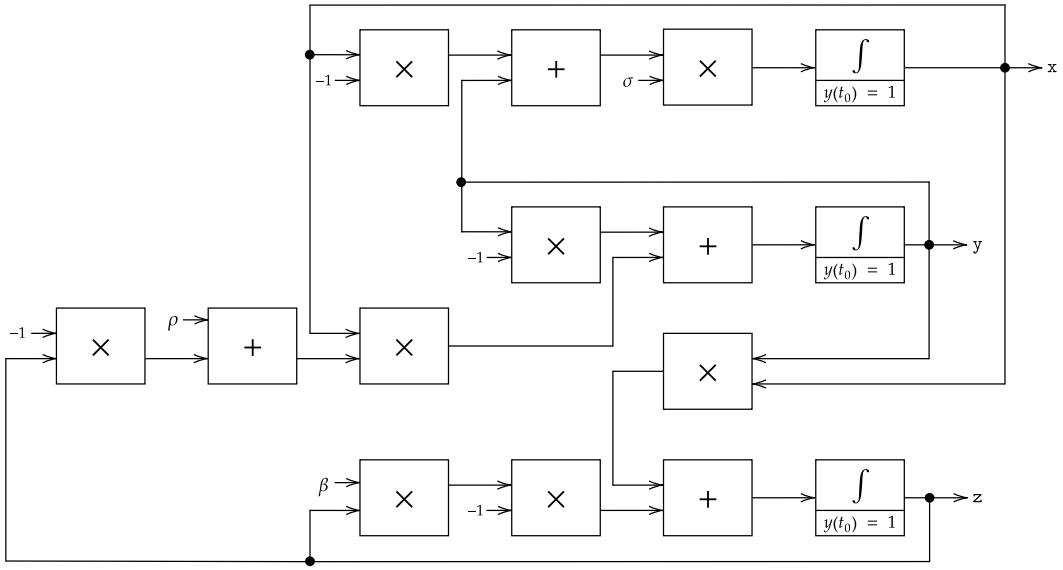


Figure 4.6: Using only FF-GPAC's basic units and their composition rules, it's possible to model the Lorenz Attractor example.

The first records, `Solver`, sets the environment (lines 1 to 4). It 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 6 to 8), 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 *runCT* 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 *runCT* function with a model, final time for the simulation and solver configurations. Following its path of execution, the *map* function (inside the driver) forces the application of a parametric record generated by the *parameterize* function to the provided model, *lorenzModel* in this case. Thus, it needs to be executed in order to return from the *runCT* 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 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 *createInteg* 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 `CT Double` that, regardless of the parameters record it receives, returns the initial condition of the system. The latter is a pointer or address that references a specific `CT 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`.

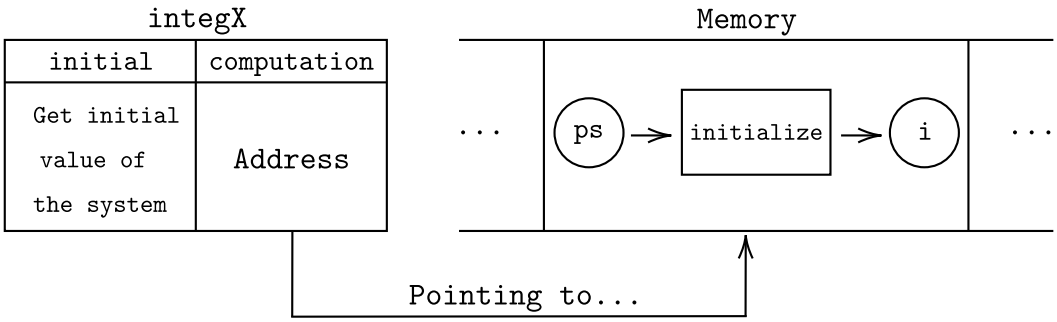


Figure 4.7: After *createInteg*, this record is the final image of the integrator. The function *initialize* gives us protecting against wrong records of the type `Parameters`, assuring it begins from the first iteration, i.e., t_0.

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 `CT 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 `CT Double` value. The state variable, x in this case, combines its received `Parameters` value, so-called `ps`, and *applies* it to the stored continuous machine. 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 *updateInteg* 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 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.

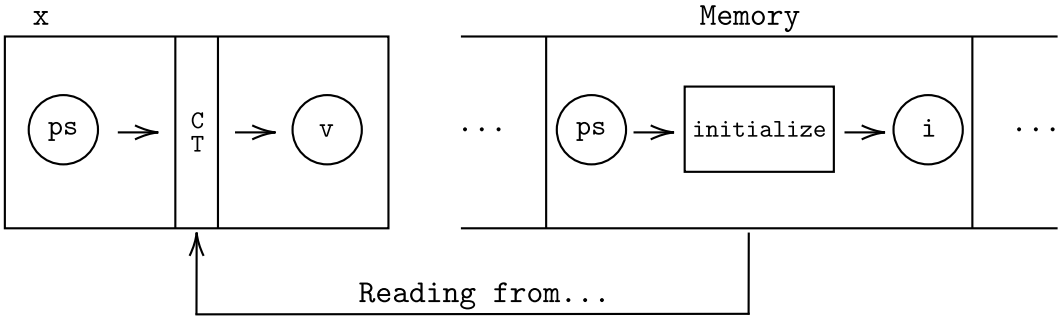


Figure 4.8: After *readInteg*, the final floating point values is obtained by reading from memory a computation and passing to it the received parameters record. The result of this application, v, is the returned value.

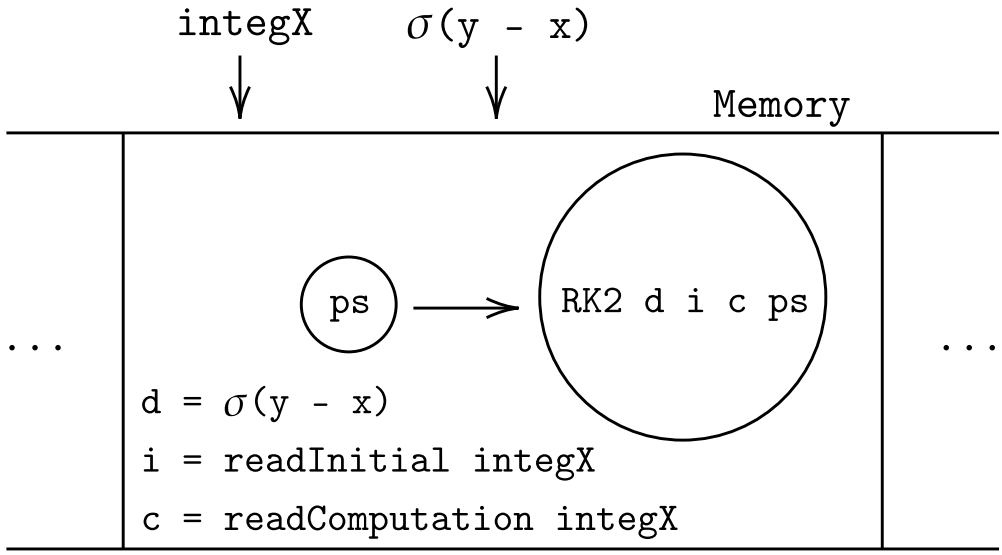


Figure 4.9: The *updateInteg* function only does side effects, meaning that only affects memory. The internal variable `c` is a pointer to the computation *itself*, i.e., the computation being created references this exact procedure.

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 `[CT Double]`, is transformed into a value with the type `CT [Double]`. The transformation can be visually understood when looking 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*

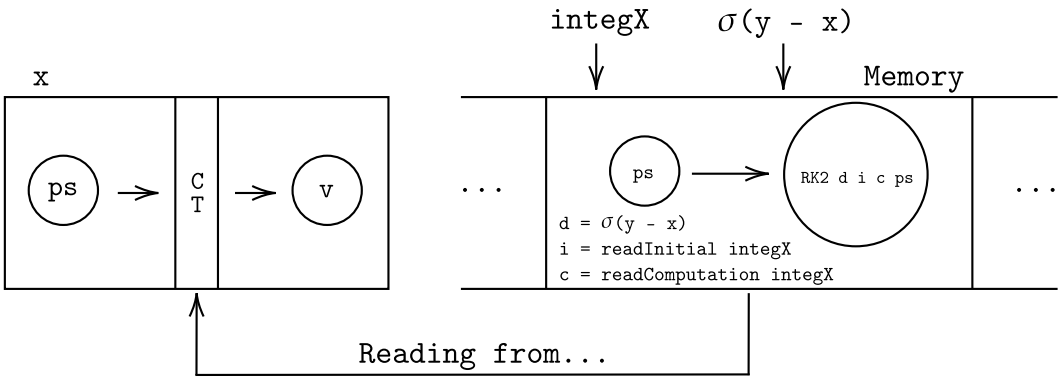


Figure 4.10: After setting up the environment, this is the final depiction of an independent variable. The reader x reads the values computed by the procedure stored in memory, a second-order Runge-Kutta method in this case.

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

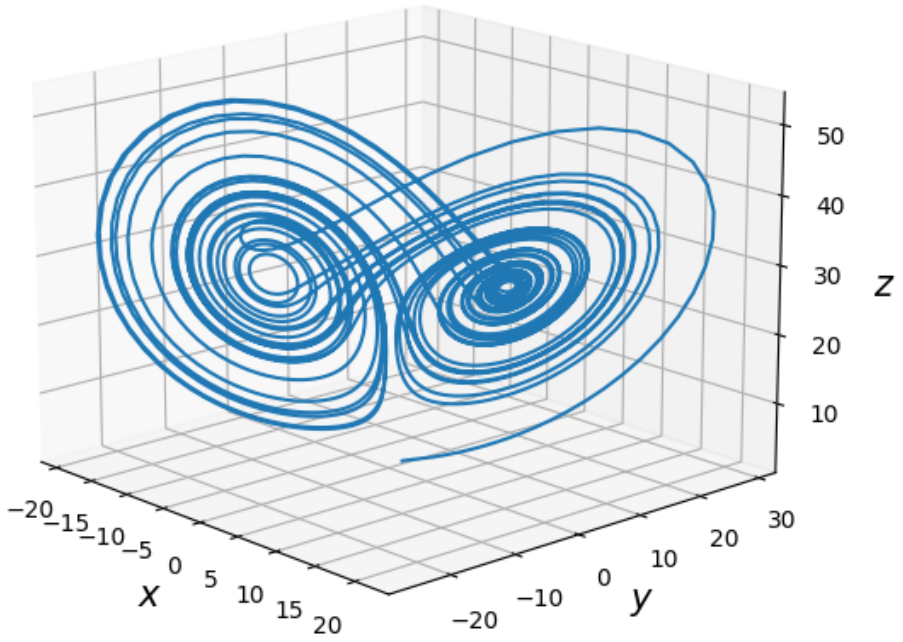


Figure 4.11: The Lorenz's Attractor example has a very famous butterfly shape from certain angles and constant values in the graph generated by the solution of the differential equations..

