Efficient implementations of expressive modelling languages

Guerric Chupin

September 2021

Thesis submitted to the University of Nottingham for the degree of Doctor of Philosophy

Abstract

This thesis is concerned with modelling languages aimed at assisting with modelling and simulation of systems described in terms of differential equations. These languages can be split into two classes: causal languages, where models are expressed using directed equations; and non-causal languages, where models are expressed using undirected equations.

This thesis focuses on two related paradigms: Functional Reactive Programming (FRP) and Functional Hybrid Modelling (FHM). FRP is an approach to programming causal time-aware applications that has successfully been used in causal modelling applications; while FHM is an approach to programming non-causal modelling applications. However, both are built on similar principles, namely, the treatment of models as first-class entities, allowing for models to be parametrised by other models or computed at runtime; and support for structurally dynamic models, whose behaviour can change during the simulation. This makes FRP and FHM particularly flexible and expressive approaches to modelling, especially compared to other mainstream languages. Because of their highly expressive and flexible nature, providing efficient implementations of these languages is a challenge. This thesis explores novel implementation techniques aimed at improving the performance of existing implementations of FRP and FHM, and other expressive modelling languages built on similar ideas.

In the setting of FRP, this thesis proposes a novel embedded FRP library that uses the implementation approach of synchronous dataflow languages. This allows for significant performance improvement by better handling of the reactive network's topology, which represents a large portion of the runtime in current implementations, especially for applications that make heavy use of continuously varying values, such as modelling applications.

In the setting of FHM, this thesis presents the modular compilation of a language based on FHM. Due to inherent difficulties with the simulation of systems of undirected equations, previous implementations of FHM and similarly expressive languages were either interpreted or generated code on the fly using just-in-time compilation, two techniques which have runtime overhead over ahead-of-time compilation. This thesis presents a new method for generating code for equation systems which allows for the separate compilation of FHM models.

Compared with current approaches to FRP and FHM implementation, there is greater commonality between the implementation approaches described here, suggesting a possible way forward towards a future non-causal modelling language supporting FRP-like features, resulting in an even more expressive modelling language.

Acknowledgment

I would like to thank my supervisor, Henrik Nilsson, for his help and guidance throughout this PhD. I'm particularly grateful for his constant enthusiasm and optimism, which never failed to lift my spirits when I needed it.

Since I first came to Nottingham as an intern in 2016, the FP lab has been an amazing environment to work in. I would like to thank all its members for making it such a nice place.

Andy McKeown deserves special thanks for the way he helped me through the end of my PhD. He has been an amazing person to be with through 2020 and 2021 and he has made these two (not so great) years so much better for me. He also helped me proofread some of my work, including this thesis.

Je remercie Olivier Nicole, pour son soutien à travers nos conversations régulières pendant ces 3 années et les multiples relectures d'articles que je lui ai fait subir, parfois jusque bien trop tard le soir.

Je remercie enfin ma famille, support sans faille à travers mon parcours scolaire et académique et sans qui je ne serais pas là.

Contents

List of Figures

Acronyms

ADT Algebraic data type

AST Abstract syntax tree

BDF Backward-Difference Formula

CCA Causal Commutative Arrow

CSE Common Subexpression Elimination

DAE Differential Algebraic Equation

DSL Domain-Specific Language

EDSL Embedded Domain-Specific Language

EOO Equation-based object-oriented

FHM Functional Hybrid Modelling

FRP Functional Reactive Programming

GADT Generalized Algebraic Data Types

GHC Glasgow Haskell Compiler

GUI Graphical User Interface

IIR Intermediate Imperative Representation

IVP Initial Value Problem

JIT Just-in-time

MCL Model Composition Language

MKL Modelling Kernel Language

ODE Ordinary Differential Equation

SFRP Scalable FRP

SSA Static Single Assignment

UODE Underlying Ordinary Differential Equation

Introduction

1

Differential equations are a ubiquitous tool for modelling and understanding systems arising in many fields of science and engineering such as mechanics [6], thermodynamics [104], chemistry [69], economics [18], etc. Converting a set of equations into useful simulation results is an involved task, generally requiring either writing custom simulation code or making use of a dedicated numerical solver. Either way, this often implies translating the equation system to low-level imperative code. Producing this code is tedious and error-prone [73] and requires extensive knowledge from the modeller [37] in addition to any domain-specific knowledge needed to produce the equation system in the first place. The resulting code is very remote from the equation system it was derived from, making sharing and reusability difficult.

Modelling languages aim to bridge these gaps by taking a declarative approach to the problem. The modeller expresses the model in terms of differential equations directly, thus focusing on *what* the model is rather than *how* it is simulated, which is left to the implementation. Further, by providing abstractions to help reusability and clarity, modelling languages can greatly ease the modelling process, particularly for large systems. In this thesis, modelling languages specifically refer to languages aimed at assisting with problems that can be formulated in terms of differential equations. These types of modelling languages can be separated into two categories. The first is that of *causal* languages, in which the model is expressed in terms of *directed* differential equations, such as Ordinary Differential Equations (ODE). The causality of the equation, that is, which variable is known and unknown, is fixed and known at the time the equation is written. Prominent representative of this category include SIMULINK [96], PTOLEMY [53] or ZÉLUS [20]. The second category is that of *non-causal* languages, in which the model is expressed in terms of *undirected* differential equations, such as Differential Algebraic Equations (DAE). In this approach, a differential equation is treated as a constraint on the quantities it relates. This gives greater freedom to the modeller, both when writing an equation and when using it, since it is left to the compiler to discover how to use an equation to produce a solution. MODELICA [62] and DYMOLA [56] are the main industrial representatives of this approach.

1.1 — Expressive modelling languages

Expressivity is a somewhat fuzzy concept. In general, it refers either to a theoretical ability of a language to express a particular concept; or

6. Archibald et al., 'The History of Differential Equations, 1670–1950'. 2004

104. Narasimhan, 'Fourier's Heat Conduction Equation'. 1999

69. Gorban et al., 'Three Waves of Chemical Dynamics'. 2015

18. Black et al., 'The Pricing of Options and Corporate Liabilities'. 1973

73. Hindmarsh et al., *Example Programs for IDA v5.7.0*. 2021

37. Cellier et al., *Continuous System Simulation*. 2006

96. Mathworks, *Simulation and Model-Based Design*. 2020

53. Eker et al., 'Taming Heterogeneity - the Ptolemy Approach'. 2003

20. Bourke et al., 'Zélus, a Synchronous Language with ODEs'. 2013

62. Fritzson et al., 'Modelica — A Unified Object-Oriented Language for System Modeling and Simulation'. 1998

56. Elmqvist et al., 'Object-Oriented Modeling of Hybrid Systems'. 1993

to the ease with which one can express a particular concept. This thesis provides a specific definition of expressivity in the context of modelling languages which covers both of these aspects. It considers an expressive modelling language to be one in which (1) models are *first-class* objects and (2) models can be *structurally dynamic*.

Models being first-class means that they are regular values of the language. As such, they can be used as parameters to functions (and thus parametrise other models), stored in data structures and more generally combined programmatically, enabling a form of *higher-order modelling* (by analogy with higher-order functions in a functional language). This facilitates the modelling of large models, by allowing the modeller to use the language to compute models; and can help with modularity.

A model is structurally dynamic if the equation system that describes it can change at discrete points in time. Such models are special cases of *hybrid* models, which more generally describe models which mix continuous dynamics (defined as solutions to differential equations) with discrete behaviours. In a language supporting first-class models and structural dynamism, it becomes possible to compute new parts of the model at simulation runtime based on simulation results. This means that the number of *modes* of a model, that is the number of all its possible structural configurations, can be (uncountably) infinite[a].

Few modelling languages support first-class and structurally dynamic models, although many support some aspects of one or the other. For causal modelling languages, support for hybrid models is commonplace, but support for first-class models is not. Simulink [96], the most prominent representative of causal modelling languages, does not support them. Zélus [20] has some support for it and its predecessor, the synchronous language Lucid Synchrone [124] was in a sense built around the goal of supporting higher-order models.

In the case of non-causal modelling languages, until relatively recently, support for hybrid models was not common. This is in great part due to difficulties inherent to simulating non-causal hybrid models that will become clear later during the thesis. It is still only supported in a limited way in the most prominent non-causal language, Modelica [103], which also does not have support for first-class models. However, recent academic languages, such as Modia, mcl or Modelyze [25, 57, 76], are getting support for first-class models and structural dynamism. Earlier attempts were also made to patch the shortcomings of existing mainstream languages. For instance mkl [24], the predecessor to Modelyze, investigated the possibility of non-causal language with first-class models but without structural dynamism; while Sol [154] investigated the possibility for a non-causal language supporting hybrid models but did not investigate notions of first-class

[a] At a given point during the simulation, the structural configuration of a model is always known. However, its future configuration may depend on arbitrary computations using arbitrary inputs, such as simulation results for the current mode or user inputs, which can result in arbitrarily many modes depending on the result of the program that computes the new structural configuration.

96. Mathworks, *Simulation and Model-Based Design*. 2020

20. Bourke et al., 'Zélus, a Synchronous Language with ODEs'. 2013

124. Pouzet, *Lucid Synchrone*. 2006

103. Modelica Association, *Modelica Language Specification*. 2021

25. Broman, 'Gradually Typed Symbolic Expressions'. 2017 — 57. Elmqvist, 'Systems Modeling and Programming in a Unified Environment Based on Julia'. 2016 — 76. Höger, 'Compiling Modelica'. 2018

24. Broman, 'Meta-Languages and Semantics for Equation-Based Modeling and Simulation'. 2010

154. Zimmer, 'Equation-Based Modeling of Variable-Structure Systems'. 2010

models. While some of these languages are implemented using standalone interpreters or compilers, some of them like Modia are also implemented as embedded languages in a general-purpose language (Julia [16] in this instance). This has the advantage of simplifying some of the aspects of the implementation by allowing the embedding to reuse already implemented constructs seamlessly, such as function calls, pattern-matching or function closures, if the host language supports them [b].

This thesis focuses on two alternative approaches for designing expressive time-aware languages, which are built around the support for first-class structurally dynamic models; namely Functional Reactive Programming (FRP) [54, 151] and Functional Hybrid Modelling (FHM) [112]. Both approaches rely on the embedding of a Domain-Specific Language (DSL) (causal in the case of FRP, non-causal in the case of FHM) in a functional programming language. Entities of the DSL are first-class values of the host language and switching constructs are provided to enable parts of a model to change its behaviour dynamically during the simulation.

However, expressivity often comes at the cost of performance. In the case of FRP, current implementations suffer from high interpretative overhead, even for models that are simple enough to be translatable into less expressive, but more efficient, languages. In the case of FHM, the difficulties associated with the implementation are different and inherent to the difficulties of simulating a system of non-causal equations while supporting structural dynamism.

Providing efficient implementations for these paradigms would allow modellers to 'have their cake and eat it (quickly) too': being able to write models using expressive languages, while still enjoying fast simulation. This would be particularly desirable at least when writing models that can be equivalently expressed using less expressive languages. The question this work aims to answer is whether this is possible, that is:

> *Is it possible to provide efficient implementations for expressive languages that show similar performance characteristics to well-known implementations existing for less expressive languages?*

This thesis claims that the answer to this question is yes. To back this claim, it explores novel interpretation and compilation approaches in an attempt to improve the performance one can expect from the FRP and FHM paradigms. The proposed techniques share a lot of commonality, especially in comparison to existing implementations of FRP and FHM. This paves the way for future development at the intersection of these approaches. For instance in the form of an FRP-like DSL with better support for modelling applications, e.g. by being better integ-

[16] Bezanson et al., 'Julia: A Fresh Approach to Numerical Computing'. 2017

[b] There may be good reasons to not support these features. Structural dynamism, understood in its general sense as the ability to compute models on the fly means that it is not possible to give some guarantees that may be interesting in some contexts. For instance, Simulink can be used to generate code that can be used in a controller or an embedded system. Limiting the expressivity of the language can be used to ensure that the code executes in bounded time or in bounded memory. Some languages, such as Zélus also aim to provide additional static guarantees on the validity of the model, which may not be possible in a more expressive setting.

[54] Elliott et al., 'Functional Reactive Animation'. 1997 — [151] Wan et al., 'Functional Reactive Programming from First Principles'. 2000

[112] Nilsson et al., 'Functional Hybrid Modeling'. 2003

rated with precise differential equation solvers; or in the form of a non-causal language with FRP-like features, combining the strong points of both modelling approaches in a unified, highly expressive and flexible modelling language. This work relies on benchmarks to evaluate the validity of its claims. The benchmarks are used to quantitatively compare the proposed implementation techniques for FRP and FHM with existing ones.

This thesis is split into two broadly independent parts. The first part is dedicated to the implementation of an efficient and scalable FRP library, which is more suited for modelling applications. The second part is dedicated to the implementation of a fully modular ahead-of-time compiler for a language based on FHM. The next sections go over the motivation for the work done in each of these parts, as well as the new contributions they make.

1.2 — Part I: causal modelling with Functional Reactive Programming

In this thesis, a reactive system is understood as a program that reacts to time-varying inputs received from an environment by producing time-varying outputs, sometimes called *reactions*. These outputs may then influence their environment and, therefore, future inputs [a]. FRP is a principled, declarative approach to programming reactive systems, focusing on describing interactions between time-varying values. As a DSL designed for causal reactive programming, it has been successfully used in variety of applications, including video games and musical applications [45, 67, 93, 109]; but also in some causal modelling modelling applications [79, 143]. FRP has been implemented in many different forms across a variety of libraries and languages [4, 28, 141]. It also inspired related approaches such as ReactiveX [128] or the original Elm [48].

This work focuses on a particular approach to FRP called *arrowized* FRP, by way of the library Yampa: a state-of-the-art FRP implementation embedded in Haskell [110]. Yampa is only used as an example of a wider problem with the approach, which this work aims to address.