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

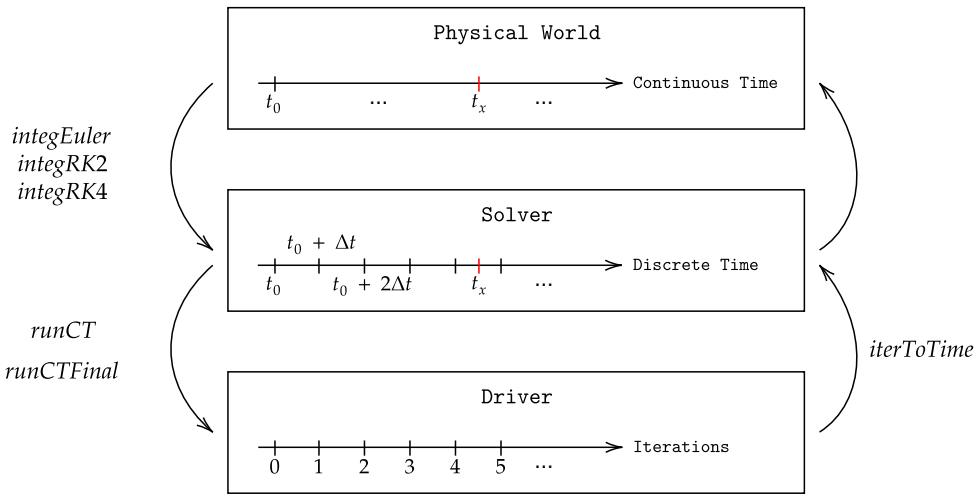


Figure 5.1: During simulation, functions change the time domain to the one that better fits certain entities, such as the `Solver` and the driver. The image is heavily inspired by a figure in [4].

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, *runCT* and *runCTFinal*. 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 function is presented bellow:

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

To achieve the total number of iterations, the function *iterationBnds* does a *ceiling* 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 solver stage in 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 fourth 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 three parts: the types, 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.

Hence, there is a need to introduce a mechanism to identify these different situations. As the solution, we will add the new type depicted in Figure 5.2.

```haskell
data Stage = SolverStage Int
           | Interpolate
           deriving (Eq, Ord, Show)
```

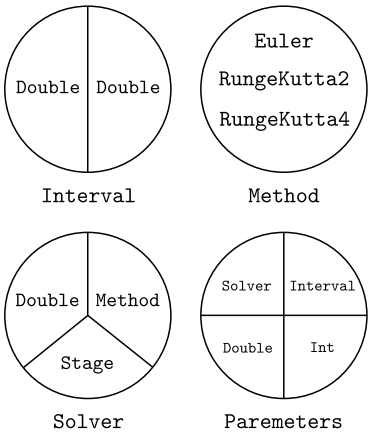


Figure 5.2: Updated auxiliary types for the `Parameters` type.

The type `Stage` allows values to be either the normal flow of execution, marked by the use of `SolverStage`, or the indication that an extra step for interpolation needs

to be done, marked by the `Interpolate` tag. Moreover, previous types and functions described in previous chapters, such as *Design Philosophy*, and *Effectful Integrals* need to be adapted to use this new type instead of the original `Int` previously proposed (in Chapter 2, *Design Philosophy*). Types like `Parameters` and functions like *integEuler*, *iterToTime*, and *runCT* need to be updated accordingly. In all of those instances, processing will just continue normally; `SolverStage` will be used.

Next, the driver needs to be updated. So, the proposed mechanism is the following: the driver will identify these corner cases and communicate to the integrator — via the new `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.

It remains to re-implement the driver functions. The driver will notify the integrator that an interpolation needs to take place. The following code 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  runCT :: Model a -> Double -> Solver -> IO [a]
6  runCT m t sl =
7    do let iv = Interval 0 t
8           (nl, nu) = iterationBnds iv (dt sl)
9           parameterize n =
10             let time = iterToTime iv sl n (SolverStage 0)
11                 solver = sl {stage = SolverStage 0}
12             in Parameters { interval = iv,
13                             time = time,
14                             iteration = n,
15                             solver = solver }
16          disct = iterToTime iv sl nu (SolverStage 0)
17          values = map (runReaderT m . parameterize) [nl .. nu]
18   sequence $
19     if disct - t < epslon
20     then values
21     else let ps = Parameters { interval = iv,
22                                time = t,
23                                iteration = nu,
24                                solver = sl {stage = Interpolate} }
25          in init values ++ [runReaderT m ps]
```

The implementation of *iterationBnds* uses *ceiling* function because this rounding is used to go to the iteration domain. However, given that 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 15 (from the previous code snippet) are equal to the previous implementation of the *runCT* function. On line 16, the discrete version of `t`, `disct`, will be used for detecting if an interpolation will be needed. All the simulation values are being prepared on line 17 — Haskell being a lazy language the label `values` will not necessarily be evaluated strictly. Line 19 establishes a condition, checking if the difference between the time of interest `t` and `disct` is greater or not than a value `epslon`, to identify if the normal flow of execution can proceed. If it can't, on line 22 a new record of type `Parameters` is created (`ps`), especifically to these special cases of mismatch between discrete and continuous time. The main difference within this special record is relevant: the stage field of the solver is being set to `Interpolate`. Finally, on line 25 the last element from the list of outputs `values` is removed and it is appended the simulation using the created `ps` with interpolation configured.

```
1  interpolate :: CT Double -> CT Double
2  interpolate m = do
3    ps <- ask
4    case stage $ solver ps of
5      SolverStage _ -> m
6      Interpolate   ->
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 (SolverStage 0)
15           t2 = iterToTime iv sl n2 (SolverStage 0)
16           ps1 = ps { time = t1,
17                      iteration = n1,
18                      solver = sl { stage = SolverStage 0 }}
19           ps2 = ps { time = t2,
20                      iteration = n2,
21                      solver = sl { stage = SolverStage 0 }}
22           z1 = local (const ps1) m
```

```
23            z2 = local (const ps2) m
24        in z1 + (z2 - z1) * pure ((t - t1) / (t2 - t1))
```

Lines 1 to 5 (from the previuos code snippet) continues the simulation with the normal workflow. 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 16 and 19) 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 22 and 23). Finally, line 24 executes the linear interpolation with the obtained values that surround the non-discrete time point. This particular interpolation was chosen for the sake of simplicity, but it can be replaced by higher order methods. Figure 5.3 illustrates the effect of the *interpolate* function when converting domains.

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

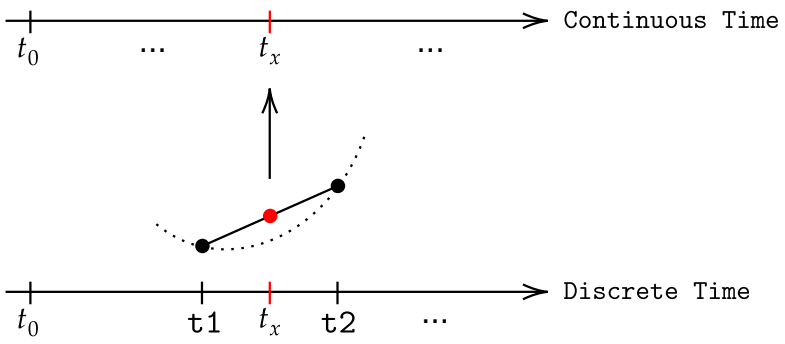


Figure 5.3: Linear interpolation is being used to transition us back to the continuous domain..

The last step in this tweak is to add this function into the integrator function *updateInteg*. The code is almost identical to the one presented in Chapter 3, *Effectful Integrals*.

The main difference is in line 11, where the interpolation function is being applied to `z`. Figure 5.4 shows the same visual representation for the *updateInteg* function used in Chapter 4, but with the aforementioned modifications.

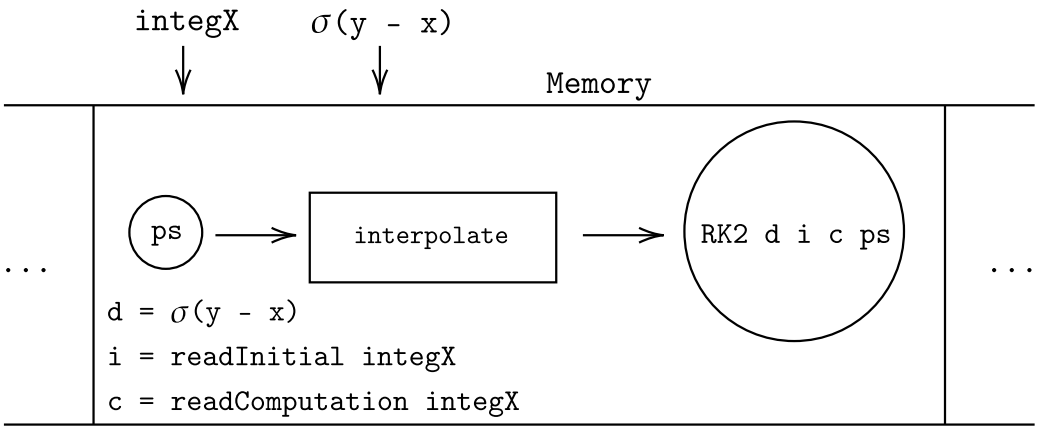


Figure 5.4: The new *updateInteg* function add linear interpolation to the pipeline when receiving a parametric record.

This concludes the first tweak in `FACT`. 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 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 `FACT`. 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 `FACT` 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 `Ryzen 7 5700X` AMD processor and 128GB 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 `FACT` is inneficient, showing an exponential 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 (milliseconds)	Consumed Memory (KB)
1	0.01	6.1
2	0.03	77.1
3	0.42	540.7
4	2.67	7040.0
5	20.33	92061.4
6	372.51	1205061.8
7	4625.33	15774437.2
8	44164.82	206490062.5
9	629253.31	2702990062.9
10	7369477.84	35382599438.3

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

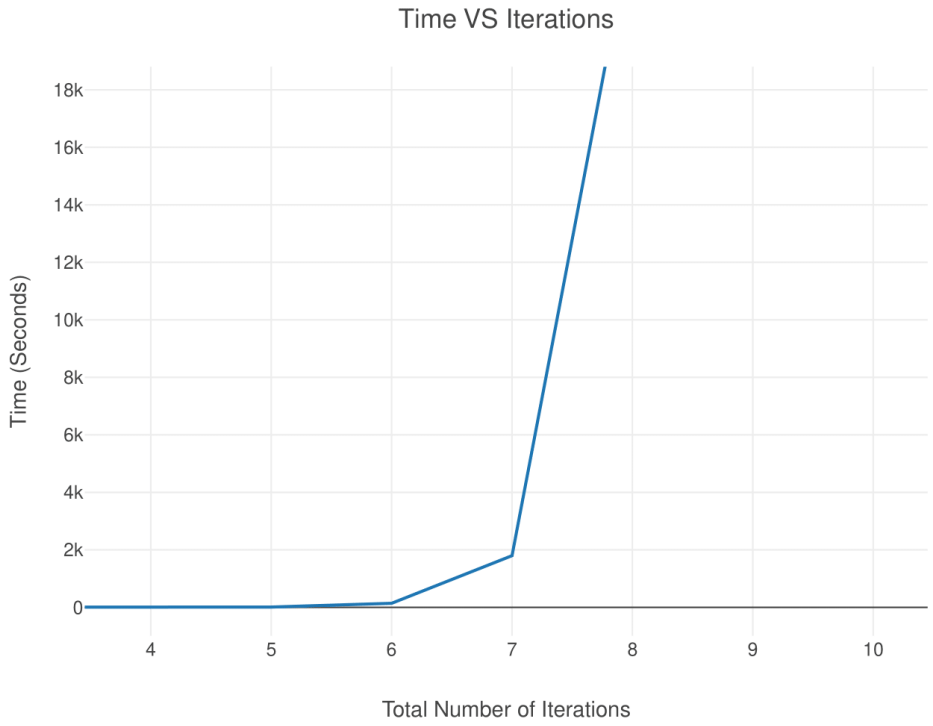


Figure 6.1: With just a few iterations, the exponential behaviour of the implementation is already noticeable.

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 :: CT Double
2             -> CT Double
3             -> CT Double
4             -> CT Double
5  integEuler diff i y = do
6    ps <- ask
7    case iteration ps of
8      0 -> i
9      n -> do
10       let iv  = interval ps
11           sl  = solver ps
12           ty  = iterToTime iv sl (n - 1) (SolverStage 0)
13           psy = ps { time = ty, iteration = n - 1, solver = sl { stage = SolverStage
             ↪  0} }
14       a <- local (const psy) y
15       b <- local (const psy) diff
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*. 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 `FACT` 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 `CT` 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 a *mutable* array, which is the case for our `Double` values. As dependencies, the *memo* function receives the 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 => (CT e -> CT e) -> CT e -> CT (CT e)
2  memo interpolate m = do
3    ps <- ask
4    let sl = solver ps
5        iv = interval ps
6        (SolverStage stl, SolverStage stu) = stageBnds sl
7        (nl, nu)   = iterationBnds iv (dt sl)
8    arr   <- liftIO $ newMemoUArray_ ((stl, nl), (stu, nu))
9    nref  <- liftIO $ newIORef 0
10   stref <- liftIO $ newIORef 0
11   let r = do
12         ps <- ask
13         let sl  = solver ps
14             iv  = interval ps
15             n   = iteration ps
16             st  = getSolverStage $ stage sl
17             stu = getSolverStage $ stageHiBnd sl
18             loop n' stage' =
19               if (n' > n) || ((n' == n) && (stage' > st))
20               then
21                 readArray arr (st, n)
22               else
23                 let ps' = ps { time = iterToTime iv sl n' (SolverStage stage'),
24                                iteration = n',
25                                solver = sl { stage = SolverStage stage' }}
26                 in do a <- runReaderT m ps'
27                       a `seq` writeArray arr (stage', n') a
28                       if stage' >= stu
29                         then do writeIORef stref 0
30                                 writeIORef nref (n' + 1)
31                                 loop (n' + 1) 0
32                         else do writeIORef stref (stage' + 1)
33                                 loop n' (stage' + 1)
34         n'  <- liftIO $ readIORef nref
35         st' <- liftIO $ readIORef stref
36         liftIO $ loop n' st'
37   pure . interpolate $ 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 7). This is used to pre-allocate the minimum amount of memory required for the execution (line 8). 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 9 and 10), to read iteration and stage values respectively.

The code block from line 11 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 it 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 continuous machine `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 reset 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 `CT` shell via the *pure* function (provided by the `Applicative` 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 :: CT Double,
2                                cache   :: IORef (CT Double),
3                                computation  :: IORef (CT Double)
4                              }
```

Next, two other functions need to be adapted: *createInteg* 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 *createInteg* 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 *updateInteg* in a model to the differential equation, is being read in lines 8 to 11 and piped with interpolation and memoization in line 12. 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 13).

Figure 6.2 shows that the updated version of the *createInteg* 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 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. This means that the same `readComp`, described in the new *createInteg* function, will receive a given `ps`. It is worth noticing that, just like with the *createInteg* function, this cache pointer indirectly interacts with the same memory location pointed by the `computation` pointer in the integrator (Figure 6.3).

```haskell
createInteg :: CT Double -> CT Integrator
createInteg i = do
  r1 <- liftIO . newIORef $ initialize i
  r2 <- liftIO . newIORef $ initialize i
  let integ = Integrator { initial = i,
                           cache = r1,
                           computation  = r2 }
      z = do
        ps <- ask
        v <- liftIO $ readIORef (computation integ)
        local (const ps) v
  y <- memo interpolate z
  liftIO $ writeIORef (cache integ) y
  return integ
```

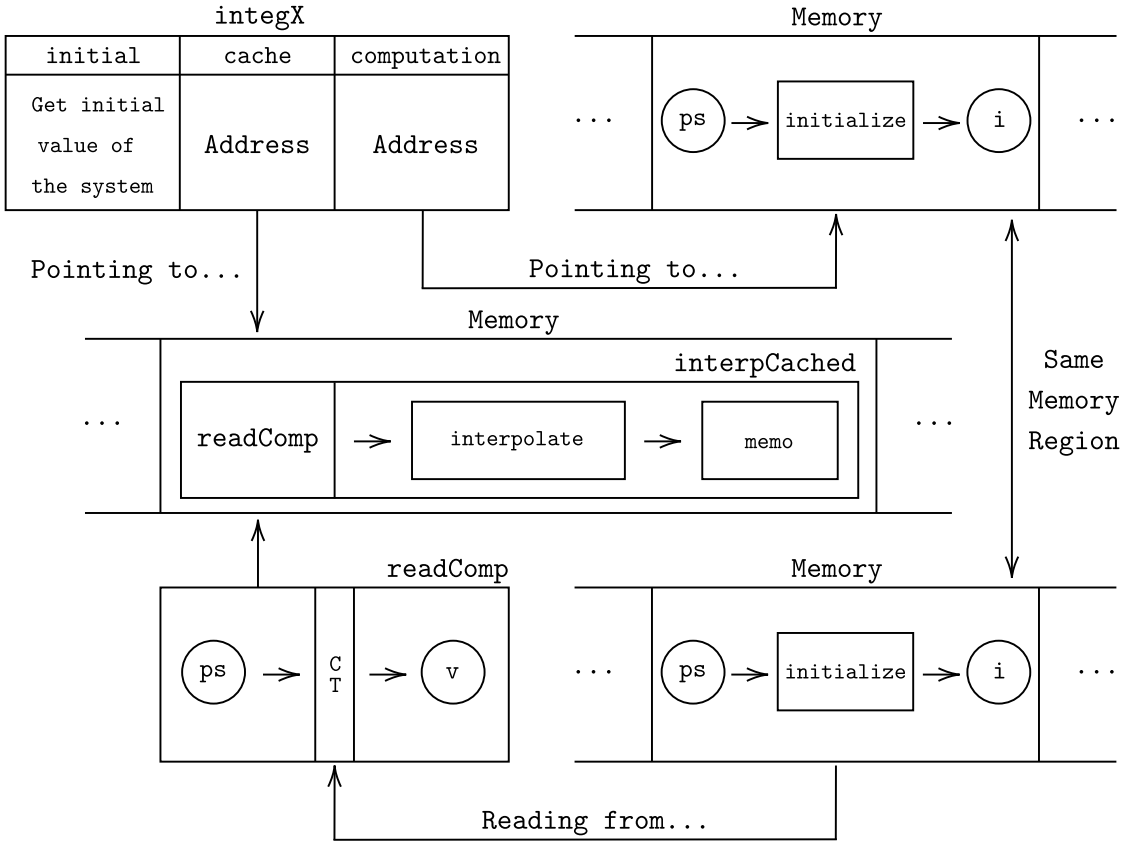


Figure 6.2: The new *createInteg* function relies on interpolation composed with memoization. Also, this combination *produces* results from the computation located in a different memory region, the one pointed by the `computation` pointer in the integrator.

```haskell
readInteg :: Integrator -> CT Double
readInteg = join . liftIO . readIORef . cache
```

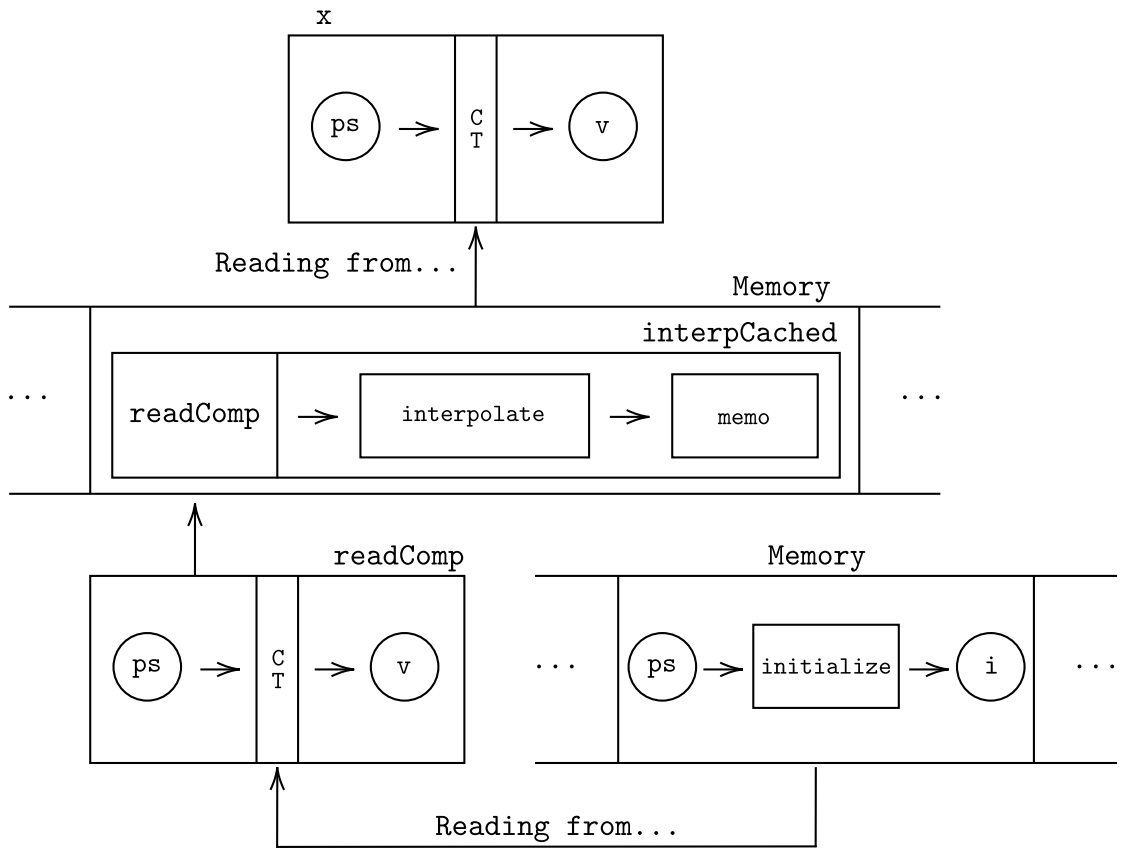


Figure 6.3: The function *reads* information from the caching pointer, rather than the pointer where the solvers compute the results.