In arrowized FRP, the reactive network [b] is structured together using arrows [80] and defines how values flow through the network (how they are transported from their producer to their consumer). By analogy with block diagrams, this structure is called *routing* or *wiring* in this thesis. Other authors [115] have also proposed the term *weaving* combinators or Jacquard combinator, in reference to the Jacquard loom [c], to designate the structure of a reactive network.

The efficiency of the wiring is crucial for the performance of an arrowized implementation. Unfortunately, with the current implementation strategies of FRP library this is not the case. In fact, the performance overhead incurred by wiring grows quadratically with the size of

[a] A distinction is sometimes made between *reactive* systems and *interactive* systems, for example by Berry [14]. In a reactive system, the interaction with the environment is governed by the environment itself while in an interactive system, it is governed by the program. For instance, the flight control system in an aeroplane is a reactive program (it must respond as soon as possible), but a web browser is interactive (it displays a page when the page is ready, which may take time). Deciding whether a system falls into one or the other category is possibly subjective and depends on the environment the program interacts with. As such, while this distinction may not be useful in clearly defining whether a system is reactive or interactive, it may help build an intuition.

45. Courtney et al., 'The Yampa Arcade'. 2003 — 67. Giorgidze et al., 'Switched-On Yampa'. 2008 — 93. Mahuet et al., 'Flappy Haskell'. 2015 — 109. Nilsson et al., 'Funky Grooves: Declarative Programming of Full-Fledged Musical Applications'. 2017

79. Hudak et al., 'Arrows, Robots, and Functional Reactive Programming'. 2003 — 143. Thaler et al., 'Pure Functional Epidemics: An Agent-Based Approach'. 2018

4. Apfelmus, *Reactive-Banana*. 2011 — 28. Bünzli, *React, Functional Reactive Programming for OCaml*. 2010 — 141. Söylemez, *Wires*. 2017

128. Microsoft, *ReactiveX*. 2011

48. Czaplicki, 'Elm: Concurrent FRP for Functional GUIs'. 2012

110. Nilsson et al., 'Functional Reactive Programming, Continued'. 2002

the network and, even for small networks, it already represents a significant part of the total cost of running the network.

Current FRP approaches also suffer from conceptual shortcomings. Most notably, due to the fact that their is no distinction between different *kinds* of time-varying values, such as continuously varying values or event signals. This causes both missed optimisation opportunities (all values are represented uniformly, regardless of how they are expected to be used) but also conceptual difficulties, especially when considering an implementation of FRP tailored for simulation and modelling applications. Indeed, such applications could benefit from using more advanced numerical integration techniques than those currently being used by FRP implementations. As demonstrated by work on principled hybrid simulation languages, such as Zélus [10, 20], such distinctions are important when working with numerical solvers to obtain reliable simulation results.

These considerations are in contrast with other approaches at designing causal reactive and modelling languages, notably synchronous dataflow languages [3, 36, 124], of which FRP is a close relative. In such languages, the reactive network is compiled to an imperative representation where time-varying values are represented by imperative references which are carefully updated in sequence to produce results over time [17], thus requiring no runtime wiring. Similar techniques are used for simulation languages such as Simulink. On the other hand, these languages do not offer the same level of flexibility and expressivity that FRP does, in favour of efficient implementation and, most importantly, static performance guarantees.

The work presented in part I shows how, by using an imperative representation very similar to that found in synchronous dataflow languages, the performance of arrowized FRP can be increased manifold by removing all wiring overhead. The implementation relies on a precise distinction, tracked at the type-level, of the kinds of signals. This allows for different representation to be used based on the kind of signals, which allows for specific optimisations for each signal kind based on how it is expected to behave at runtime. It also allows to impose additional restrictions on the reactive network, addressing some of the conceptual shortcomings mentioned before and opening the way to the implementation of an expressive causal modelling language based FRP principles.

The resulting library is called *Scalable FRP (SFRP)* [a]. It has been evaluated on a range of benchmarks designed to test various performance aspects. Additionally, both to check the maturity of this new implementation, and to get an indication of what performance gains one might expect in real applications, the new implementation was used as a mostly drop-in replacement for Yampa in an existing game, Flappy Birds [93]. Although it is already very capable, the library is still in

[b] A reactive program often consists of smaller reactive programs interacting with each other. The term reactive network designates that set of programs and its structure.

80. Hughes, 'Generalising Monads to Arrows'. 2000

115. Oeyen et al., 'Reactive Sorting Networks'. 2020

[c] See `https://en.wikipedia.org/wiki/Jacquard_machine`

10. Benveniste et al., 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. 2017 — 20. Bourke et al., 'Zélus, a Synchronous Language with ODEs'. 2013

3. Amagbégnon, *Implementation of the Data-Flow Synchronous Language SIGNAL*. 1995 — 36. Caspi, *LUSTRE: A Declarative Language for Programming Synchronous Systems*. 1987 — 124. Pouzet, *Lucid Synchrone*. 2006

17. Biernacki et al., 'Clock-Directed Modular Code Generation for Synchronous Data-flow Languages'. 2008

[a] The implementation can be found at `https://gitlab.com/chupin/scalable-frp`.

93. Mahuet, *Flappy Haskell*. 2015

a prototype state and doesn't support some of the more advances features of YAMPA, such as collection-based switching [110]. It also has some additional limitations that are more fundamental, in particular with regards to feedback.

110. Nilsson et al., 'Functional Reactive Programming, Continued'. 2002

1.2.1 — *Organisation*

The first part of this thesis is organised as follows.

Chapter 2 presents the necessary background behind causal modelling languages, with an overview of common integration techniques for directed differential equations. It then introduces Functional Reactive Programming (with a particular focus on arrowized FRP) and shows how it can be used as a simple causal modelling language. It then presents the current implementation of YAMPA as an embedding in HASKELL and its shortcomings.

Chapter 3 presents Scalable FRP (SFRP), a library which addresses the shortcomings outlined in the previous chapter. A detailed implementation is presented, complete with systematic benchmarks showing important performance improvements. The library is also evaluated on small programs where it shows a three-fold performance improvement.

Finally, chapter 4 consists of an overview of related approaches for the design of causal modelling languages and for the implementation of efficient FRP systems. It then offers some concluding remarks on that line of work and directions for future research.

1.2.2 — *Contributions*

The main contributions of this part lie in the design and implementation of the SFRP library. Specifically the following:

- A precise description of FRP networks which is constrained at the type level, necessitating being more explicit about intent thus allowing for efficient implementations.
- An explicit type for routers. The type allows to describe all types of routing occurring in an FRP network explicitly. This allows later to handle routing in an efficient way (§3.2.3).
- The interpretation of SFRP network as a network of imperative references, derived from the compilation techniques used for synchronous dataflow languages (§3.4). The main new contribution here resides in the efficient handling of the switching constructs of FRP (§3.4.3).
- A new approach to embedding arrow-like DSL is introduced (§3.3). This approach is implemented via quasi-quotation [94] to provide a convenient syntax to write SFRP networks, and is based on the existing arrow notation [120] (used by arrowized FRP libraries). A systematic method to desugar the new notation into SFRP networks is presented. Unlike the arrow notation, it does not require the use of higher-order function; instead making use of

94. Mainland, 'Why It's Nice to Be Quoted'. 2007

120. Paterson, 'A New Notation for Arrows'. 2001

explicit explicit router types. This allows for routing to remain explicit in the resulting reactive network. This algorithm also serves as a demonstration that the type of routers introduced in §3.2.3 is indeed sufficient to describe any kind of routing. This work can be used to improve the current desugaring of arrow notation within existing Haskell compilers and potentially benefit other Embedded Domain-Specific Language (edsl) based on arrows.

Note that, while these contributions are presented in the context of designing an frp library tailored for modelling applications, these ideas can be easily adapted for the efficient implementation of frp libraries designed for traditional reactive applications.

This work presented in this part is an extended version of the work presented in the following peer-reviewed publication:

> Guerric Chupin and Henrik Nilsson. 'Functional Reactive Programming, Restated'. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. PPDP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–14. isbn: 978-1-4503-7249-7. doi: 10.1145/3354166.3354172

1.3 — Part II: modular compilation for Functional Hybrid Modelling

Functional Hybrid Modelling (fhm) is an approach aimed at designing expressive non-causal modelling languages. fhm is very close in spirit to frp; indeed it can be viewed as an extension. Being a non-causal modelling language, fhm allows for models to be described as sets of undirected equations, also known as Differential Algebraic Equations (dae). It provides similar switching constructs to frp, leading to a similar level of expressivity and flexibility.

Non-causal modelling languages in general allow for a much more modular approach to modelling. Indeed, where in a causal modelling language, an equation can only be used to solve for one variable, in a non-causal language, the same equation can be used to solve for (in principle) all the variables that appear in it, enabling for a much greater reuse of models.

However, from a compiler implementer's perspective, non-causal models are not modular in the slightest. The simulation of a dae indeed poses inherent difficulties, particularly the *differentiation index* [31]. In general, a non-causal model features constraint equations that restrict the set of possible solutions but cannot readily be used by a numerical solver in computing a solution. Recovering a system suitable for simulation can be done by differentiating some of the equations

[31] Campbell et al., 'The Index of General Nonlinear DAEs'. 1995

of the system. How many times a set of equations needs to be differentiated is the differentiation index of the DAE. The resulting latent equations can then be used for simulation.

There exist efficient algorithms [116, 126] for determining the set of equations to differentiate to obtain a system suitable for simulation. However, they all require a *complete* view of the model being simulated. Thus, before a model is completely assembled, it is not possible to fully determine the equations that must be differentiated for simulation. This is problematic for code generation: because the index of a DAE can be arbitrarily large [129], it is not possible to generate separate code ahead of time for all derivatives that may be needed. Hybrid languages exacerbate the problem as the differentiation index may be different across modes and therefore requires new equations to be differentiated (and code generated) when a structural change occurs.

For the reasons above, hybrid non-causal languages have traditionally been either interpreted [24, 154] or Just-in-time (JIT) compiled [57], including previous implementations of FHM [66]. Other languages that support limited hybrid features such as MODELICA are compiled ahead-of-time but not in a modular way: the equation system representing the fully assembled model is extracted (a process known as flattening) and code is generated afterwards.

This second part of the thesis presents a new implementation of HYDRA, a DSL based on FHM principles. This new implementation is standalone: a compiler has been written both for the modelling language and the functional host, unlike previous implementations of HYDRA [64], and the work presented in the previous part where HASKELL was used as the host language.

In this part, the problem of modular compilation for HYDRA is addressed by exploring ways to generate code capable of computing an *arbitrary* derivative of an expression. The technique considered is called *order-parametric differentiation*: the code generated for an expression is parametrised by the order of the derivative required by the caller and can thus be used to compute any derivative at any point (including the undifferentiated value of the expression, or order zero derivative). The objective is to generate modular code in the usual, programming-language sense of the term. Compilers for programming languages, like C, JAVA or HASKELL, compile the code for a function once and then simply use a symbol to jump into the body of the function when it is being called. This allows separately compiled modules to be joined by a linker, without further compilation as such. The objective is for simulation code for models to be compiled into a function and use the same mechanism when one model is used in another, allowing separately compiled models to be linked in the conventional sense.

116. Pantelides, 'The Consistent Initialization of Differential-Algebraic Systems'. 1988 — 126. Pryce, 'A Simple Structural Analysis Method for DAEs'. 2001

129. Reissig et al., 'Differential–Algebraic Equations of Index 1 May Have an Arbitrarily High Structural Index'. 2000

24. Broman, 'Meta-Languages and Semantics for Equation-Based Modeling and Simulation'. 2010 — 154. Zimmer, 'Equation-Based Modeling of Variable-Structure Systems'. 2010

57. Elmqvist et al., 'Systems Modeling and Programming in a Unified Environment Based on Julia'. 2016

66. Giorgidze et al., 'Mixed-Level Embedding and JIT Compilation for an Iteratively Staged DSL'. 2011

64. Giorgidze et al., 'Embedding a Functional Hybrid Modelling Language in Haskell'. 2011

1.3.1 — *Organisation*

The second part is independent from part I, with the exception of chapter 5. It is organised as follows.

Chapter 5 introduces the additional background needed to simulate systems of undirected equations directly. It builds on the notions introduced in chapter 5 for the simulation of directed equations. It then introduces FHM and the non-causal modelling approach through the new implementation of Hydra presented in this thesis.

Chapter 6 presents a detailed specification of Hydra's surface syntax and type system. It presents a simple simulation scheme based on the translation of Hydra models in terms of flattened systems of undirected equations.

Chapter 7 presents a modular compilation scheme for Hydra. It presents the mathematical background behind order-parametric differentiation and how code for equations can be generated using it. It then shows how this can be used to compile the rest of the Hydra language in a modular way. Finally, this chapter presents benchmarking results to compare code generated by order-parametric differentiation and code generated by traditional means, such as repeated first-order automatic differentiation.

Chapter 8 consists of an overview of related non-causal modelling languages and related approaches to the modular compilation of these languages. It then gives concluding remarks and suggestions for improvements to the approach and to the Hydra language as a whole.

1.3.2 — *Contributions*

The contributions made in this part of the thesis are as follows:

— A new version of the Hydra language, based on FHM principles and its compiler (chapter 6 and 7).

— The notion of order-parametric differentiation and shows how it can be used to compile mathematical expressions to imperative code able to produce the value of any derivative of the original expression (§7.2.2).

— A scheme to compile hybrid models (made of groups of equations and submodels) to low-level code. It introduces simple schemes to keep track of variable ordering and the state of a hybrid model that map naturally to compiled low-level code.

— A performance study of order-parametric code. Order-parametric code is sometime much less efficient than with other approaches. It is expected that the performance figures presented in this thesis can be used as a guide for implementers, for optimisation purposes or for using this technique alongside other existing approaches, such as JIT compilation.