Lastly, Figure 6.4 depicts the new version of the *updateInteg* 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
updateInteg :: Integrator -> CT Double -> CT ()
updateInteg integ diff = do
  let i = initial integ
      z = do
        ps <- ask
        let f =
              case (method $ solver ps) of
                Euler -> integEuler
                RungeKutta2 -> integRK2
                RungeKutta4 -> integRK4
        y <- liftIO $ readIORef (cache integ)
        f diff i y
  liftIO $ writeIORef (computation integ) z
```

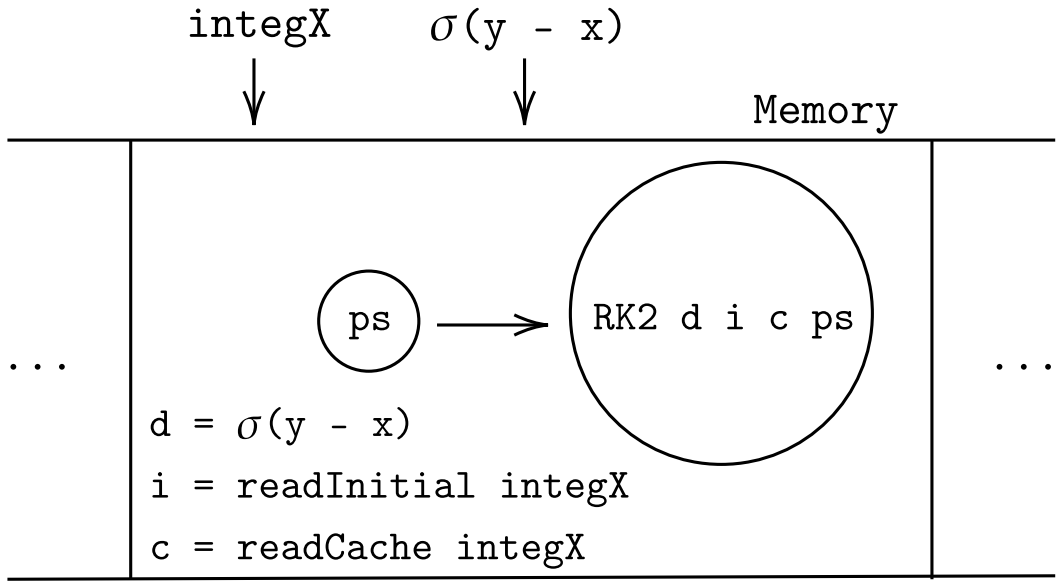


Figure 6.4: The new *updateInteg* function gives to the solver functions access to the region with the cached data.

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 *runCTFinal* 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 `FACT`. 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.

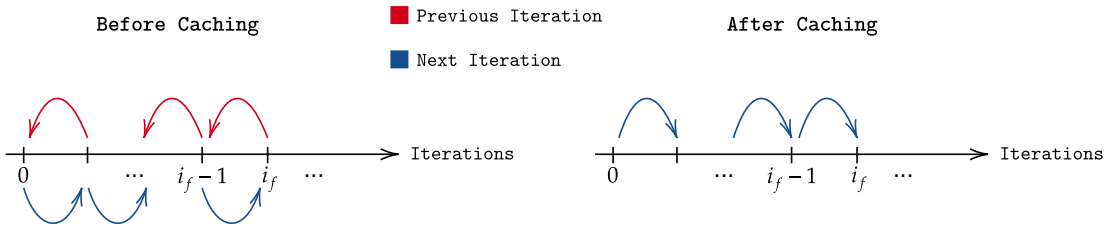


Figure 6.5: Caching changes the direction of walking through the iteration axis. It also removes an entire pass through the previous iterations.

6.5 Tweak III: Model and Driver

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

```
1 exampleModel :: Model Vector
2 exampleModel =
3   do integX <- createInteg 1
4      integY <- createInteg 1
5      let x = readInteg integX
6          y = readInteg integY
7      updateInteg integX (x * y)
8      updateInteg 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 *createInteg* 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 `CT` 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 = CT (CT a)

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

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

```
runCT :: Model a -> Double -> Solver -> IO [a]
runCT m t sl = do
  d <- runReaderT m $ Parameters { interval = Interval 0 t,
                                   time = 0,
                                   iteration = 0,
                                   solver = sl { stage = SolverStage 0}}
  sequence $ subRunCT d t sl

subRunCT :: CT a -> Double -> Solver -> [IO a]
subRunCT m t sl = do
  let iv = Interval 0 t
```

[2] Garbage Collector wiki page.

```
11        (nl, nu) = iterationBnds iv (dt sl)
12        parameterize n =
13          let time = iterToTime iv sl n (SolverStage 0)
14              solver = sl {stage = SolverStage 0}
15          in Parameters { interval = iv,
16                          time = time,
17                          iteration = n,
18                          solver = solver }

19        disct = iterToTime iv sl nu (SolverStage 0)
20        values = map (runReaderT m . parameterize) [nl .. nu]
21    if disct - t < epslon
22    then values
23    else let ps = Parameters { interval = iv,
24                               time = t,
25                               iteration = nu,
26                               solver = sl {stage = Interpolate} }
27         in init values ++ [runReaderT 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 is doing the remaining functionality. Thus, this is the final implementation of the driver in `FACT`.

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. It is worth noting that there is an overhead due to the memoization strategy when running fewer iterations (such as 1 in the table), in which most time is spent preparing the caching setup — the in-memory data structure, and etc.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 (milliseconds)	Execution Time (milliseconds)	Consumed Memory (KB)
1	0.01	0.05	70.73
2	0.03	0.02	64.84
3	0.42	0.03	91.79
4	2.67	0.03	122.72
5	20.33	0.03	145.68
6	372.51	0.04	172.62
7	4625.33	0.04	199.56
8	44164.82	0.05	226.51
9	629253.31	0.06	253.45
10	7369477.84	0.06	280.40

Table 6.3: These values were obtained using the same hardware. It shows that the caching strategy drastically improves `FACT`'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 (milliseconds)	Consumed Memory (MB)
100	1.57	2.73
1K	5.80	26.95
10K	50.55	269.61
100K	519.92	2696.05
1M	5099.89	26960.65
10M	51957.02	269606.50
100M	520663.60	2696065.05

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

The project is currently capable of executing interpolation as well as applying memoization to speed up results. These two drawback solutions, detailed in Chapter 5 and 6, adds practicality to `FACT` as well as makes it more competitive. But we can, however, go even further and adds more familiarity to the DSL. The next Chapter, *Fixing Recursion*, will address this concern.

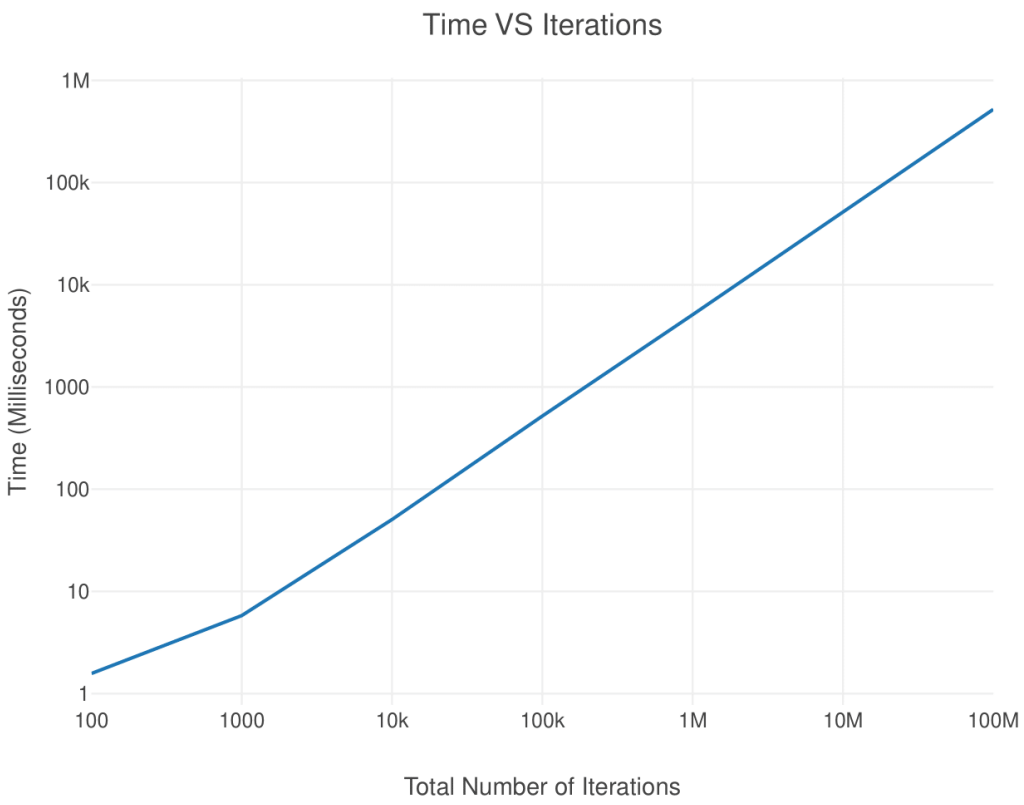


Figure 6.6: By using a logarithmic scale, we can see that the final implementation is performant with more than 100 million iterations in the simulation.

Chapter 7

Fixing Recursion

$$f(x) = x$$

The last improvement for FACT is in terms of *familiarity*. When someone is using the DSL, so-called **designer** of the system, the main goal should be that the least amount of friction when using the simulation software, the better. Hence, the requirement of knowing implementation details or programming language details is something we would like to avoid, given that it leaks noise into the designer's mind. The designer's concern should be to pay attention to the system's description and FACT having an extra step of translation or noisy setups just adds an extra burden with no real gains on the engineering of simulating continuous time. This Chapter will present *FFACT*, an evolution of FACT which aims to reduce the noise even further.

7.1 Integrator's Noise

Chapter 4, *Execution Walkthrough*, described the semantics and usability on an example of a system in mathematical specification and its mapping to a simulation-ready description provided by FACT. We have this example modeled using FACT (same code as provided in Section 1.1):

```
1  sigma = 10.0
2  rho = 28.0
3  beta = 8.0 / 3.0

4  lorenzModel :: Model Vector
5  lorenzModel =
6    do integX <- createInteg 1.0
7       integY <- createInteg 1.0
8       integZ <- createInteg 1.0
9       let x = readInteg integX
```

```
10          y = readInteg integY
11          z = readInteg integZ
12      updateInteg integX (sigma * (y - x))
13      updateInteg integY (x * (rho - z) - y)
14      updateInteg integZ (x * y - beta * z)
15      return $ sequence [x, y, z]
```

It is noticeable, however, that FACT imposes a significant amount of overhead from the user's perspective due to the **explicit use of integrators** for most memory-required simulations. When creating stateful circuits, an user of FACT is obligated to use the integrator's API, i.e., use the functions `createInteg` (lines 6 to 8), `readInteg` (lines 9 to 11), and `updateInteg` (lines 12 to 14). Although these functions remove the management of the aforementioned implicit mutual recursion mentioned in Chapter 3, *Effectful Integrals*, from the user, it is still required to follow a specific sequence of steps to complete a model for any simulation:

1. Create Integrators for future use setting initial conditions (via the use of *createInteg*);
2. Retrieve access to state variables by reading integrators (via the use of *readInteg*);
3. Update integrators with the actual ODEs of interest (via the use of *updateInteg*).

Visually, this step-by-step list for FACT's models follow the pattern detailed in Figure 4.5 in Chapter 4, *Execution Walkthrough*:

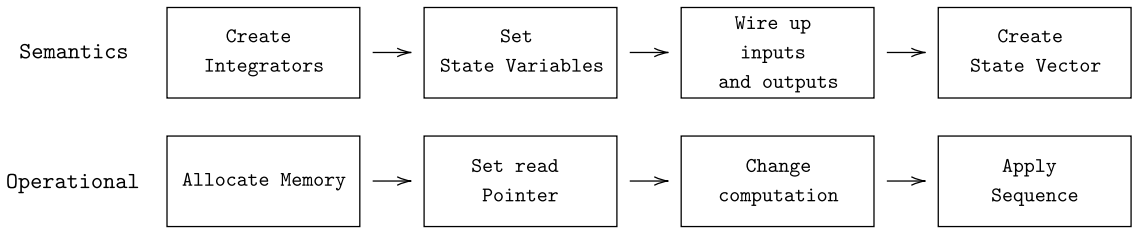


Figure 7.1: Pipeline of execution when creating a model in `FACT`.

More importantly, *all* those steps are visible and transparent from an usability's point of view. Hence, a system's designer *must* be aware of this *entire* sequence of mandatory steps, even if his interest probably only relates to lines 12 to 14. Although one's goal is being able to specify a system and start a simulation, there is no escape – one has to bear the noise created due to the implementation details of the DSL. In fact, this *abtraction leak* of exposing operational semantics is a major obstacle that keeps FACT away from one of its goals: to provide a direct map between the mathematical description of the system and its software counterpart.

To address this, FACT was upgraded to FFACT: a *fixed-point* based version of FACT. FFACT leverages the *fixed-point combinator* from the realm of mathematics to significantly reduce the surface noise when using the DSL. This, combined with Haskell's *laziness*, is the required piece to get rid of the `Integrator` type, thus also removing its noise.

7.2 The Fixed-Point Combinator

It is worth noting that the term *fixed-point* has different meanings in the domains of engineering and mathematics. When referencing the fractional representations within a computer, one may use the *fixed-point method*. Thus, to avoid confusion, the following is the definition of such concept in this dissertation, alongside a set of examples of its use case as a mathematical combinator that can be used to implement recursion.

On the surface, the fixed-point combinator is a simple mapping that fulfills the following property: a point p is a fixed-point of a function f if $f(p)$ lies on the identity function, i.e., $f(p) = p$. Not all functions have fixed-points, and some functions may have more than one [30]. Further, we seek to establish theorems and algorithms in which one can guarantee fixed-points and their uniqueness, such as the Banach fixed-point theorem [31]. In programming terms, by following specific requirements one could find the fixed-point of a function via an iterative process that involves going back and forth between it and the identity function until the difference in outcomes is less than or equal to an arbitrary ϵ.

Of particular interest is to find *the least fixed-point*, a mathematical construct useful when describing the denotational semantics of recursive definitions [32, 30]. Within mathematics, you can find fixed-points in domains such as lattices [32], metric spaces [31], lambda calculus, among others areas that study convergence and stability of processes.

In the Haskell programming language, the fixed-point combinator is under the `Data.Function` [1] package with the following implementation:

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

This function allows the definition of recursive functions without the use of self-reference, such as:

```
factorial :: Int -> Int
factorial = fix (\f n -> if n == 1 then 1 else n * f (n - 1))
```

For readers unfamiliar with the use of this combinator, equational reasoning [30] can help understanding its meaning.

[1] `Data.Function` hackage documentation.

```
  factorial 5
= {definition of factorial, alpha equivalence to remove clashes on f}
  fix (\g n -> if n == 1 then 1 else n * g (n - 1)) 5
= {definition of fix}
  (\f -> f (f (f (...)))) (\g n -> if n == 1 then 1 else n * g (n - 1)) 5
= {function application, f = (\g n -> if n == 1 then 1 else n * g (n - 1))}
  (\g n -> if n == 1 then 1 else n * g (n - 1)) (f (f (...))) 5
= {function application, g = (f (f (...)))}
  (\n -> if n == 1 then 1 else n * f (f (...)) (n - 1)) 5
= {function application, n = 5}
  if 5 == 1 then 1 else 5 * f (f (...)) 4
= {replace f with its binding}
  if 5 == 1 then 1
  else 5 * ((\g n -> if n == 1 then 1 else n * g (n - 1)) (f (...))) 4
= {function application, g = (f (...))}
  if 5 == 1 then 1
  else 5 * ((\n -> if n == 1 then 1 else n * f (...)) (n - 1)) 4
= {function application, n = 4}
  if 5 == 1 then 1
  else 5 * (if 4 == 1 then 1 else 4 * f (...) 3)
...
```

The result of this process will yield the factorial of 5, i.e., 120. When using `fix` to define recursive processes, the function being *applied* to it must be the one defining the convergence criteria for the iterative process of looking for the fixed-point. In our factorial case, this is done via the conditional check at the beginning of body of the lambda. The fixed point combinator's responsibility is to keep the *repetition* process going – something that may diverge and run out of computer resources.

Furthermore, this process can be used in conjunction with monadic operations as identified by Levent [33]:

```haskell
countDown :: Int -> IO ()
countDown = fix (\f n -> if n == 0
                         then print "Done!"
                         else do print n
                         f (n - 1))
```

This combination, however, cannot address *all* cases when using side-effects. Executing the side-effect in `countDown` do not contribute to its own *definition*. There is no construct or variable that requires the side-effect to be executed in order to determine its meaning. This ability – being able to set values based on the result of running side-effects whilst keep the fixed-point running – is something of interest because, as we are about to see, this allows the use of *cyclic* definitions.

7.3 Value Recursion with Fixed-Points

Consider the following block diagram as the representation of a resettable circuit in hardware:

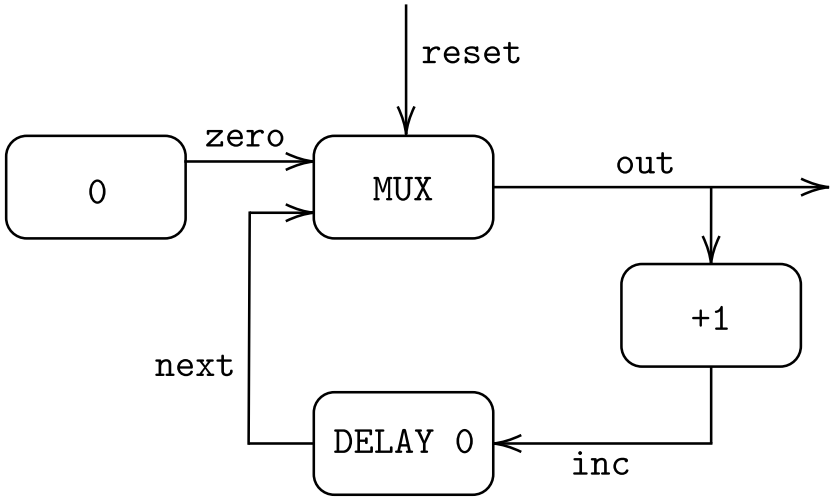


Figure 7.2: Resettable counter in hardware, inspired by Levent's works [5, 6].

When attempting to model the circuit in Figure 7.2 in programming languages other than specific hardware description languages (e.g. Verilog or VHDL), a very natural first draft of the implementation may look like:

```
counter :: Signal Bool -> Circuit (Signal Int)
counter reset = do next <- delay 0 inc
                   inc <- lift1 (+1) out
                   out <- mux reset zero next
                   zero <- lift0 0
```

This example, although idiomatic with the visual representation, does not work in its current state. This is due to the presence of *cyclic definitions* within the `do`-block, which are not allowed by Haskell's desugaring rules. This implementation detail forbid us from describing the implementation of feedback-loops in this natural way, even though it better represents the problem on-hand.

Effectively, we wish to have the flexibility of *letrec* [33, 34], in which bindings can be defined with variables out of scope when reading sequentially. By allowing this behavior, mutually recursive bindings are made possible and thus more natural implementations are available. Haskell's vanilla `let` already acts like a `letrec`, and it would be useful to replicate this property to monadic bindings as well.

In the case of the `counter` example, the execution of a side-effect is mandatory to evaluate the values of the bindings, such as `next`, `inc`, `out`, and `zero` (lines 2 to 5). In contrast, the example `countDown` in Section 7.2 has none of its bindings locked by side-effects, e.g, the bindings `f` and `n` have nothing to do with the effect of printing a message on `stdout`. When dealing with the latter of these cases, the usual fixed-point combinator is enough to model its recursion. The former case, however, needs a special kind of recursion, so-called *value recursion* [33].

As we are about to understand on Section 7.4, the use of value recursion to have monadic's bindings with the same convenience of `letrec` will be the key to our improvement on FFACT over FACT. Fundamentally, it will *tie the recursion knot* done in FACT via the complicated implicit recursion mentioned in Section 3.3. In terms of implementation, this is being achieved by the use of the `mfix` construct [5], which is accompanied by a *recursive do* syntax sugar [6], with the caveat of not being able to do shadowing – much like the `let` and `where` clauses in Haskell. In order for a type to be able to use this construct, it should follow specific algebraic laws [33] to then implement the `MonadFix` type class found in `Control.Monad.Fix` [2] package:

```
class (Monad m) => MonadFix m where
  mfix :: (a -> m a) -> m a


createInteg :: CT Double -> CT Integrator
createInteg i = do
```

[2] `Control.Monad.Fix` hackage documentation.