The work presented in this part was, in part, presented in the following peer reviewed publication:

Guerric Chupin and Henrik Nilsson. 'Modular Compilation for a Hybrid Non-Causal Modelling Language'. en. In: *Electronics* 10.7 (Jan. 2021), p. 814. DOI: `10.3390/electronics10070814`

1.4 — Part III: conclusions

This part contains concluding remarks about the thesis and considers avenues for future work at the intersection of the two approaches studied in this thesis.

1.5 — Prerequisites

This thesis assumes some knowledge of typed functional programming in general.

Part I further relies on a good understanding of the Haskell language and some of the more advanced features supported only by the Glasgow Haskell Compiler (GHC), such as Generalized Algebraic Data Types (GADT) or type families [38, 39, 122].

38. Chakravarty et al., 'Associated Type Synonyms'. 2005 — 39. Chakravarty et al., 'Associated Types with Class'. 2005 — 122. Peyton Jones et al., 'Simple Unification-Based Type Inference for GADTs'. 2006

1.6 — Conventions

This thesis uses Lagrange notation for derivatives. Given a time-varying function x, x' denotes its first-order derivative, x'' its second-order derivative, etc. When the number of differentiation is large or unknown, the notation $x^{(n)}$ is used to denote the n-th derivative. It is extended to include the case where n is 0 with $x^{(0)} = x$. When it is unambiguous, the point at which a time-varying function is applied is not specified, hence x may designate either the function itself or the value $x(t)$, for some arbitrary time t. Partial derivatives of a multivariate function are denoted using the usual notation $\frac{\partial f}{\partial x_i}$.

The norm of a vector of x of $\mathbb{R}^n$ will be denoted using $\|x\|$.

Part I

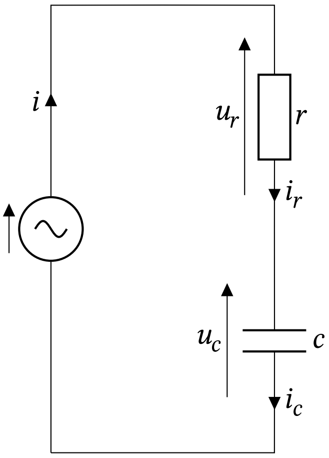
Scalable Functional Reactive Programming

Introduction to causal modelling 2

This chapter serves as an introduction to the basic principles behind causal modelling. §2.1 introduces the concept of Ordinary Differential Equation (ODE) through a simple example and introduces some useful terminology for the rest of the thesis. §2.2 gives a general overview of common methods used for computing an approximated solutions to an ODE, with a particular focus on understanding some of their limits, which have an influence on the design of the language. §2.3 introduces Functional Reactive Programming (FRP), which will be used as an example for a simple causal modelling language. §2.4 presents the implementation of the FRP library YAMPA, which is used as the basis for the work presented in this thesis. It will show some of its shortcomings, which will be addressed in the next chapter.

2.1 — Modelling with differential equations

Figure 2.1: A RC-circuit



Consider the electrical circuit depicted in figure 2.1. It consists of 3 components, a voltage source, a resistor with resistance r and a capacitor of capacitance c; and relates 6 time-varying functions: u, i, u_r, i_r, u_c and i_c, which are the voltages and currents flowing across each component. These functions are related through equations derived from the laws of electrical circuits. For instance, i_r and u_r are related by Ohm's law:

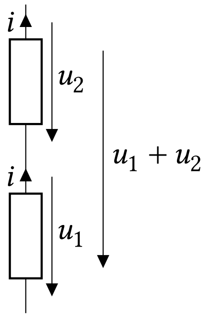
$$u_r = ri_r$$

and i_c and u_c are related by:

$$cu_c' = i_c$$

These relations are *invariant*: they are characteristic of the individual component and hold regardless of the context in which the component is used and at any point in time.

Figure 2.2: Two components connected in series



Kirchhoff's law gives additional equations for the circuit, that the voltage across the source u is equal to the sum of the voltage across the capacitor and resistor; and that currents through all components are equal. Assuming the voltage source produces a sine-wave yields the following system of 6 equations:

$$u = u_0 \sin(2\pi f t + \varphi) \tag{2.1}$$
$$cu_c' = i_c \tag{2.2}$$
$$u_r = ri_r \tag{2.3}$$

$$i = i_r \tag{2.4}$$

$$i_r = i_c \tag{2.5}$$

$$u = u_r + u_c \tag{2.6}$$

From this system of equations, finding an explicit formula for all 6 time-varying functions it relates is difficult. While it is possible in this case, this is very much an exception. The main difficulty naturally comes from solving for u_c which appears differentiated. By symbolic manipulation, it is possible to extract a relation purely between u_c and u_c'. Indeed, in equation 2.2, one can replace i_c by i_r (by equation 2.5), then i_r by $\frac{1}{r}u_r$ (by equation 2.3) and finally u_r by $u - u_c$ (by equation 2.6). This yields the following Ordinary Differential Equation (ODE):

$$u_c' = \frac{1}{rc}(u - u_c) \tag{2.7}$$

In general, an ODE is a differential equation of the form:

$$x' = f(x, t)$$

where $x : \mathbb{R} \to \mathbb{R}^n$ and $f : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$, for some $n \in \mathbb{N}$. The function f is the *residual function* of the ODE. An ODE together with a known-value of x as a given point in time (usually 0) forms an Initial Value Problem (IVP). Finding an explicit solution to an ODE is, in most cases, impossible. As an alternative, numerical integration [37] allows to approximate a solution to an IVP. The next section goes over some common numerical integration techniques, from the most basic ones to some more advanced strategies able to produce very accurate results, and integrated in industrial-strength ODE solvers.

37. Cellier et al., *Continuous System Simulation*. 2006

2.2 — Numerical integration

When an analytic solution to a differential equation is intractable, a numerical integration technique can be used. The general idea is to produce successive approximations of the solution at discrete points in time, starting from a known initial value for the solution. To do so requires estimating the rate of change of the solution between the previous time instant and the next one. The rate of change of a function x between two instants t and $t + \delta t$, with $\delta t > 0$ is given by $\frac{x(t+\delta t) - x(t)}{\delta t}$. Knowing its value and the value of $x(t)$, it is obvious how $x(t+\delta t)$ can be computed. To estimate the rate of change, numerical integration techniques exploit the fact that the derivative of a function corresponds to the instantaneous rate of change of that function, therefore computing it at different points over the interval $[t, t + \delta t]$ can give an approximation for the overall rate of change over that interval. The derivative itself can be computed by exploiting its relation with the solution given by the differential equation.

59. Euler, *Institutionum Calculi Integralis*. 1768

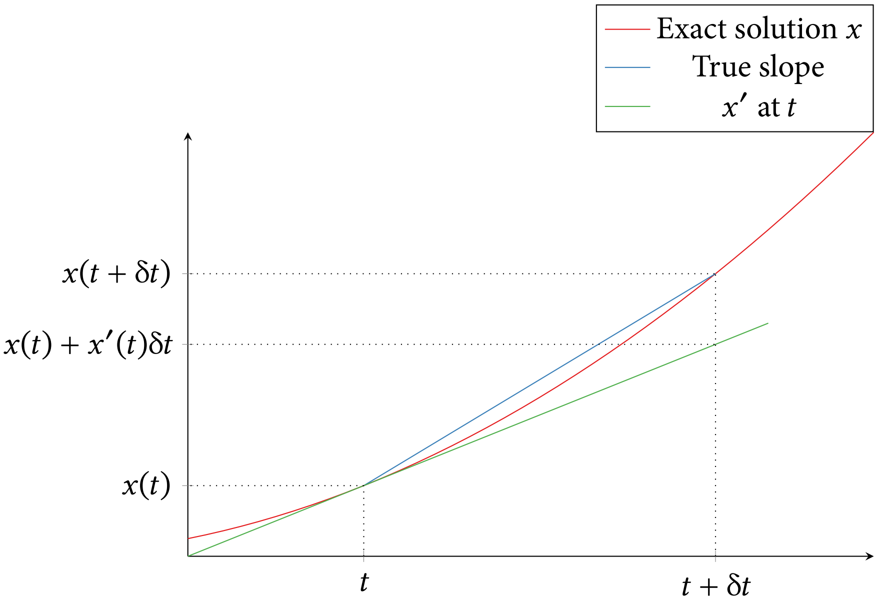


Figure 2.3: Illustration of the forward Euler's method

2.2.1 — *Simple methods*

The forward Euler's method [59] is arguably the simplest integration method. It consists of locally approximating the rate of change of the solution between t and $t+\delta t$ by $x'(t)$. Given an approximation for x at time t, the value of x at time $t+\delta t$ can be approximated like so:

$$x(t+\delta t) = x(t) + x'(t)\delta t$$

Since x is the solution to an ODE, $x'(t)$ can be replaced by $f(x(t),t)$:

$$x(t+\delta t) = x(t) + f(x(t),t)\delta t$$

The method is illustrated in figure 2.3. Euler's method is fairly imprecise and crude. It only gives an exact approximation when the solution is linear (i.e., the function f is constant). On the example figure, where the function is convex (its second-order derivative is positive), integrating with Euler's method systematically underestimates the solution (conversely, it would overestimate it if it had been concave). The Runge-Kutta methods [88, 132] form a family of methods (of which the forward Euler's method is a special case, the RK1 method) that attempt to provide a better estimate of the rate of change by computing several estimates for the derivative x', at different points on the interval $[t, t+\delta t]$. The classic Runge-Kutta method is RK4, which splits the interval $[t, t+\delta t]$ at the midpoint $t+\frac{\delta t}{2}$. The scheme computes 4 intermediate values, which correspond to different approximations of the derivative of x:

$$k_1 = f(x(t), t)$$
$$k_2 = f\left(x(t) + \delta t\frac{k_1}{2}, t + \frac{\delta t}{2}\right)$$

[88] Kutta, 'Beitrag Zur Naherungsweisen Integration Totaler Differentialgleichungen'. 1901 — [132] Runge, 'Über Die Numerische Auflösung von Differentialgleichungen'. 1895

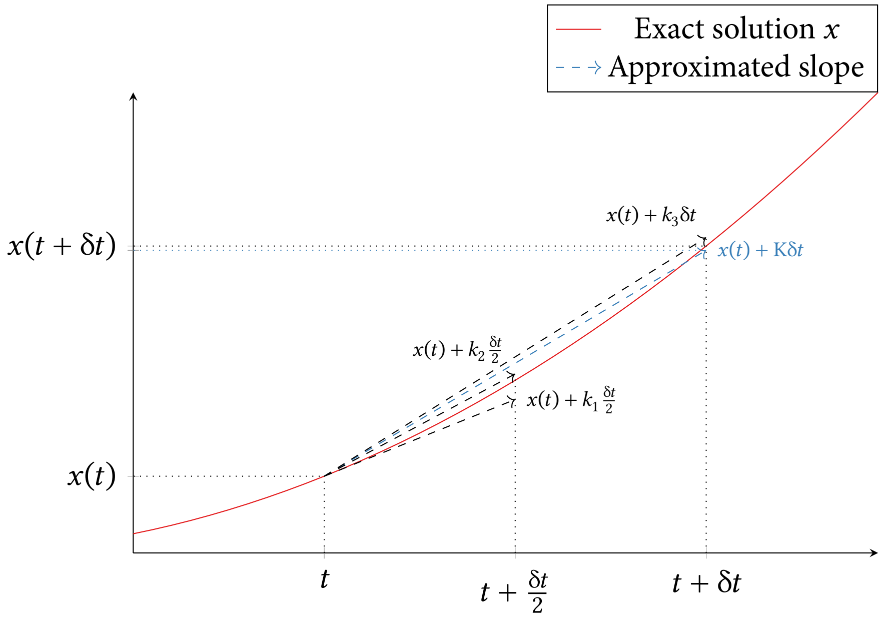


Figure 2.4: Illustration of the RK4 method

On the graph, K designates the averaged out rate of change:

$$K = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}$$

The function being integrated is the same as in figure 2.3. The guess for $x(t + \delta t)$ is much better with the RK4 method.

$$k_3 = f\left(x(t) + \delta t\frac{k_2}{2}, t + \frac{\delta t}{2}\right)$$
$$k_4 = f(x(t) + \delta t k_3, t + \delta t)$$
$$x(t+\delta t) = x(t) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}\delta t$$

k_1 corresponds to $x'(t)$. It is the rate of change that would have been computed by the forward Euler method. Instead of using k_1 to produce a solution however, it is used to estimate the solution at $t + \frac{\delta t}{2}$. Using this approximated solution, a value for the derivative at that point is computed k_2 along with a new estimate for $x\left(t + \frac{\delta t}{2}\right)$, which is used to compute a new approximation of the derivative k_3. Finally, k_4 is an estimation of the slope at $t + \delta t$, computed by approximating $x(t + \delta t)$ using k_3. RK4 then produces a final estimate for the rate of change between t and $t + \delta t$ by averaging out k_1, k_2, k_3 and k_4, giving more weight to the approximations corresponding to the rate of change at the midpoint. A graphical illustration for the method is presented in figure 2.4.

2.2.2 — *Implicit methods*

The Runge-Kutta methods and the forward Euler method are *explicit methods*, in the sense that the estimate for $x(t + \delta t)$ is produced through an explicit formula in terms of $x(t)$:

$$x(t + \delta t) = F(x(t), t)$$

Implicit methods on the other hand, produce an estimate of $x(t + \delta t)$ as an implicit solution of an equation of the form:

$$G(x(t + \delta t), x(t)) = 0$$

For instance, the backward Euler method produces $x(t + \delta t)$ as the solution of:

$$x(t + \delta t) = x(t) + f(x(t + \delta t), t + \delta t)\delta t$$

The presentation of algorithms for finding solutions to such equations is left for part II. Despite being harder to implement, implicit methods can turn out to be more efficient for a large category of problems where explicit methods require very small step sizes to stay numerically stable. Numerical stability here refers to the property of the integration algorithm to not wildly diverge from the solution as integration progresses.