```
  r1 <- liftIO . newIORef $ initialize i
  r2 <- liftIO . newIORef $ initialize i
  let integ = Integrator { initial = i,
                           cache = r1,
                           computation  = r2 }
      z = do
        ps <- ask
        v <- liftIO $ readIORef (computation integ)
        local (const ps) v
  y <- memo interpolate z
  liftIO $ writeIORef (cache integ) y
  return integ

readInteg :: Integrator -> CT Double
readInteg = join . liftIO . readIORef . cache

updateInteg :: Integrator -> CT Double -> CT ()
updateInteg integ diff = do
  let i = initial integ
      z = do
        ps <- ask
        let f =
              case (method $ solver ps) of
                Euler -> integEuler
                RungeKutta2 -> integRK2
                RungeKutta4 -> integRK4
        y <- liftIO $ readIORef (cache integ)
        f diff i y
  liftIO $ writeIORef (computation integ) z
```

7.4 Tweak IV: Fixing FACT

The primitive `createInteg` is the one that first establishes what we are calling implicit recursion. This scheme can be better perceived by Figure 7.3. The mutable references `cache` and `computation` target memory regions that *reference each other*. When creating an integrator, the field `computation` references a memory region that holds a continuous machine which yields the initial value for the ODE. Later on, the primitive `updateInteg` will mutate this region to use the proper differential equation of interest. On the other hand, the field `cache` references a continuous machine called `z`. This machine will take care of both interpolation and memoization strategies. Notice that the `z` machine *reads* from the memory region that `computation` references. The red arrows in the figure should help with this visualization: these `IORef` fields are indirectly interacting with each other;

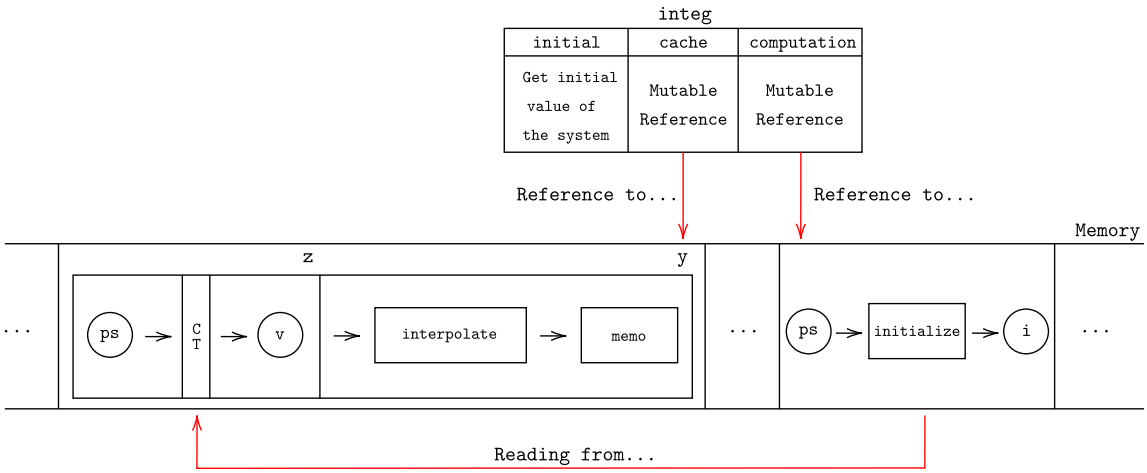


Figure 7.3: Diagram of `createInteg` primitive for intuition.

a change in one of them affects the other. This process, however, is completely hidden from an usability's point of view – the user of the FACT will not and *should* not interact with this behavior.

Both remaining primitives work in a simpler manner. When retriving a state variable, via the primitive `readInteg`, the function `readInteg` hooks onto the layout created by the previous primitive and exposes its value when using FACT. The final step, plugging the differential equations in integrators, is simply a writing operation on the reference of `computation` with the correct continuous machine; replacing the one with the initial value settled when we created the integrator initially. Notice that the differential equation needs to interact with what is being referenced via the `cache` field – it will leverage the use of memoization and interpolation using the same `z` machine from before.

As previously detailed, these primitives are not only *required* to model with FACT, but the *order* in which the user needs to write them also is due to highly imperative nature of this implementation. Differential equations of interest use state variables, provided by `readInteg`, and the only way FACT allows one to retrive those is via the use of an integrator. In order to have one of those, the user is obligated to first invoke `createInteg`, although what we are really interested in – transposing mathematical descriptions of differential equations to software – can only happens at step `updateInteg`. Further, because a model description in FACT uses a `do`-notation block, the *sequential* behavior imposed by bind, a requirement for the `Monad` type class, cannot be transpassed – regardless if doing so would be more *natural* from a modeling perspective. Hence, FACT solved this problem by introducing the `Integrator` data type with its mutable reference fields. This way allowed the use of good translation of differential equations into Haskell, via the continuous machines, with the trade-off of exposing and forcing the use the integrator's

primitives at the user level.

In contrast, FFACT's use of `mdo` removes this existing limitation in FACT's `do`. With `letrec`'s flexibility via the type class `MonadFix`, one can use order-independent bindings, which may need effects to be defined, as one would do with a piece of paper. From an usability's perspective, value recursion is closing the gap between the informal notion of bindings mathematicians have and the more restricted notion programmers have, which usually vary from programming language to programming language and each one comes with its own different quirks and solutions to not incur into scoping issues.

With this context in mind, below we present the definition of `mfix` in the `MonadFix` type class for the `ReaderT` type, the underlying type of the `CT` type alias, present in the `Control.Monad.Trans.Reader` [3] package:

```
instance (MonadFix m) => MonadFix (ReaderT r m) where
  mfix f = ReaderT $ \ r -> mfix $ \ a -> runReaderT (f a) r
```

Because a continuous machine has embedded `IO` in its data type definition, the monadic fix implementation of a continuous machine monad, here presented by the `ReaderT`, also leverages the instances of its internal, e.g., `IO`, for the same type class. Further, because a continuous machine type is a type alias to `ReaderT`, which is a generalization to the Reader (or Environment) monad, it can be shown that continuous machine's implementation of `MonadFix` satifies the laws mentioned at the end of Section 7.3 [33]. With access to `mdo` syntax sugar, a new function, called `integ`, can therefore be implemented to perform what FACT's integrator was accomplishing instead:

```
1  integ :: CT Double -> CT Double -> CT (CT Double)
2  integ diff i =
3    mdo y <- memo interpolate z
4        z <- do ps <- ask
5                let f =
6                      case (method $ solver ps) of
7                        Euler -> integEuler
8                        RungeKutta2 -> integRK2
9                        RungeKutta4 -> integRK4
10               pure $ f diff i y
11       return y
```

This new function received the differential equation of interest, named `diff`, and the initial condition of the simulation, identified as `i`, on line 2. Interpolation and memoization requirements from FACT are being maintained, as shown on line 3. Lines 3 to 6 demonstrate the use case for FFACT's `mdo`. A continuous machine created by the memoization function (line 3), `y`, uses another continuous machine, `z`, yet to be defined. This continuous machine, defined on line 4, retrieves the numerical method chosen by a value of

[3] `Control.Monad.Trans.Reader` hackage documentation.

type `Parameters`, via the function `f`. The outcome of the function `integ` is the outcome of running the simulation of interest in the context of memoization and interpolation. As a final note, just as with `fix`, there is a need for the function being applied to the combinator to terminate the recursive process: this is being done via the function `memo` within the `integ` function.

Finally, the Lorenz Attractor example is rewritten as the following:

```
1 lorenzModel :: Model Vector
2 lorenzModel = mdo
3    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    return $ sequence [x, y, z]

7 lorenzSystem = runCT lorenzModel 100 lorenzSolver
```

Not surprisingly, the results of this new approach using the monadic fixed-point combinator are very similar to the performance metrics depicted in Chapter 6, *Caching the Speed Pill* — indicating that we are *not* trading performance for a gain in conciseness. Figure 7.4 shows the new results.

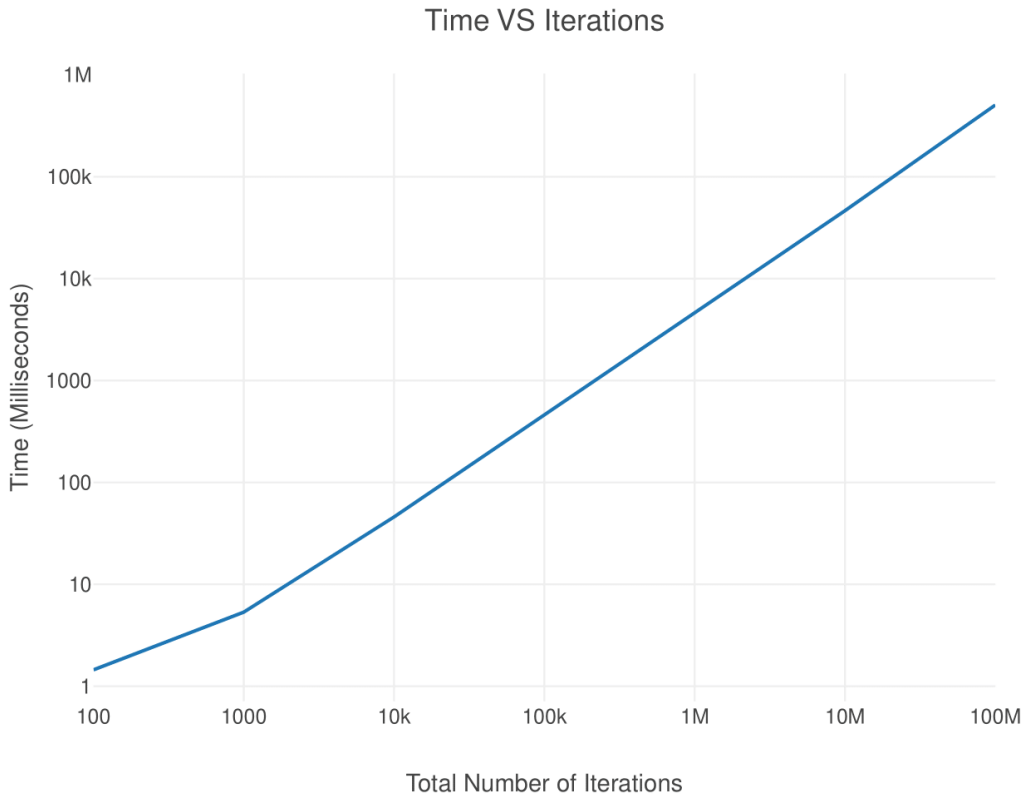


Figure 7.4: Results of FFACT are similar to the final version of FACT..

7.5 Examples and Comparisons

In order to assess how *concise* model can be in FFACT, in comparison with the mathematical descriptions of the models, we present comparisons between this dissertation's proposed implementation and the same example in SimulinkSimulink [4], Matlab [5], Mathematica [6], and `Yampa` [7]. It is worth noting that the last one, `Yampa`, is also implemented in Haskell as a HEDSL. In each pair of comparisons both conciseness and differences will be considered when implementing the Lorenz Attractor model. Ideally, a system's description should contain the *least* amount of notation noise and artifacts to his mathematical counterpart. It is worth noting that these examples only show *the system's description*, i.e., the *drivers* of the simulations are being omitted when not necessary to describe the system.

Figure 7.5 depicts a side-by-side comparison between FFACT and Simulink. The Haskell HEDSL specifies a model in text format, whilst Simulink is a visual tool — you

[4]Simulink documentation.
[5]Matlab documentation.
[6]Mathematica documentation.
[7]Yampa hackage documentation.

draw a diagram that represents the system, including the feedback loop of integrators, something exposed in Simulink. A visual tool can be useful for educational purposes, and a pictorial version of FFACT could be made by an external tool that from a diagram it compiles down to the correspondent Haskell code of the HEDSL.

```
lorenzModel = mdo
  x <- integ (sigma * (y - x)) 1.0
  y <- integ (x * (rho - z) - y) 1.0
  z <- integ (x * y - beta * z) 1.0
  let sigma = 10.0
      rho = 28.0
      beta = 8.0 / 3.0
  return $ sequence [x, y, z]
```

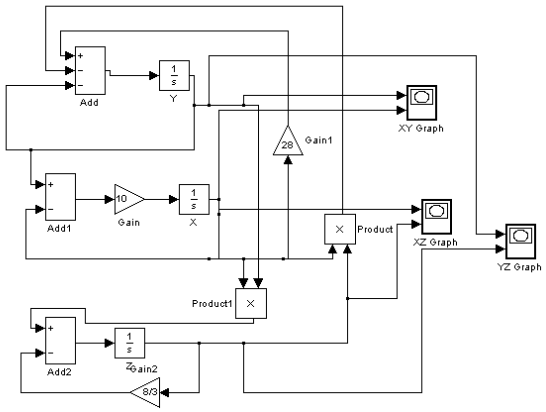


Figure 7.5: Comparison of the Lorenz Attractor Model between FFACT and a Simulink implementation [7].

Figure 7.6 shows a comparison between FFACT and Matlab. The main differetiating factor between the two implementations is in Matlab the system, constructed via a separate lambda function (named `f` in the example), has the initial conditions of the system at t_0 only added when calling the *driver* of the simulation — the call of the `ode45` function. In FFACT, the interval for the simulation and which numerical method will be used are completely separate of the system's description; a *model*. Furthermore, Matlab's description of the system introduces some notation noise via the use of `vars`, exposing implementation details to the system's designer.

```
lorenzModel = mdo
  x <- integ (sigma * (y - x)) 1.0
  y <- integ (x * (rho - z) - y) 1.0
  z <- integ (x * y - beta * z) 1.0
  let sigma = 10.0
      rho = 28.0
      beta = 8.0 / 3.0
  return $ sequence [x, y, z]
```

```
sigma = 10;
beta = 8/3;
rho = 28;
f = @(t,vars)
    [sigma*(vars(2) - vars(1));
     vars(1)*(rho - vars(3)) - vars(2);
     vars(1)*vars(2) - beta*vars(3)];
[t,vars] = ode45(f,[0 50],[1 1 1]);
```

Figure 7.6: Comparison of the Lorenz Attractor Model between FFACT and a Matlab implementation.

The next comparison is between Mathematica and FFACT, as depicted in Figure 7.7. Differently than Matlab, Mathematica uses the state variables' names when describing the system. However, just like with Matlab, the initial conditions of the system are only provided when calling the driver of the simulation. Moreover, there's significant noise in Mathematica's version in comparison to FFACT's version.

```
lorenzModel = mdo
  x <- integ (sigma * (y - x)) 1.0
  y <- integ (x * (rho - z) - y) 1.0
  z <- integ (x * y - beta * z) 1.0
  let sigma = 10.0
      rho = 28.0
      beta = 8.0 / 3.0
  return $ sequence [x, y, z]
```

```
lorenzModel = NonlinearStateSpaceModel[
   {{sigma (y - x),
     x (rho - z) - y,
     x y - beta z}, {}},
   {x, y, z},
   {sigma, rho, beta}];
soln[t_] = StateResponse[
   {lorenzModel, {1, 1, 1}},
   {10, 28, 8/3},
   {t, 0, 50}];
```

Figure 7.7: Comparison of the Lorenz Attractor Model between FFACT and a Mathematica implementation.

Finally, Figure 7.8 contrasts FFACT with `Yampa`, another HEDSL for time modeling and simulation. Although `Yampa` is more powerful and expressive than FFACT — `Yampa` can accomodate hybrid simulations with both *discrete* and *continuous* time modeling — its approach introduces some noise in the Lorenz Attractor model. The introduction of `proc`, `pre`, `>>>`, `imIntegral`, and `-<` all introduce extra burden on the system's designer to describe the system. After learning about `proc-notation` [35] and Arrows [8], one can describe more complex systems in Yampa.

```
lorenzModel = mdo
  x <- integ (sigma * (y - x)) 1.0
  y <- integ (x * (rho - z) - y) 1.0
  z <- integ (x * y - beta * z) 1.0
  let sigma = 10.0
      rho = 28.0
      beta = 8.0 / 3.0
  return $ sequence [x, y, z]
```

```
lorenzModel = proc () -> do
  rec x <- pre >>> imIntegral 1.0 -< sigma*(y - x)
      y <- pre >>> imIntegral 1.0 -< x*(rho - z) - y
      z <- pre >>> imIntegral 1.0 -< (x*y) - (beta*z)
      let sigma = 10.0
          rho = 28.0
          beta = 8.0 / 3.0
  returnA -< (x, y, z)
```

Figure 7.8: Comparison of the Lorenz Attractor Model between FFACT and a Yampa implementation.

The function `integ` alone in FFACT ties the recursion knot previously done via the `computation` and `cache` fields from the original integrator data type in FACT. Hence, a lot of implementation noise of the DSL is kept away from the user — the designer of the system — when using FFACT. With this Chapter, we addressed the third and final concerned explained in Chapter 1, *Introduction*. 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.

[8] Arrows hackage documentation.

Chapter 8

Conclusion

Our motivation was to mitigate the high difficulty of modeling arbitrary closed feedback loops, using the DSL proposed by Medeiros et al. [1]. In their DSL, time-varying signals are abstracted by a function data type that updates the state of the system, and the topology of the system can only be described via a set of composition operators instead of dealing with the signals explicitly. In this work, we tackled this by proposing `FACT`: a reimplementation of the DSL based on a new implementation abstraction, called `CT`, whilst maintaining GPAC as the formal inspiration. This new data type holds the state of the system indirectly, thus allowing the user of the DSL to directly manipulate the signals when describing a system of equations. The guiding example used throughout this work, the Lorenz Attractor in Figure 1.1, is an example of a system with feedback loops that the former DSL could not express. Despite solving this expressivenness problem, `FACT` introduced an abstraction leaking, exposing to the users of the DSL internal implementation details. We solved this issue leveraging the monadic fixed-point combinator, resulting `FFACT` and thus improving the notation and usability.

Chapter 2 established the foundation of the implementation, introducing functional programming (FP) concepts and the necessary types to model continuous time simulation — with continuous time machines (`CT`) being the main type. Chapter 3 extended its power via the implementation of typeclasses to add functionality for the `CT` type, such as binary operations and numerical representation. Further, it also introduced the `Integrator`, a CRUD-like interface for it, as well as the available numerical methods for simulation. As a follow-up, Chapter 4 raised intuition and practical understanding of `FACT` via a detailed walkthrough of an example. Chapter 5 explained and fixed the mix between different domains in the simulation, e.g., continuous time, discrete time and iterations, via an additional linear interpolation when executing a model. Chapter 6 addressed performance concerns via a memoization strategy. Finally, Chapter 7 introduced the fixed-point combinator and its monadic counterpart in order to increase conciseness of the HEDSL,

bringing more familiarity to systems designers experienced with the mathematical descriptions of their systems of interest. This notation enhancement is the defining feature between FACT and FFACT.

8.1 Future Work

The following subsections describe the three main areas for future improvements in `FFACT`: formalism, possible extensions, and code refactoring.

8.1.1 Formalism

One of the main concerns is the *correctness* of `FACT` 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 — no model checking tools were used for its 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 for this work.

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 — something that can be achieved with *dependently typed* languages and/or tools such as Rocq, PVS, Agda, Idris and Lean. In Haskell, this burden is on the developer to manually write down such proofs, a non-explored aspect of this work. Hence, this work can be better understood as a *proof of concept* for FFACT, and one potential improvement would be to port it to more powerful and specialized programming languages, such as the ones mentioned earlier. Because FP is highly encouraged in those languages, such port would not be a major roadblock. Thus, these tools would assure a solid mappping between the mathematical the description of the problem, GPAC's specification and FFACT's implementation, including the use of the chosen typeclasses.

8.1.2 Extensions

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 universality [25, 26]. The proposed software didn't touch on those enhancements and restricted the set of functions to only algebraic functions. 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.

8.1.3 Refactoring

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. One potential improvement would be to explore an entirely *pure* based approach, 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 the `memo` function acts as a *state* of the numerical solver. Other Haskell abstractions, such as the `ST` monad [1], could be considered for future improvements towards purity. Going even further, given that `FACT` already uses `ReaderT`, a combination of monads could be used to better unify all different behavior – in Haskell, an option would be to use *monad transformers*. For instance, if the reader and state monads, something like the `RWS` monad [2], 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 [3], it might be possible to better express these basic units and take advantage of better performance and convenience that these alternatives provide.

[1] `ST` Monad wiki page.
[2] `RWS` Monad hackage documentation.
[3] Examples of alternative preludes.

Chapter 9

Appendix

9.1 Literate Programming

This dissertation made use of literate programming [1], a concept introduced by Donald Knuth [36]. Hence, this document can be executed using the same source files that the `PDF` is created

This process requires the following dependencies:

- `ghc` - minimum version `9.6.6`
- `pdflatex` - minimum version `3.141592653-2.6-1.40.25`
- `bibtex` - minimum version `0.99d`

The script located in `doc/literate.sh` is responsible to run all literate programming functionalities. The available commands are (all of them need to run within the directory `doc`):

- `./literate colorful` - Generates the `PDF` named `thesisColorful` with the documentation with colorful code.
- `./literate gray` - Generates the `PDF` named `thesisGray` with the documentation with verbatim code. An extra dependency, `lhs2Tex` is necessary to run this subcommand.
- `./literate repl` - Enters the `ghc` REPL [2] with the code available for exploration.
- `./literate compile` - Compiles an executable. Currently, the thesis is set to run the final version of `FACT` (`FFACT`) running the latest iteration of the Lorenz Attractor example, time step of 0.01 with the second-order Runge-Kutta method, with start

[1] Literate Programming.
[2] Read–eval–print loop.

time set to 0 and final time set to 100. All intermadiate values from the three state variables, x, y, and z, are displayed in `stdout`. Failures on commands for specific OSes, such as commands for Windows when running in a Linux machine and vice-versa, should be ignored.

9.2 FFACT's Manual

This is a concise and pragmatic manual on how to create and run simulations using `FFACT`. For a deeper and more detailed description of the internals of the DSL, including a walkthrough via examples, please consult (and generate via `literate.sh`) either `GraduationThesis` (for `FACT`) or `MasterThesis` (for `FFACT`).

9.2.1 Models

A simulation model is defined using `mdo-notation` (check recursive do) to describe a system of *differential equations*. The current version of `FFACT` only supports continuous simulations, i.e., discrete or hybrid simulations are future work. Alongside the equations, one must provide *initial conditions* for each individual equation, such as the following:

```
lorenzModel :: Model [Double]
lorenzModel = mdo
   x <- integ (sigma * (y - x)) 1.0
   y <- integ (x * (rho - z) - y) 1.0
   z <- integ (x * y - beta * z) 1.0
   let sigma = 10.0
       rho = 28.0
       beta = 8.0 / 3.0
   return $ sequence [x, y, z]
```

In this example, `lorenzModel` will return the state variables of interest via a list, hence the model having the type `Model [Double]`. Recursive monadic bindings are possible due to `mdo`, which makes the description of a model in code closer to its mathematical counterpart.

9.2.2 Solver

Solver-specific configurations, e.g., which numerical method should be used and with which *time step*, which solver stage should it start with, are configured *separately* from the model and from executing a simulation. This sort of configuration details are set via a separate record, such as the following:

```
lorenzSolver = Solver { dt = 1,
                        method = RungeKutta2,
                        stage = SolverStage 0
                      }
```

Available numerical methods:

- `Euler`
- `RungeKutta2`
- `RungeKutta4`

9.2.3 Simulation

A model and a record with solver's configuration are some of the *arguments* to a *driver function*. A driver function runs the simulation starting from 0 until a provided timestamp (in seconds). Currently, `runCTFinal` outputs the final result of the system at the provided final time and `runCT` outputs a *list* of intermediate values from the start until the provided final time spaced by the time step within the solver's configuration. The type signatures of these functions are the following (`Double` is the final time of choice):

```
runCTFinal :: Model a -> Double -> Solver -> IO a
runCT :: Model a -> Double -> Solver -> IO [a]
```

9.2.4 Interpolation

Both `FACT` and `FFACT` use *linear interpolation* to approximate results in requested timestamps that are not reachable via the chosen time step within the solver's configuration. Driver functions automatically take care of detecting and running interpolations. The type signature of the provided interpolation function (and probably future extensions) is the following:

```
interpolate :: CT Double -> CT Double
```

9.2.5 Caching

Both `FACT` and `FFACT` employ a *memoization strategy* for caching, in order to speed up the simulation execution. Without this, simulations recompute previously computed values multiple times, due to the recursive nature of the numerical methods available. A table is saved in memory with already calculated values, and lookups are done instead of triggering a new computation. The type signature of the provided memoization function (and probably future extensions) is the following:

```
memo :: UMemo e => (CT e -> CT e) -> CT e -> CT (CT e)
```

The typeclass `UMemo` is provided custom typeclass.

9.2.6 Example

Lorenz Attractor complete example:

```
lorenzModel :: Model [Double]
lorenzModel = mdo
   x <- integ (sigma * (y - x)) 1.0
   y <- integ (x * (rho - z) - y) 1.0
   z <- integ (x * y - beta * z) 1.0
   let sigma = 10.0
       rho = 28.0
       beta = 8.0 / 3.0
   return $ sequence [x, y, z]

lorenzSolver = Solver { dt = 1,
                        method = RungeKutta2,
                        stage = SolverStage 0
                      }

lorenz = runCTFinal lorenzModel 100 lorenzSolver
```

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, 4, 5, 6, 7, 9, 10, 78

[2] Lemos, Eduardo: *Continuous time modeling made functional: solving differential equations with haskell.* 2022. `https://bdm.unb.br/handle/10483/32536`. x, 4, 6, 7, 16, 20

[3] Medeiros, Edil, Eduardo Peixoto, and Eduardo Lemos: *Fact: A domain-specific language based on a functional algebra for continuous time modeling.* In *2023 Winter Simulation Conference (WSC)*, pages 2650–2661, 2023. x, 6, 7

[4] Medeiros, José E. G.: *Unscented transform framework for quantization modeling in data conversion systems.* 2017. xii, 43

[5] Erkök, Levent and John Launchbury: *Recursive monadic bindings.* SIGPLAN Not., 35(9):174–185, sep 2000, ISSN 0362-1340. `https://doi.org/10.1145/357766.351257`. xiii, 69, 70

[6] Erkök, Levent and John Launchbury: *A recursive do for haskell.* In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 29–37, New York, NY, USA, 2002. Association for Computing Machinery, ISBN 1581136056. `https://doi.org/10.1145/581690.581693`. xiii, 69, 70

[7] Ekhande, Rahul: *Chaotic signal for signal masking in digital communications.* IOSR Journal of Engineering, 4, January 2014. xiii, 76

[8] Derler, Patricia, Edward A. Lee, and Albert Sangiovanni Vincentelli: *Modeling cyber-physical systems.* Proceedings of the IEEE, 100(1):13–28, 2012. 1, 2, 5, 16, 23

[9] 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

[10] Lee, Edward A.: *Fundamental limits of cyber-physical systems modeling.* ACM Transactions on Cyber-Physical Systems, 1(1), 2016. `https://doi.org/10.1145/2912149`. 1, 2

[11] Lee, Edward A.: *Constructive models of discrete and continuous physical phenomena.* IEEE Access, 2:797–821, 2014. 1

[12] Ungureanu, George, Jose E. G. de Medeiros, and Ingo Sander: *Bridging discrete and continuous time models with atoms.* In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 277–280, 2018. 1

[13] 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. 1, 2, 6

[14] Ungureanu, George, Edil G. de Medeiros, Timmy Sundstrom, Ingemar Soderquist, Anders Ahlander, and Ingo Sander: *Forsyde-atom: Taming complexity in cyber physical system design with layers.* ACM Transactions on Embedded Computing Systems, 20(2):1–27, 2021. 1

[15] Graça, Daniel and José Costa: *Analog computers and recursive functions over the reals.* Journal of Complexity, 19:644–664, October 2003. 1, 4, 9, 33

[16] Shannon, Claude E: *Mathematical theory of the differential analyzer.* Journal of Mathematics and Physics, 20(1-4):337–354, 1941. 1, 4, 8, 9

[17] Bush, Vannevar: *The differential analyzer. a new machine for solving differential equations.* Journal of the Franklin Institute, 212(4):447–488, 1931. 1

[18] 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

[19] Churchill, Winston: *HC Deb 28 October 1943 (House Of Commons Rebuilding)*, volume 393 of *5th*, page 403. Bantam, London, 1943. https://hansard.parliament.uk/commons/1943-10-28/debates/4388c736-7e25-4a7e-92d8-eccb751c4f56/HouseOfCommonsRebuilding. 1

[20] 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, 16

[21] Chupin, Guerric and Henrik Nilsson: *Functional reactive programming, restated.* In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, pages 1–14, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450372497. https://doi.org/10.1145/3354166.3354172. 2

[22] Perez, Ivan: *The beauty and elegance of functional reactive animation.* In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2023, page 8–20, New York, NY, USA, 2023. Association for Computing Machinery, ISBN 9798400702952. https://doi.org/10.1145/3609023.3609806. 2

[23] Rovers, K.C.: *Functional model-based design of embedded systems with UniTi.* Phd thesis - research ut, graduation ut, University of Twente, Netherlands, December 2011, ISBN 978-90-365-3294-5. eemcs-eprint-21156 http://eprints.ewi.utwente.nl/21156. 2

[24] 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, 6

[25] Graça, Daniel: *Some recent developments on shannon's general purpose analog computer*. Math. Log. Q., 50:473–485, September 2004. 4, 8, 9, 27, 80

[26] Bournez, Olivier, Daniel Graça, and Amaury Pouly: *On the functions generated by the general purpose analog computer*. Information and Computation, 257, January 2016. 4, 8, 80

[27] Backus, John: *Can programming be liberated from the von neumann style? a functional style and its algebra of programs*. Commun. ACM, 21(8):613–641, aug 1978, ISSN 0001-0782. https://doi.org/10.1145/359576.359579. 5

[28] Landin, P. J.: *The next 700 programming languages*. Communications of the ACM, 9(3):157–166, 1966. https://doi.org/10.1145/365230.365257. 6

[29] Wadler, Philip and Stephen Blott: *How to make ad-hoc polymorphism less ad hoc*. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. Association for Computing Machinery. 12

[30] Tennent, R.D.: *Semantics of Programming Languages*. PHI series in computer science. Prentice Hall, 1991, ISBN 9780138055998. https://books.google.com.br/books?id=K7N7QgAACAAJ. 67

[31] Bryant, Victor: *Metric Spaces: Iteration and Application*. Cambridge University Press, 1985. 67

[32] Stoy, Joseph E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977, ISBN 0262191474. 67

[33] Erkök, Levent: *Value recursion in monadic computations*. PhD thesis, 2002, ISBN 0493822941. AAI3063791. 68, 70, 73

[34] Adams, N. I., D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson: *Revised5 report on the algorithmic language scheme*. SIGPLAN Not., 33(9):26–76, sep 1998, ISSN 0362-1340. https://doi.org/10.1145/290229.290234. 70

[35] Perez, Ivan: *The beauty and elegance of functional reactive animation*. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, FARM 2023, page 8–20, New York, NY, USA, 2023. Association for Computing Machinery, ISBN 9798400702952. https://doi.org/10.1145/3609023.3609806. 77

[36] Knuth, Donald E.: *Literate programming.* Center for the Study of Language and Information, USA, 1992, ISBN 0937073806. 81