A widely used family of implicit methods are Backward-Difference Formula (BDF) methods (of which the backward Euler method is a special case) [22, 26, 74]. Unlike the methods presented so far, the BDF methods are multistep. While singlestep methods use only $x(t)$ to compute $x(t + \delta t)$, multistep methods use $x(t)$ along with other previous approximations $x(t - \delta t)$, $x(t - 2\delta t)$, etc. The BDF-n method produces an approximation of $x(t + n\delta t)$ as the solution to:

$$\sum_{k=0}^{n} a_k x(t + k\delta t) = \delta t \beta f(t + n\delta t, x(t + n\delta t))$$

The values for the coefficients a_k and β can be found in [37, §4.7].

22. Brenan et al., 'Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations'. 1995 — 26. Brown et al., 'DASKR Package: DAE Solver with Krylov Methods and Rootfinding'. 2011 — 74. Hindmarsh et al., 'SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers'. 2005

37. Cellier et al., *Continuous System Simulation*. 2006

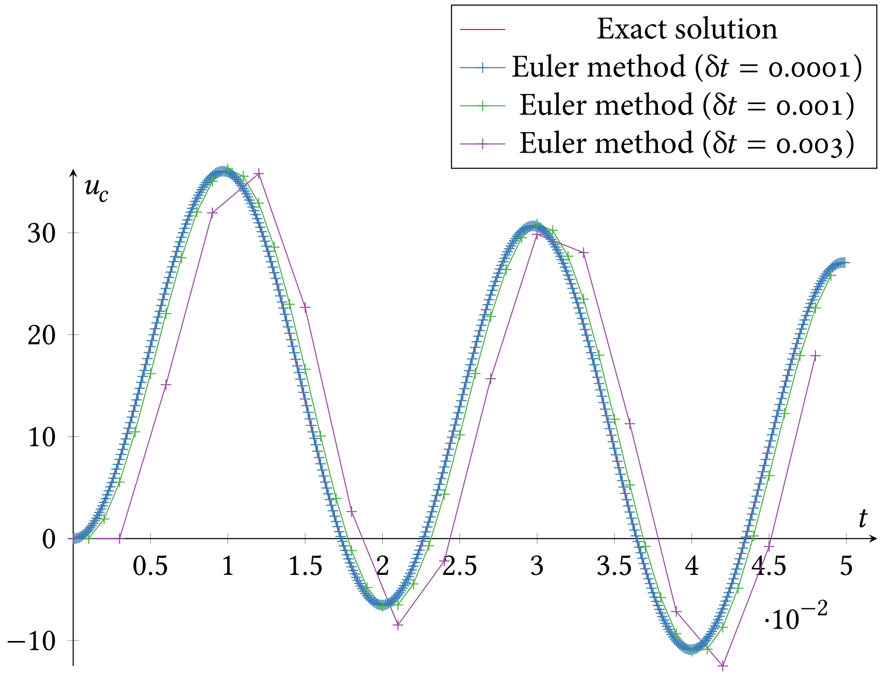


Figure 2.5: Simulation results for equation 2.7 using Euler's scheme

The simulations used the following numerical values: $u_0 = \sqrt{2} \cdot 220\,\mathrm{V}$, $f = 50\,\mathrm{Hz}$, $\varphi = 0$, $r = 50\,\Omega$, $c = 1\,\mathrm{mF}$. The exact solution is given by:

$$u_c = \kappa\left(\sin(\omega t) + \tau\omega\left(e^{-\frac{t}{\tau}} - \cos(\omega t)\right)\right)$$

given $\tau = rc$, $\omega = 2\pi f$ and $\kappa = \frac{u_0}{(\tau\omega)^2+1}$.

2.2.3 — *Variable-step methods*

So far, all the methods considered now used a fixed step size δt. This step size needs to be decided by the user of the method. This is potentially problematic. If the step size is too large for the problem at hand, the scheme will lose in precision or become unstable, the approximation widely diverging from the true solution. Conversely, if the step size is very low, while this should give a very good approximation, it will cause suboptimal performance. This is illustrated in figure 2.5, which shows simulation results for equation 2.7 using Euler's method with different step sizes. When the step size is small, the simulation is indistinguishable (on the graph at least) from the exact solution but a difference is clearly noticeable with larger step sizes. Unfortunately, determining a good step size from a system of equations may not be obvious. In large systems, different constituents may require different step sizes and the optimal step size may change during integration. Variable-step methods [7, 47] aim to address this issue by dynamically varying the step size based on the function being integrated. When the function is very stiff, the step size is reduced to gain precision; when it is not, the step size is increased to improve the performance.

[7] Ascher et al., 'Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations'. 1998 — [47] Curtiss et al., 'Integration of Stiff Equations'. 1952

A consequence of the ability to vary the step-size is that solvers may decide to evaluate the function being integrated out-of-order. Indeed, if the solver notices that it evaluated the function too late it may decide to reevaluate it at an earlier point. When implementing the residual function as a computer program, it is therefore highly preferable that the function does not depend on any state and certainly does not depend on the function always being evaluated in time-ascending order. The use of a varying time-step method is illustrated in figure 2.6. Equation 2.7 is simulated using the SUNDIALS CVODE solver [74], which uses a variable step BDF method to perform integration. The plot shows both the approximated solution and all the points at which it has evaluated the residual function. The solver was asked for a solution every 3 ms, which corresponds to step size used for the least precise approximation in figure 2.5. It clearly shows how the step size varies, getting smaller when the derivative of the solution changes the most, for instance around local extrema. By nature of the ODE, this is also when the derivative of the function being integrated is largest (in absolute value). The resulting approximation is very good, but requires considerably fewer evaluations than the most precise approximation in figure 2.5.

[74] Hindmarsh et al., 'SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers'. 2005

2.2.4 — *Integrating across discontinuities*

All numerical integration methods assume some regularity properties over the function being integrated, for instance, that it is continuous. Integrating across a discontinuity is generally not well-defined and can cause the solver to fail, or to take a very large number of steps to compute a solution. This is demonstrated on figure 2.7, where the

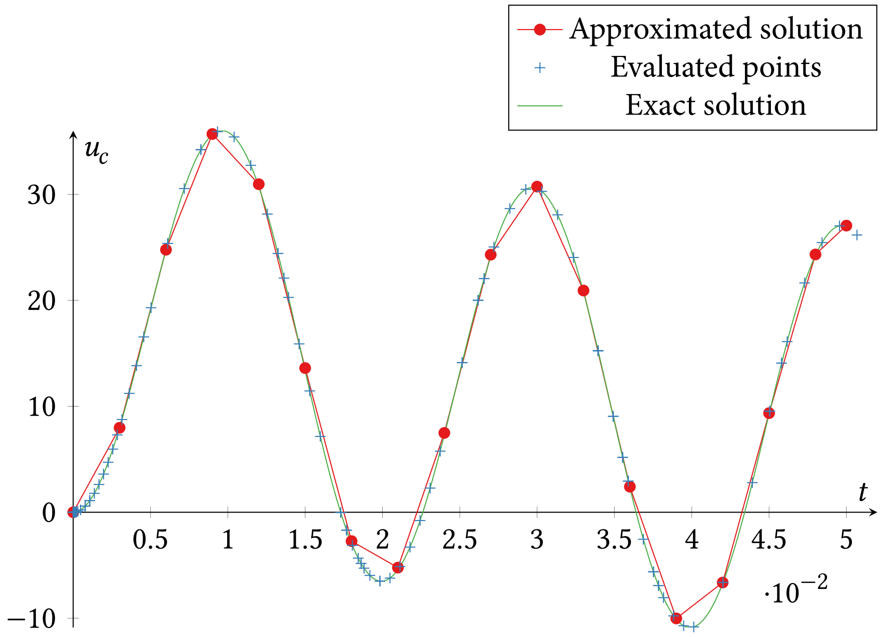


Figure 2.6: Simulation results using Sundials cvode solver

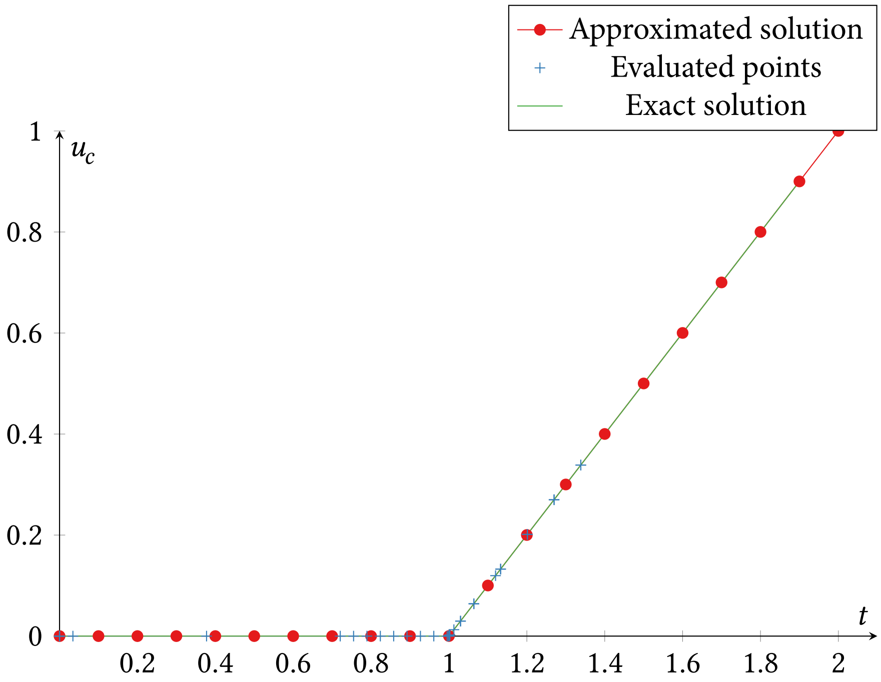


Figure 2.7: Integration of a discontinuous function using Sundials cvode

following IVP is passed to SUNDIALSCVODE:

$$\begin{aligned} x' &= \begin{cases} 0 & \text{if } t < 1 \\ 1 & \text{otherwise} \end{cases} \\ x(0) &= 0 \end{aligned}$$

To avoid the problem, it is preferable to split the problem in two. First integrate the IVP:

$$\begin{aligned} x' &= 0 \\ x(0) &= 0 \end{aligned}$$

until t passes 1 and then *reset* the solver as $t = 1$. To continue the integration, the solver can now be passed the following IVP:

$$\begin{aligned} x' &= 1 \\ x(1) &= x(1^{-}) \end{aligned}$$

where $x(1^{-})$ designates the value computed from x at $t = 1$ just before the solver reset. This can be achieved by making use of the root finding capabality of solvers. Indeed, most solvers allow the user to provide functions that compute a signal that should stop the solver when it crosses 0. These events are usually referred to as *zero-crossings* and allow to handle complex discontinuities or state changes, even when they depend on the computed value for the solution. The root-finding capabilities of solvers allow to 'pinpoint' the point in time at which the zero-crossing occurred.

Integrating the discontinuous problem in this way yields the graph in figure 2.8. It shows that the function being integrated was evaluated at far fewer points this way than in the case where the solver had to integrate across the discontinuity.

2.2.5 — *Summary*

This section presented an overview of some common numerical integration techniques and some of their limits. In particular, what will be important for what follows are the two following points:

— integrating across discontinuities requires special care, both for performance and accuracy reasons. In general, it is much preferable to handle discontinuities *outside* of the numerical solver. This can be done by presenting the solver with a problem without discontinuity, which is stopped at the point the discontinuity occurs. The solver is then reset and tasked with continuing integration after the discontinuity. Discontinuities can be precisely located by using a root-finding technique, as provided by many numerical solvers.

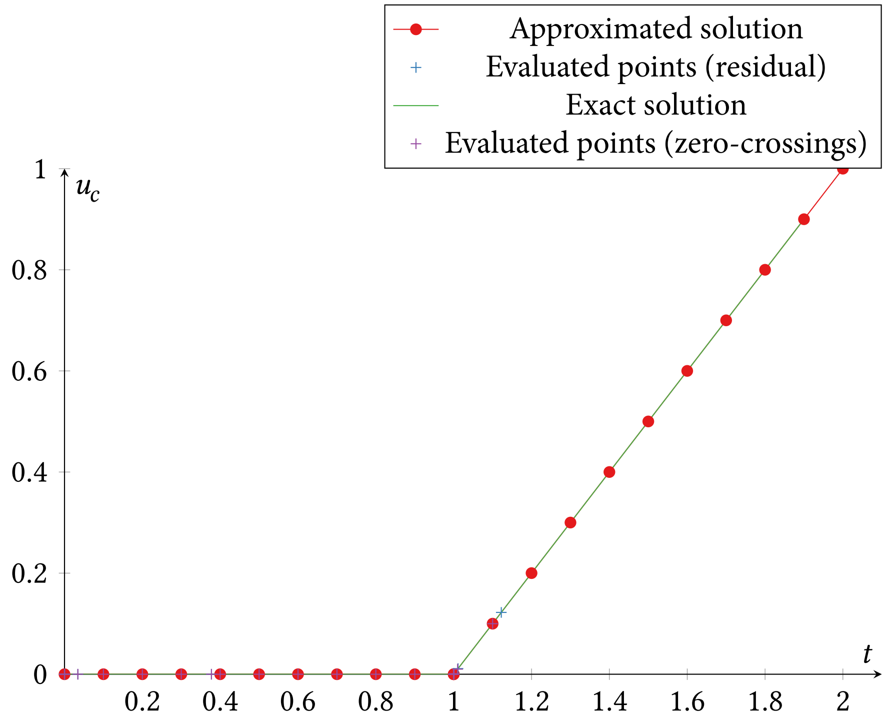


Figure 2.8: Integration of a discontinuous function using Sundials CVODE, using zero-crossings and solver reset at the point of discontinuity.

The graph depicts both the points at which the residual function was evaluated and where the root finding function was evaluated.

— integrating a computer program that depends on an internal state also requires special care. This is both due to the fact that solvers uses variable-state methods with backtracking, which can cause the state to be invalidated. Discrete state jumps can also cause discontinuities, which leads back to the problems described above. A good rule of thumb is that any state change should therefore be treated like a discontinuity.

Note that these considerations are not new [10].

The next section describes Functional Reactive Programming (FRP). FRP is an approach aimed at programming causal reactive applications. However, it has also successfully been applied for describing simulations. It will be used as an example for a causal approach to modelling. FRP on its own is a fairly naïve modelling language. Its shortcomings in that regard will be made explicit in §2.4.5 and relate, in part, to the points made above.

2.3 — Functional Reactive Programming

The idea behind causal modelling languages is to avoid the need to resort directly to a numerical solver, by providing language constructs to define variables as integrals of others. In other words, to define these variables as solutions of an ODE. The implementation of the language is then in charge of integration.

In this section, Functional Reactive Programming (FRP) [54, 151] is presented. FRP is a principled, declarative approach to programming reactive systems, focusing on describing interactions between time-

[10] Benveniste et al., 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. 2017

[54] Elliott et al., 'Functional Reactive Animation'. 1997 — [151] Wan et al., 'Functional Reactive Programming from First Principles'. 2000

varying values rather than reacting to individual events, and commonly supporting both continuous and discrete notions of time. As such, FRP addresses some of the inherent difficulties in programming reactive systems, and, in various concrete incarnations, it has had a considerable uptake [4, 28, 140] as well as inspired related approaches such as ReactiveX [128] and (the original) Elm [48]. While catering primarily for reactive applications, FRP is in fact a useful way to structure time-aware programs more generally, including simulations [79, 143].

4. Apfelmus, *Reactive-Banana*. 2011 — 28. Bünzli, *React, Functional Reactive Programming for OCaml*. 2010 — 140. Söylemez, *Netwire*. 2011

128. Microsoft, *ReactiveX*. 2011

48. Czaplicki, 'Elm: Concurrent FRP for Functional GUIs'. 2012

79. Hudak et al., 'Arrows, Robots, and Functional Reactive Programming'. 2003 — 143. Thaler et al., 'Pure Functional Epidemics: An Agent-Based Approach'. 2018

FRP is often realized as an EDSL in a functional host language like Haskell. Yampa [45] is a prominent representative of this approach. This section gives an introduction to FRP by means of Yampa and shows how it can be used as a simple causal modelling language.

45. Courtney et al., 'The Yampa Arcade'. 2003

2.3.1 — *Signals and signal functions*

FRP is centred around two abstractions: *signals* and *signal functions*. A signal represents a time-varying value, conceptually a function from time to values:

$$\textbf{Signal}\ \alpha \approx \mathit{Time} \rightarrow \alpha$$

In Yampa, signals are not first-class objects. Indeed, it is generally preferable to restrict what can be done with a signal. For instance, being able to sample a signal arbitrarily far back in time requires retaining in memory all the information about that signal, leading to memory leaks [92]. Conversely, signals are expected to be temporally causal: they cannot depend on a value computed in the future. The name classic FRP refers to those libraries that treat signals as first-class entities [54]. By contrast, Yampa and a wider family of libraries [8, 90, 135, 140, 141] grouped under the name arrowized FRP have first-class *signal functions*. Conceptually, these are functions from signals to signals:

92. Liu et al., 'Plugging a Space Leak with an Arrow'. 2007

54. Elliott et al., 'Functional Reactive Animation'. 1997

8. Bärenz, *Rhine*. 2017 — 90. Le, *Auto*. 2015 — 135. Scivally, *Varying*. 2015 — 140. Söylemez, *Netwire*. 2011 — 141. Söylemez, *Wires*. 2017

$$\textbf{SF}\ \alpha\ \beta \approx \textbf{Signal}\ \alpha \rightarrow \textbf{Signal}\ \beta$$

This enables libraries to define only a few signal functions that precisely define what can be done to a signal in such a way that an efficient implementation can be provided. For instance, in Yampa, referring to the previous value of a signal is done through the following `delay` signal function:

```
delay :: Time → a → SF a a
```

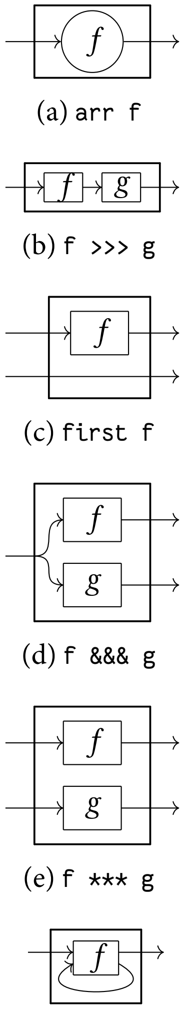
`delay t i` is a signal function that delays its input by `t` seconds, setting the output signal to `i` in the meantime. Using `delay` bounds the amount of time the previous values of a signal need to be remembered from at the time `delay` is applied, which lends itself to a reasonable implementation, by contrast to allowing the amount of delay to change with time without a bound.

2.3.2 — *Arrows*

Signal functions are an example of *arrows* [80], an abstract interface unifying ‘function-like’ types. Programming in Yampa is done by composing primitive signal functions using arrow combinators. These combinators are regular Haskell functions whose signatures are given below:

```
arr    :: (a → b) → SF a b
(>>>)  :: SF a b → SF b c → SF a c
first  :: SF a b → SF (a,c) (b,c)
(&&&)  :: SF a b → SF a c → SF a (b,c)
(***)  :: SF a b → SF c d → SF (a,c) (b,d)
loop   :: SF (a,c) (b,c) → SF a b
```

Figure 2.9: Main arrow combinators



(a) arr f

(b) f >>> g

(c) first f

(d) f &&& g

(e) f *** g

(f) loop f

arr lifts an ordinary function to a signal function, which applies the function to the input signal to produce the output signal. Composing signal functions is provided through the >>> serial composition operator and the two parallel composition operators, *** and &&&, the latter often called ‘fan-out’. Finally the feedback combinator loop, that instantly feedbacks one output of a signal function to itself. The meaning of these combinators are represented graphically in figure 2.9 using ‘boxes and arrow’ diagrams, similar to the block diagrams of Simulink for instance. frp libraries structured around arrows in this way are usually referred to by the name *arrowized frp*, in contrast with classic frp [54] which allows for the manipulation of signals directly.

To illustrate how to use them, let us define exponential, a signal function whose output is the solution to the ode defining the exponential function. Indeed, the exponential can be defined as the solution to the following ode:

$$\begin{aligned} y' &= y \\ y(0) &= 1 \end{aligned} \tag{2.8}$$

Integrating this equation on both sides yields:

$$\underbrace{\int_0^t y'(u)\mathrm{d}u}_{y(t)-y(0)} = \int_0^t y(u)\mathrm{d}u$$

Simplifying transforms the equation to:

$$y(t) = 1 + \int_0^t y(u)\mathrm{d}u$$

Yampa exposes the integral signal function which computes the integral of a signal between the first instant and the current time. Using the loop combinator, it is possible to feedback the computed value of the integral back to itself, much in the way the ode is defined. This leads to the following implementation:

[80] Hughes, ‘Generalising Monads to Arrows’. 2000

```
exponential :: SF a Double
exponential =
  loop (arr snd >>> integral >>> arr (+ 1) >>> arr dup)
```

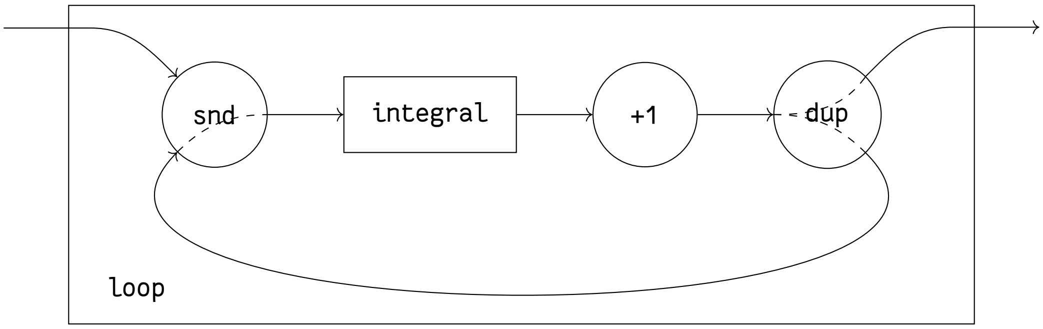
The `snd` function is the function that given a pair returns its second-element, `dup` on the other hand takes the a value and produces a pair containing the value in both fields:

```
snd :: (a, b) -> b
snd (_, y) = y

dup :: a -> (a, a)
dup x = (x, x)
```

Let us analyse the definition of `exponential`. `exponential` is defined in terms of `loop`. In this instance, the argument signal function that is passed to `loop` has type `SF (a, Double)   (Double, Double)`. Its input signal is a pair made of the input of `exponential` type `a` as the left argument, which is unused; and the current value of the integral. The inner signal function is then the serial composition of 4 signal functions: the first one discards the input of type `a` using `snd`. This leaves only the current value of the integral to pass to the integral. The integral's initial value is always 0, but problem 2.8 requires that the value at the first-instant of `exponential` is 1, hence why the value produced by `integral` is added 1. The final signal function duplicates the result so that the `loop` produces the value of the integral and feedbacks it to be integrated. Figure 2.11 shows a diagrammatic view of the definition of `integral`.

Figure 2.11: Block-based illustration of the definition of `exponential`



`loop` feedbacks its second output immediately as an input to the inner signal function. Although it may seem that this would cause an infinite loop or require solving a fix-point equation, in this case, it does not. Indeed, while `integral` depends on its input, it doesn't depend *instantly* on it: it can produce a value at time *t* without knowing the value of its input at time *t*. This makes `integral` a loop-breaker, like it is in other causal modelling languages. Note that Yampa does not check if the `loop` actually contains a loop breaker and it will accept nonsensical programs such as:

```
absurd :: SF a Int
absurd = loop (arr snd >>> arr (+1) >>> arr dup)
```

which defines a signal following the equation $y = y+1$, which of course has no solution.

2.3.3 — proc-*notation*

It quickly becomes tedious to program solely using combinators. Fortunately, arrows, such as Yampa signal functions, can be constructed using Paterson's arrow notation [120], also called *proc*-notation. It allows intermediate signals to be named and explicitly give them as arguments to signal functions. It is treated specially by the compiler and desugared using the arrow combinators introduced in §2.3.2. Consider the problem of calculating the trajectory of an object in free fall, with vertical position y and velocity v. Such a system is described by the following pair of equations:

$$
\begin{aligned}
v(t) &= v_0 + \int_0^t -g \,\mathrm{d}u \\
y(t) &= y_0 + \int_0^t v(u) \,\mathrm{d}u
\end{aligned}
$$

The equations are translated to *proc*-notation like so, where `y0` and `v0` are respectively the initial position and velocity of the object:

```
freeFall :: (Double, Double) → SF a (Double, Double)
freeFall (y0, v0) = proc _ → do
  let g = 9.81
  v ← arr (v0 +) <<< integral ⤙ -g
  y ← arr (y0 +) <<< integral ⤙ v
  returnA ⤙ (y,v)
```

The `proc` keyword introduces names for the input signal of the signal function, like λ in lambda-abstractions. Then a list of statements instantiate subordinate signal functions, with inputs to the signal functions to the right of ⤙, and their outputs to the left of ←. A **`let`** statement can be used to define additional signals as pure expressions of existing signals. The last statement's output is the output of the defined signal function. Here we use `returnA`, which is a synonym for the identity arrow.

The *proc*-notation also supports defining signals with feedback with the `rec` keyword. Doing so, the `exponential` signal function from §2.3.2 can be redefined as:

[120] Paterson, 'A New Notation for Arrows'. 2001

```
exponential :: SF a Double
exponential = proc _ -> do
  rec y <- arr (+1) <<< integral -< y
  returnA -< y
```

`rec` defines a sub-block which can span across multiple statements. This way, it's possible to describe a larger model, for instance the one for the circuit from §2.1.

```
simpleCircuit :: SF a Double
simpleCircuit = proc _ -> do
  t <- time -< ()
  u <- arr sourceWave -< t
  rec ur <- arr (\(u, uc) -> u - uc) -< (u, uc)
      ir <- arr (/ r) -< ur
      let ic = ir
      uc <- arr (/ c) <<< integral -< ic
  returnA -< uc
  where sourceWave t =
          u0 * sin(2 * pi * f * t + phi)
```

Above, `time` is a signal function that produces a signal with the local time. It is then used in `sourceWave` to produce the source voltage `u`. The next statements are mutually recursive and define the other signals `ur`, `ir`, `ic` and `uc`.

2.3.4 — *Discrete behaviour*

Dynamic changes in the structure of the network are possible through a *switching* combinator:

```
switch :: SF a (b, Event c) -> (c -> SF a b) -> SF a b
```

The **`Event`** type is isomorphic to an option type and is used to model discrete-time signals:

```
data Event c = NoEvent | Event c
```

The output of a switch is the `b` output from the first signal function, until an event is produced, after which a new signal function is computed from the second argument, using the value `c` carried by the **`Event`**. Then the output of the whole switch is the output from that new signal function.

Using `switch`, one can use the `freeFall` signal function to model a bouncing object. A bouncing object acts like a free-falling one until it reaches the ground (at coordinate $y = 0$). When it does, its position remains the same, but its velocity is inverted (assuming bouncing is lossless):

```
bouncing :: (Double, Double) → SF a (Double, Double)
bouncing (y0, v0) = switch bounceAux bouncing
  where bounceAux = proc _ → do
          (y, v)  ← freeFall (y0, v0)  ⤙ ()
          bounce  ← edge ⤙ y ⩽ 0
          returnA ⤙ ((y, v), bounce `tag` (y, - v))
```

The code above defines an auxiliary signal function bounceAux which pairs the signal function freeFall with an event. The event is generated by mean of the edge signal function of type **SF Bool** (**Edge** ()), which produces an event that carries the () value whenever the input signal goes from **False** to **True**. Note that edge never reports an event at the first instant. It would be problematic if it did since after the first switch occurs, bouncing is always called with a negative initial position. If edge detected that, it would immediately trigger the next switch, which would trigger the next switch, etc. without ever producing any result since bounceAux would never run. bounce is only an **Event** (), but switch needs bounceAux to produce an **Event** (**Double**, **Double**) in order to construct a new instance of bouncing. As the name indicates, tag tags an event with a new value, overwriting the one it is carrying. Here it is used with Haskell's infix notation, bounce `tag` (y, -v) is the same as tag bounce (y, -v).

There is, strictly speaking, no structural change happening in the definition of bouncing: the switch here is simply used for introducing a discontinuity in the vertical velocity of the falling object. But it can also be used to radically change the dynamic behaviour of the system. For instance, one could model a defective capacitor, which breaks at some point in time and behaves like an open switch afterward:

```
breakingCapacitor :: Double
                  → Double
                  → SF (Double, Event ()) Double
breakingCapacitor c uc0 =
  switch capacitor openSwitch
  where
    capacitor = proc (ic, break) →
      uc ← arr (uc0 +) <<< arr (1 / c) <<< integral ⤙ ic
      returnA ⤙ (ic, break)
    openSwitch = proc _ →
      returnA ⤙ 0
```

2.4 — Yampa's implementation

This section describes the implementation of a typical Yampa-like frp library as a continuation-based embedding. Yampa's implementation itself is slightly more complicated [107]. These details are left-out until §2.4.4 to help simplify the discussion.

[107] Nilsson, 'Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types'. 2005

Although Yampa is meant for programming hybrid continuous-time and discrete-time systems, its implementation is discrete. It emulates hybrid features, such as integration, by very simple approximations. The definition of the type **SF** is:

```
data SF a b = SF (a -> (b, DTime -> SF a b))
```

where **DTime** represent a time difference and is defined as a synonym for **Double**. A signal function is a wrapper around a *transition function* that, given an input a produces an output b along with a continuation. The continuation is a function that when applied to some amount of time that has passed produces a new signal function to execute at that instant. The following sections show how to define the combinators and primitives of Yampa that have been presented so far.

2.4.1 — *Implementation of the arrow combinators*

The simplest arrow combinator is `arr`, which lifts a pure function to a signal function by applying the function to the input signals to produce an output. This leads naturally to this implementation:

```
arr :: (a -> b) -> SF a b
arr f = SF sf
  where sf _ a = (f a, const (arr f))
```

Note how the continuation of `arr f` is always itself, regardless of the time delta since its behaviour does not change with time.

Composition combinators have similarly straight-forward implementations. The fan-out operator for instance:

```
(&&&) :: SF a b -> SF a c -> SF a (b, c)
SF tf1 &&& SF tf2 = SF tf3
  where tf3 dt a = ((b, c), tf3')
          where (b, tf1') = tf1 dt a
                (c, tf2') = tf2 dt a
                tf3' dt = tf1' dt &&& tf2' dt
```

`&&&` simply applies the transition functions of both subordinate signal functions to the same input. The continuation function is computed from the continuation of the subordinates by applying them to the same time difference, thus maintaining synchrony, and recursively composing them with `&&&`.

Along the same principles, here are the implementations of serial composition:

```
(>>>) :: SF a b -> SF b c -> SF a c
SF tf1 >>> SF tf2 = SF tf3
  where tf3 a = (c, tf3')
          where (b, tf1') = tf1 a
```

```
                (c, tf2') = tf2 b
                tf3' dt = tf1' dt >>> tf2' dt
```

Finally, `loop` is implemented by making use of Haskell's laziness:

```
loop :: SF (a,c) (b,c) → SF a b
loop (SF tf) = SF ltf
  where ltf a = (b, ltf')
          where ((b, c), tf') = tf (a, c)
                ltf' dt = loop (tf' dt)
```

Notice how the output `c` of the transition function is also passed as its input. This technique is sometimes called 'tying-the-knot' and relies on the fact that function calls in Haskell do not evaluate their arguments and that the evaluation of a particular value only causes the evaluation of exactly those values that are needed to compute the result. Let's look at an example on a simplistic example (modified from [125]):

```
g :: (Int, Int) → (Int, Int)
g (x, y) = (x + 1, y + 1)

f :: Int → Int
f x = z
  where (y, z) = g (x, y)
```

When evaluating `f 0`, the result forces the second element of the pair returned by `g (x, y)`, which is `y + 1`. To evaluate `y + 1`, the first element of the pair returned by `g (x, y)` must be computed, which is `0 + 1`. `y` therefore evaluates to `1` and `z` evaluates to `2`. What laziness allows for in this case is a form of dynamic scheduling of operations. Instead of requiring the programmer to perform this scheduling manually, lazy evaluation allows for it to be discovered at runtime. Note that knot-tying has other useful applications not relevant to the work presented here, for instance for defining cyclic data structures. The small example above can be lifted as is into a signal function using `loop` and `arr` like so:

```
sf :: SF Int Int
sf = loop (arr g)
```

2.4.2 — *Implementation of primitives*

So far, the continuation returned by the transition function hasn't really showed its usefulness. In Yampa's implementation, it has essentially two use-cases. The first is to allow local state to be maintained across invocations. This is illustrated by the implementation of `integral` that uses Euler's method:

[125] Pouzet et al., 'Modular Static Scheduling of Synchronous Data-Flow Networks'. 2010

```
integral :: SF Double Double
integral = SF tf
  where tf v = (0, cont 0 v)
        cont i pv dt = SF tf'
          where tf' v = (ni, cont ni v)
                  where ni = i + dt * pv
```

integral is defined using two transition functions. The first one is only applied at the first time-step, after which the second one is used. Let us look at the latter first. In addition to the time passed since the previous invocation dt it has two additional arguments: the value of the integral at the previous time step and the value of the input signal at the previous time step. Using these two extra-arguments, it computes the new value of the integral at the current time with a forward Euler method (see §2.2) and produces the continuation by passing the new value of the integral and the current value of the input. The initial transition function tf only exists to bootstrap the process, since the value of the input value before the first instant is undefined. The value produced by integral at the current time step is indeed independent from its input at the current time, like §2.3.2 relied on.

The second use for the continuation is to handle dynamic structural changes, as the implementation for switch shows:

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
switch (SF tf) next = SF stf
  where stf a =
          case evt of
            NoEvent  → (b, stf')
            Event c  → nextSF a
              where SF nextSF = next c
          where ((b, evt), sf') = tf a
                stf' dt = switch (sf' dt) next
```

The transition function for switch uses the transition function of the first signal function until it produces an event, after which a new signal function (and transition function) is computed and used as a replacement.

2.4.3 — *Interfacing with Yampa*

Running a signal function is done with the reactimate function, whose (simplified) type signature is:

```
reactimate :: IO a
           → IO (DTime, a)
           → (b → IO Bool)
           → SF a b
           → IO ()
```

reactimate init sense react sf starts by running the init action to produce the initial input to the signal function. Then, it runs the sense action that produces the time difference since the previous instant and a new input. Finally react is used to handle the result produced by the signal function at this instant and returns a boolean indicating whether the signal function should keep running or stop. The fact that both the sense and react are **IO** actions allow the signal function to interface with the outside world. This can be used to communicate key presses events in a game for instance, or read the value of a sensor in an embedded application.

As an example, let's write a program that simulates simpleCircuit (which models the circuit in figure 2.1). It samples the result every 1×10^{-4} s until 5×10^{-2} s.

```
simpleCircuit :: SF a Double

main :: IO ()
main = do
  timeRef ← newIORef 0
  reactimate init
            (sense timeRef)
            (react timeRef)
            simpleCircuit
  where init = pure ()
        sense timeRef = do
          t ← readIORef timeRef
          let dt = 1e-4
          writeIORef timeRef (t + dt)
          pure (dt, ())
        react timeRef uc = do
          print uc
          t ← readIORef timeRef
          pure (t ⩾ 5e-2)
```

To keep track of the current time, the sense and react actions make use of a mutable reference timeRef to share the current time. A mutable reference pointing to a value of type a is represented with the type **IORef** a. It can be read and written from using the functions:

```
readIORef  :: IORef a → IO a
writeIORef :: IORef a → a → IO ()
```

sense is in charge of updating the time reference when it runs, while react checks whether the current time is more than the time limit, stopping the simulation when it is.

2.4.4 — *Self-simplification & dynamic optimizations*

There are a number of advantages to the current Yampa representation. Mainly its simplicity, flexibility and overall reasonable performance. It also has the property that FRP networks are self-simplifying. Consider the implementation of switch: once the switch has occurred, the transition function of the switch vanishes and is completely replaced by the transition function the switch switched into. This property can be further exploited by having a special representation for some signal functions that are amenable to simplifications, thus allowing dynamic optimizations over the FRP network [107].

For instance, one can introduce an additional constructor to the type **SF** to distinguish signal functions that are applications of arr, like so:

```
data SF a = SFArr (a → b)
          | SF (a → (b, DTime → SF a b))

arr :: (a → b) → SF a b
arr f = SFArr f
```

The definition of combinators and primitive can now be made aware that they are being applied on applications of arr and provide optimized implementations. The serial composition operator can then dynamically simplify the FRP network by transforming compositions of two signal functions constructed using **SFArr** into a single signal function. It can also provide more efficient implementations in the case where only one of the subordinate signal function is constructed using **SFArr**:

```
(>>>) :: SF a b → SF b c → SF a c
SFArr f >>> SFArr g = SFArr (g . f)
SFArr f >>> SF tf = SF tfc
  where tfc a = (c, tfc')
          where (c, tf') = tf (f a)
                tfc' dt = SFArr f >>> tf' dt
SF tf >>> SFArr g = SF tfc
  where tfc a = (g b, tfc')
          where (b, tf') = tf a
                tfc' dt = tf' dt >>> SFArr g
SF tf1 >>> SF tf2 = ...
```

Furthermore, because the definition of >>> applies itself recursively to the continuations of its subordinate signal functions, it can dynamically apply the optimization if the subordinates get simplified enough. For instance, in the following:

```
sg :: SF Int Int
sg = switch aux next
```

[107] Nilsson, 'Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types'. 2005

```
  where aux = proc n -> do
          evt <- edge -< n > 0
          returnA -< (n, evt)
        next () = returnA

sf :: SF Int Int
sf = arr (+1) >>> sg
```

`sf` cannot be statically optimized since `sg` is defined in terms of `switch`. However, once `sg` has switched, its representation reduces to the representation of `next ()`, which is `SFArr id`. At this point, the representation of `sf` can be optimized to a single `SFArr` constructor by `>>>`.

Representing the identity arrow with `SFArr id` still incurs the overhead of a function composition when it is used, which could be simplified if it was represented explicitly, like `SFArr`. Unfortunately, this is not possible with the conventional Algebraic data type (ADT) machinery of Haskell, since defining:

```
data SF a b = SFId
            | SFArr (a -> b)
            | SF (a -> (b, DTime -> SF a b))
```

defines `SFId` as having type `SF a b`, but the identity arrow should have type `SF a a`. At first glance, it would seem that it is possible to use that constructor by simply exposing a definition with a restricted type:

```
returnA :: SF a a
returnA = SFId
```

However, the problem now becomes the implementation of `>>>`. Indeed, we would wish for the ability to write the following definition:

```
(>>>) :: SF a b -> SF b c -> SF a c
SFId >>> sf = sf
```

Unfortunately, this is not well-typed, since `sf` in general has type `SF b c` and the return type should be `SF a c`. It is then necessary, to perform this optimization, to force the type of `SFId` to always be `SF a a`. This is possible using Generalized Algebraic Data Types (GADT)[122]. GADT are a form of dependent types which allow for a lot more flexibility when defining the constructors of a type. Defining `SF` in terms of a GADT is done like so:

```
data SF a b where
  SFId  :: SF a a
  SFArr :: (a -> b) -> SF a b
  SF    :: (a -> (b, DTime -> SF a b)) -> SF a b
```

[122] Peyton Jones et al., 'Simple Unification-Based Type Inference for GADTs'. 2006

In a GADT definition, the return type of each constructor application can be given its own type. This allows for defining **SFId** as having type **SF** a a at all time. This then enables the compiler to perform additional type unifications during pattern-matching. Thus, in the definition of >>>:

```
(>>>) :: SF a b → SF b c → SF a c
SFId >>> sf = sf
```

the compiler unifies a and b, making the clause well-typed, and allowing for an even more efficient implementation.

2.4.5 — *Shortcomings*

2.4.5.1 — *Performance shortcomings* A natural assumption when writing arrow code is that the 'wiring' is essentially free. In this context, wiring (or routing) designates the process of guiding a signal from the output of the signal function producing it to the inputs of the consuming signal functions. After all, the way signals flow between signal functions is mostly static, with switch being the prominent exception. But, as can be seen from the implementation &&& for instance, routing is not quite free: &&& has to perform a tuple allocation at each step to pack its result, which other combinators then have to unpack or rearrange later.

This looks like a small price to pay, especially when allocations are cheap like in most functional language implementations. However it turns out that this wiring code is often a large part of an FRP network. The definition of exponential (at the start of §2.3.2) for instance makes use of two combinators that only exist for the purpose of routing signals correctly: arr snd and arr dup. Code that uses the *proc*-notation is also particularly susceptible to this due to the way it is desugared. Consider this block:

```
proc x → do
  y ← sf ⤙ x
  z ← sg ⤙ y
  returnA ⤙ (x,z)
```

It is desugared by GHC into a slightly more complicated version of the following:

```
arr (\x → (x,x)) >>> first sf >>>
arr (\x → (x,x)) >>> first (arr (\(x,y) → y) >>> sg) >>>
arr (\(z, (x,y)) → (x,z)) >>> returnA
```

Most of the combinators in the above code handle routing, and all of them allocate and deallocate tuples. In fact, §3.5 will show that the cost associated with routing grows quadratically with the number of lines in a *proc*-notation block.

Surely, it is possible to do better. Indeed, the intuition that the wiring is, for the most part, static, is correct. For static routing, there are well-known techniques used in e.g. the implementation of synchronous dataflow languages [17] and continuous system simulation [37] that essentially eliminate the routing overhead by representing signals with shared imperative variables. All that is required is that the writing and reading of each variable, for each time step, is carefully coordinated, which in the setting of (first-order) synchronous dataflow languages and simulation of structurally static continuous systems can be achieved through static scheduling of the computations; i.e., generation of a sequence of imperative assignments in a suitable order.

To adopt this method for an arrowized FRP library requires two changes. First, it becomes important to precisely identify what routing primitives are necessary in order to be able to replace the use of `arr` by explicit routing operations that the implementation can optimise away. This can be done in the same way as in §2.4.4, by introducing additional constructors to the **`SF`** type that represent signal functions that only perform routing. Then, it requires a more precise representation for signals. In particular, it becomes important to distinguish between pairs of signals and signals of pairs, as the former can be represented using a pair of references while the first can be represented as a single reference on a pair.

2.4.5.2 — *Conceptual reasons* There are further advantages, both practical and conceptual, of making a more profound distinction between different kinds of signals. Not simply between pairs, but also between individual kinds of signals themselves. So far, only (notionally) continuously varying signals have been considered. Even events are represented as continuous signals of an option type. However, a discrete-time signal can be expected to change relatively infrequently compared to continuous-time signals. Exploiting this through a different implementation strategy for such signals can lead to efficiency gains.

Further, there are advantages to distinguishing between truly continuous signals and signals that exist all the time but change only at certain points in time. For instance, while it is clear what a continuous signal of real numbers means, it is less clear what a continuously varying signal of booleans means and whether sampling from such a signal is ever meaningful. On the other hand, the meaning of a constant by part signal of booleans is clear.

As another example of the usefulness of this separation, consider Yampa's timed delay combinator `delay :: Time → a → SF a a`. This combinator allows any signal to be delayed a specific amount of time. In terms of implementation, it is essentially a buffer for signal samples. However, since Yampa makes no assumptions as to the regularity of sampling, it may happen that individual samples either need to be discarded or duplicated in order to make up for variations in the sampling

[17] Biernacki et al., 'Clock-Directed Modular Code Generation for Synchronous Data-flow Languages'. 2008

[37] Cellier et al., *Continuous System Simulation*. 2006

frequency. This is fine for signals that conceptually are continuous. However, it is a disaster for signals carrying events, as this can mean that events get lost or are duplicated. Therefore Yampa provides a separate delay combinator for events that works by scheduling a single output event a fixed time into the future. However, because Yampa does not fundamentally differentiate between continuous-time signals and events, there is nothing that stops a user from using the continuous-time delay also for events.

Finally, §2.2.5 summed up the capabilities of numerical integration techniques. While a better approach to numerical integration would be desirable for Yampa, it is important that the program suitably limits what can be done with models to make sure that integration produces sensible results in a timely manner.

2.5 — Summary

This chapter provided an introduction to causal modelling, numerical integration and Functional Reactive Programming. It presented the notion of ODE and techniques to numerically approximate their solutions. It then showed how FRP, and specifically arrowized FRP as implemented in Yampa, could be used as a simple modelling language to express models in terms of ODE; and the principles behind its implementations.

The next chapter is dedicated to addressing the shortcomings described in the previous section. It relies on a more precise description of the nature of the signals a signal function operates on. Doing so allows for the implementation of an FRP network as an efficient imperative program, providing great performance benefits. While this work was originally carried-out with a focus on efficiency, the resulting library allows to place additional limitations on FRP that are desirable in the context of a modelling language. While this has been left as future work, this paves the way to the implementation of an efficient causal modelling EDSL integrated with precise ODE solvers.

Scalable Functional Reactive Programming

3

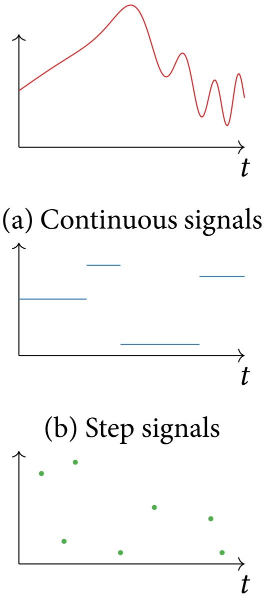
This chapter presents an implementation of the ideas sketched in §2.4.5. The main goal of this work is to improve the performance of FRP networks by compiling away all wiring. To do so requires a strict separation between different kinds of signals, in particular pairs of signals and signals of pairs. Further minor improvements can be provided by distinguishing between different kinds of individual signals. For instance, between continuous signals, expected to change very frequently, and event signals. Doing so allows for providing additional restrictions on FRP which can be used to make writing FRP networks safer, and also paves the way to the integration of more advanced numerical integration techniques, such as the ones presented in §2.2; although this has been left as future work.

The resulting FRP library is called Scalable FRP (SFRP), since it aims at supporting very large FRP networks. It supports most of the features of YAMPA with two notable exceptions: it only supports a limited form of feedback (implemented in YAMPA with the arrow `loop` combinator), and it does not support collection based switching as presented in [110].

This chapter is organized as follows. §3.1 discusses how different kinds of signals can be distinguished, using *signal descriptors*. §3.2 then describes how signal functions are described in SFRP. In YAMPA, signal functions were implemented by shallow-embedding. For the purpose of this work, SFRP will be implemented as an Abstract syntax tree (AST); i.e., as a deep-embedding. §3.3 presents work that implements a custom *proc*-notation desugarer using GHC meta-programming capabilities. Indeed, SFRP is not able to use the standard *proc*-notation from GHC. However, like was demonstrated with YAMPA, special syntax is essential for the usability of an arrowized FRP library. The compilation of the SFRP AST to an efficient imperative representation is the subject of §3.4. Finally, performance evaluation of SFRP against YAMPA is carried out in §3.5.

[110] Nilsson et al., 'Functional Reactive Programming, Continued'. 2002

Figure 3.1: Signal kinds



(a) Continuous signals

(b) Step signals

(c) Event signals

3.1 — Signal representation

Signals are distinguished in two categories: individual signals and pairs of signals. Following the distinction introduced in [136], individual signals themselves are separated in three groups, illustrated in figure 3.1:

[136] Sculthorpe et al., 'Keeping Calm in the Face of Change: Towards Optimisation of FRP by Reasoning about Change'. 2011

- Continuous signals (C): (notionally) continuously varying time signals (but may exhibit discontinuities). Typically representing a physical quantity such as time, position, etc.
- Step signals (S): piecewise constant continuous-time signals. For instance input from a discrete controller.
- Event signals (E): signals that are present only at discrete points in time.

This distinction is captured in a type of *signal descriptors* **SD**:

```
data SD a where
  C  :: a -> SD a
  S  :: a -> SD a
  E  :: a -> SD a
  P  :: SD a -> SD b -> SD (a, b)
  N  :: SD Void
```

SD is not for use at the value level, but at the type level to describe the kind of signals a signal function operates on. GHC allows to lift a data constructor to the type level [153]. A lifted data constructor is prefixed with a quote, thus **'C Double** is the descriptor of continuous signals carrying doubles. Much as data constructors are lifted to the type level, type constructors are lifted to the kind (type-of-type) level. The kind of **'C Double** is **SD Type**, where **Type** is the kind of Haskell types [a]. Pairs of signals have a distinct descriptor. For instance **'P ('C Double) ('E Bool)** is a pair of a continuous signal of **Double** and of an event signal carrying a **Bool**. There is also a descriptor for signals that carry no value, **N**. **N** is of type **SD Void**, where **Void** is the empty type of Haskell.

These descriptors can now be used in GADT declarations as constraints on the type of data-constructors. For instance, the type of values associated to a signal descriptor can be defined as:

```
data SDVal a where
  CVal  :: a                       -> SDVal ('C a)
  SVal  :: (Bool, a)               -> SDVal ('S a)
  EVal  :: Maybe a                 -> SDVal ('E a)
  PVal  :: SDVal a -> SDVal b      -> SDVal ('P a b)
  NVal  ::                            SDVal 'N
```

As a minor optimization, **SVal** is represented as a tuple of a Boolean and a value, the Boolean indicating whether the signal has changed since the previous step. This allows to avoid some computation when a signal changes rarely, like a step signal is expected to.

In the following, for readability, {.,.} is used to denote a pair of signal descriptors, {} denotes a signal carrying no value, and the quote of lifted constructors are omitted. Thus **'P ('C Double) ('E Bool)** will be written {**C Double, E Bool**}.

[153] Yorgey et al., 'Giving Haskell a Promotion'. 2012

[a] Historically, **Type** has also been denoted by ⋆ and GHC supports both syntax. This work only uses the notation **Type**.

3.2 — Signal function representation

The definition of signal functions can be refined using descriptors:

```
data SF (i :: SD p) (o :: SD q) where
```

In this definition, `i` and `o` are given an explicit kind, which must be signal descriptors. Since the goal is to compile signal functions to an imperative representation, they need to be represented in a way that is easy to inspect; i.e., as a deep embedding or abstract syntax tree, except that parts that are not expected to be compiled can be represented shallowly. The constructors of the type **SF** are introduced over the next sections. In §3.2.1, combinators mimicking arrow combinators are introduced, in §3.2.2 constructors for additional primitives and for mediating between signals of different kinds are introduced and finally in §3.2.3, combinators for precise routing are defined.

3.2.1 — *Arrow-like combinators*

3.2.1.1 — *Composition operators* Let's begin with the composition operators, which are defined like so:

```
(:>>>:)  :: SF a b → SF b c → SF a c
(:&&&:)  :: SF a b → SF a c → SF a      {b, c}
(:***:)  :: SF a c → SF b d → SF {a, b} {c, d}
```

As intended, the parallel composition operators `:***:` and `:&&&:` truly operate on pairs of signals.

3.2.1.2 — *arr* `arr` is a little different. Unlike the composition operators, it directly operates on the signal passing through it. Clearly, applying a pure function to a signal cannot change its kind. The type chosen for `arr` is therefore the following:

```
SFArr :: (a → b) → SF (k a) (k b)
```

Because `a` and `b` are types, `k` must have kind `Type → SD Type`, thus one of **C**, **S** or **E**, but not a partially applied **P**. Note however that the function passed to `arr` is still between `a` and `b`, not between `SDVal a` and `SDVal b` for instance. This is intentional. It prevents the creation events out of thin air when applying `arr` over event signals. As for **S**-kinded signals, an optimisation becomes possible. **S** signals, being expected to change rarely, carry a Boolean flag indicating if they have changed. Hence when `arr` is used on such a signal, work can be saved by not recomputing the output if the input didn't change. This restriction on `arr` is very useful, but requires a new primitive, `arr2` that combines two signals of the same kind together, represented by the following constructor of **SF**:

```
SFArr2 :: (a → b → c) → SF {k a, k b} (k c)
```

While `arr` could be expressed in terms of `arr2`, it is preferable not to as this would lose some optimisation opportunities.

3.2.1.3 — *Loop* The final arrow combinator is `loop`. A general `loop` has proven difficult to implement and in fact, several FRP approaches do not support it[a]. The reason why SFRP doesn't support it will become clear in the next chapter when the implementation is discussed. However, some form of feedback is very important, especially in modelling applications where it is used to define ODE.

An alternative is a delayed loop combinator: instead of instantaneously sending the output of the inner function to the input, the current output is fed to the signal function at the next time step (and the input at the current time step is therefore the value of the output at the previous time step). Disallowing well-founded instantaneous loops without any delays, like the example at the end of §2.4.1, does not terribly limit the expressivity. In practice, it seems that the only time where such loops are used is to avoid having to schedule statements manually when using the *proc*-notation, like in the following example:

```
proc x → do
  rec y ← f ⤙ z
      z ← g ⤙ x
  returnA ⤙ (y, z)
```

While this surely is convenient, it also makes it quite easy to make a mistake. Since YAMPA has no mechanism to detect these errors[a], they can be quite frustrating to debug[b].

In a hybrid context, the meaning of delaying however requires some thought. §2.3.1 presented the `delay` signal function, which can delay a signal by a given length of time. However YAMPA also provides a `pre` signal function. Inherited from LUSTRE [36], it delays the input signal by one step. The size of that step depends on the way the signal function is being run with `reactimate`. Because of YAMPA's implementation, this is not too much of a problem. At worse, this makes the `pre` signal function a bit of a leaky abstraction, since it exposes the fact that the implementation is discrete.

However, in a truly hybrid setting, the ability to refer to the previous value of a continuous signal is problematic. Recall the properties of numerical solvers that were discussed in §2.2. The use of variable-step methods means that, unlike with YAMPA, the user is not necessarily in charge of the step size. SIMULINK for instance allows for the equivalent of the `pre` signal function, while still using variable-step solvers. It has been shown that this can make the simulation of a model drastically depend on the way it is used: for instance, a model produces a different results when it is simulated alongside another one, even though there are no interactions between the two, but simply by virtue of the fact that the existence of the presence of the other model forced the solver at to use a different step size [10].

[a] See §4.1

[a] Currently, attempting to execute a non-well founded loop in YAMPA produces either an infinite loop or, if GHC's runtime is able to detect that it's in a loop, the cryptic message «`loop`» followed by program termination.

[b] Speaking from personnal experience.

[36] Caspi et al., 'LUSTRE: A Declarative Language for Programming Synchronous Systems'. 1987

[10] Benveniste et al., 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. 2017

While the plan is not to use an ODE solver for SFRP, it would be desirable if its design didn't rule it out immediately. Therefore, the `Loop` construct is limited in two ways. For step signals, a regular delayed loop construct is provided. Defined like so:

```
DLoop :: StepSignal c
      ⇒ SDVal c
      → SF {a, c} {b, c}
      → SF a b
```

Notice the **StepSignal** constrain on the feedback type parameter `c`. **StepSignal** is a type-class defined on signal descriptors like so:

```
class StepSignal c
```

Is is meant to be enforced when `c` is either an **S**-kinded signal or a collection of step signals. This leads to the following two instances:

```
instance StepSignal (S a)
instance (StepSignal a, StepSignal b) ⇒ StepSignal {a, b}
```

Note the extra argument of type **SDVal** `c` to **DLoop**, which contains the value produced initially by the signal function. This function can be used to implement `pre`, but exclusively on step signals:

```
pre :: StepSignal c ⇒ SDVal c → SF c c
pre init = DLoop init (SFRouter Swap)
```

For continuous signals, the choice is to not provide any way to delay a signal, to avoid the problems described earlier. Instead, SFRP exposes an integral-loop combinator:

```
IntegralLoop :: Continuous c
             ⇒ SDVal c
             → SF {a, c} {b, c}
             → SF a b
```

Instead of sending the previous value of the output of the inner signal function, **IntegralLoop** feedbacks the integral of the output (which can be initialised with the extra **SDVal** `c` passed to **IntegralLoop**, like with **DLoop**). **Continuous** serves the same role as **StepSignal**, in that it can be used to designate both a signal continuous signal or a collection of continuous signal. Unlike **SignalStep** however, **Continuous** defines a method:

```
class Continuous c where
  integrate :: DTime → SDVal c → SDVal c → SDVal c
```

integrate implements an integration scheme for signal descriptors of kind c. The first argument is the step size, the second argument is the current value of the integral and the third argument is the computed value of the derivative of the signal. For single continuous signals, this enables implementing the forward Euler method like in Yampa:

```
instance Continuous (C Double) where
  integrate dt (CVal x) (CVal x') = CVal (x + dt * x')
```

for pairs of continuous signals, the output is simply the pair of the results of calling integrate on both members of the pair:

```
instance (Continuous a, Continuous b) ⇒ Continuous {a, b} where
  integrate dt (x `PVal` y) (x' `PVal` y') =
    integrate dt x x' `PVal` integrate dt y y'
```

Note that the interface of **Continuous** could be generalised to support more advanced integration methods. For instance, integrate could be given the ability to poll the result of executing the inner signal function directly. This could then be used to implement more advanced integration technique or communicate with an ODE solver. Like for pre, integral can be implemented in terms of **IntegralLoop**:

```
integral :: SF (C Double) (C Double)
integral = IntegralLoop (CVal 0) (SFRouter Swap)
```

Note that an alternative design is possible which provides a single loop combinator, capable of handling both step signals and continuous signal. The behaviour of the combinator is that of **DLoop** when the signal is an **S**-kinded signal, that of **IntegralLoop** when it is **C**-kinded and a mix of both when it is **P**-kinded. It is unclear whether providing such combinator would be a good idea. It has, for now, not been needed and this strange mixed semantics might not do it justice.

One could summarise the approach of SFRP as 'loopifying' Yampa's loop breaker. Instead of having both loop and its loop-breaker integral, there is only one combinator with the two behaviours mixed together. While this avoids having to deal with unbounded loops, and, as §3.4 will show, helps with implementation, it has one drawback: there are many loop breakers in Yampa for example delay, which cannot be implemented in terms of pre or integral. In order for them to be used with feedback, one must then implement a 'looping' version of these loop-breakers, instead of just the original loop-breakers (which can then be implemented in terms of the new looping combinator).

3.2.2 — *Primitive signal functions*

This new library needs all the primitives from Yampa, e.g. switch:

```
Switch :: SF a {E c, b} → (c → SF a b) → SF a b
```

However it also need additional primitive in order to mediate between signals of different kinds. In Yampa, this is done by using `arr`. Since `arr` and `arr2` are limited to operating on signals of the same kind, this is not possible in sfrp. Here are some examples of primitives that are defined with the goal of mediating between signals:

— Tagging the event with the value of the signal at the time it occurs:

```
Tag :: (a -> b -> c)
    -> SF {C a, E b} (E c)
```

— Edge detection for continuous signals:

```
Edge :: s
     -> (a -> s -> (s, Maybe b))
     -> SF (C a) (E b)
```

— Change detection for step-signals:

```
Jump :: s
     -> (a -> s -> (s, Maybe b))
     -> SF (S a) (E b)
```

— Accumulation [a]:

```
Accum :: s
      -> (a -> s -> (s, b))
      -> SF (E a) (S b)
```

— Removal of events from an event signal:

```
FilterE :: (a -> Maybe b)
        -> SF (E a) (E b)
```

— Event merging:

```
MergeE :: (a -> a -> a)
       -> SF {E a, E a} (E a)
```

Events act as mediators between step and continuous signals, akin to what is being done in hybrid simulation languages [10]. These combinators all have equivalents in Yampa.

3.2.3 — *Routers*

Suppose we wish to compose two signal functions of type:

```
sf  :: SF k {{E a, C b}, S c}
sg  :: SF   {C b, {E a, S c}} k'
```

Surely `sf` and `sg` should be composable, provided the output of `sf` can be rearranged to match the way the signals are paired in `sg`. In Yampa, this would be done by interleaving an `arr` rearranging the signals:

```
sf >>> arr (\((ea, cb), sc) -> (cb, (ea, sc))) >>> sg
```

[a] Note that this combinator can be used to implement a version of **Tag** for step-signals.

[10] Benveniste et al., 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. 2017

But using arr, thereby not making a distinction between scaffolding used for routing and the actual values carried by signals, is precisely what this approach aims at avoiding! The function inside arr is purely rearranging the signals, but not performing any operations on them. This fact cannot be known by an FRP library, since a function is a black-box that cannot be analysed at runtime. Therefore, regardless of the representation of signals or signal functions, an implementation cannot do better that execute any code written with arr (including the routing code above) at every iterations. For that reason, for an implementation to efficiently implement wiring, a way to represent transformations between signals explicitly is needed. §3.4 will show how this enables SFRP to handle routing with little runtime overhead compared to current implementations.

Signal descriptors have the shape of binary trees: **P** being the nodes and **N**, **C**, **S** and **E** being the leaves. The set of transformations needed is therefore the set of transformations needed to match the shape of any binary tree into any other, provided all the non-**N** leaves of the destination tree are present in the original tree. §3.3 shows that the sufficient set of transformations to do so are left- and right-rotations, to modify the shape of the tree without changing the order of the leaves; duplication and deletion of a tree, to remove unused leaves, or add extra ones; and swapping two children of a node, to change the order of the leaves in the tree. In addition, combinators to compose routers together and to apply them to a specific subtree on a node are needed. This points to a type defined in a way similar to **SF**:

```
data Router i o where
  IdRout    :: Router a a
  -- Tree rotations
  LeftRot   :: Router {a, {b, c}} {{a, b}, c}
  RightRot  :: Router {{a, b}, c} {a, {b, c}}
  -- Subtree deletions
  DelRout   :: Router i N
  FstRout   :: Router {a,b} a
  SndRout   :: Router {a,b} b
  -- Leaves duplications
  DupRout   :: Router a {a, a}
  -- Swapping leaves
  SwapRout  :: Router {a,b} {b,a}
  -- Combining routers
  AppLeft   :: Router a b → Router {a,c} {b,c}
  AppRight  :: Router c d → Router {a,c} {a,d}
  CompRout  :: Router a b → Router b c → Router a c
```

Like **SF**, the type constructor **Router** operates on signal descriptors and not on types directly. A router can then be lifted into a signal function with a dedicated constructor:

```
SFRouter :: Router i o → SF i o
```

To give an idea for what a router can be used for, let's define a function routeVar which rearranges the shape of a value of type **SDVal** accordingly:

```
routeVal :: Router i o → SDVal i → SDVal o
routeVal IdRout a = a
routeVal LeftRot (a `PVal` (b `PVal` c)) = (a `PVal` b) `PVal` c
routeVal RightRot ((a `PVal` b) `PVal` c) = a `PVal` (b `PVal` c)
routeVal DelRout _ = NVal
routeVal FstRout (a `PVal` _) = a
routeVal SndRout (_ `PVal` b) = b
routeVal DupRout iv = PVal iv iv
routeVal (AppLeft r) (a `PVal` c) = routeVal r a `PVal` c
routeVal (AppRight r) (a `PVal` b) = a `PVal` routeVal r b
routeVal (CompRout p q) a = routeVal q (routeVal p a)
```

The types of **Router** and **SDVal** are constrained in such a way that there is only one possibility for the function in each cases.

The example between sf and sg can now be rewritten as:

```
sf >>> Router (CompRout (AppLeft SwapRout) RightRot) >>> sg
```

Users are not expected to use routers directly, except maybe for simple ones; much like users of Yampa don't write code using complex calls to arr. However having explicit routers is crucial for implementing an efficient desugaring for the *proc*-notation. This is the topic of the next section.

3.3 — Custom *proc*-notation implementation

The appeal of arrowized FRP is, at least partially, the possibility to use a convenient notation for expressing FRP networks, the *proc*-notation [120]. Unfortunately, the new **SF** type is not a standard arrow which means that it is not possible to make use of it. While GHC supports another syntax extension, the Rebindable Syntax, which allows to use GHC's desugarer with custom arrow combinators instead of the standard ones, it is not powerful enough to circumvent that problem. Indeed, the arr combinator exposed by **SF** is too restrictive to be compatible with the way GHC operates, since it is used for routing, while it would need to use the routing combinator introduced in §3.2.3.

Fortunately, GHC supports extensions to its syntax through quasi-quoting [94]. Thus it's possible to define a custom *proc*-notation that will be desugared into the constructors of **SF**. This custom notation is as close as possible to the one used for arrows, in an attempt to make porting existing Yampa programs (and users) easier. The falling object example from §2.3.2 can be rewritten like so:

[120] Paterson, 'A New Notation for Arrows'. 2001

[94] Mainland, 'Why It's Nice to Be Quoted'. 2007

```
freeFall :: (Double, Double)
        → SF a [sd| {C Double, C Double} |]
freeFall (y0, v0) = [sf| proc i → do
  v ← arr (v0 +) <<< integral ⤙ {| -9.81 |}
  y ← arr (y0 +) <<< integral ⤙ v
  returnA ⤙ {y,v} |]
```

The syntax `[sf| ... |]` introduces a quasi-quote, instructing GHC to translate the string between the brackets using a quasi-quoter named `sf`. Inside the quasi-quote, most things can be read as normal *proc*-notation code. Since we insist on distinguishing signal kinds, pairs of signals are denoted using `{,}`. When introducing a signal name, it is possible to also indicate its kind with the syntax `x :: C`, here indicating that `x` is of kind **C**. It can be used to improve clarity or to give information to GHC's typechecker. While annotations rarely if ever are needed for GHC to infer a signal's kind, they can at least help to generate better error messages. Using Haskell expressions as input to signal functions is supported, but they must be enclosed in `{|...|}`. They may refer to signals but all signals being referred to must be of the same signal kind as we do not wish to mix signals of different kinds implicitly. Pattern matching on Haskell constructors is not supported, however. There is also an `sd` quasi-quoter for typesetting signal descriptors in a more convenient way.

Let us go back to the example from §2.4.5:

```
proc x → do
  y ← sf ⤙ x
  z ← sg ⤙ y
  returnA ⤙ (x,z)
```

which is desugared by GHC into:

```
arr (\x → (x,x)) >>> first sf >>>
arr (\x → (x,x)) >>> first (arr (\(x,y) → y) >>> sg) >>>
arr (\(z, (x,y)) → (x,z)) >>> returnA
```

The general form for desugaring a statement of the form `y ← sf ⤙ x` is:

```
arr (\x → (x,x)) >>> first (<glue> >>> sf)
```

First, the input signal is duplicated, as one must conserve the current inputs so that later statements can use them. Note that the input signals here are not only the input signals of the whole signal function, but also the output signals of all the preceding signal functions. For instance, `x` is used by both `sf` and `returnA`. Hence, in the following block:

```
proc a → do
  b ← sf ⤙ {a,a}
  c ← sg ⤙ {b,a}
  d ← sh ⤙ c
  e ← si ⤙ {{a, b}, a}
```

the input signals when desugaring the statement for `si` will have the shape:

```
{d, {c, {b, a}}}
```

One of the duplicated inputs is then fed into the signal function making up the body of the statement. Some 'glue' must then filter and rearrange the inputs so that signal function can use them. This glue is routing code, which GHC implements using `arr`. To desugar the new custom *proc*-notation, the only thing needed is a way of computing that glue in terms of routers.

In §3.2.3, we mentioned that routers are really transformations on binary trees. Hence, the glue is simply the transformation between the tree representing all the inputs at this point in the network and the tree of inputs expected by the signal function. Below is an algorithm to compute this transformation.

The first step consists of computing the transformation that deletes unused leaves from the initial tree, and duplicates the leaves that appear more than once. In the example above with the input signal:

```
{d, {c, {b, a}}}
```

since the signal function expects:

```
{{b, a}, a}
```

this requires duplicating the `a` leaf and removing the `c` and `d` leaves. This is done by using the duplicating (**`RouterDup`**) and deleting routers (**`DelRout`**, **`FstRout`** and **`SndRout`**). Because a signal cannot be produced in multiple sources, it is guaranteed that every signal in the input signal appears uniquely. Performing this transformation is therefore only a matter of counting how many times each signal appears in the desired input, traversing the tree that represents the current input and duplicating or removing a leaf based on how many are needed. There is only one slight caveat: it is important to also make sure that there are enough **`N`**-kinded signals in the input, as a signal function may rely on this. For the example above, the router generated to perform this transformation is:

```
CompRout DelFst
        (CompRout DelFst
                  (AppLeft IdRout)
                  (AppRight DupRout))
```

which would yield the following signal, when applied to the input:

```
{b, {a, a}}
```

This now has the correct number of leaves, but these leaves are not in the right order and the signal is not in the right shape. The problem of matching the shape of two binary trees using only tree rotations is well-known (typically, in the context of binary search tree) [139]. Since tree rotations (on the whole tree or on subtrees) can be expressed with routers, the only thing needed is to compute a router that will sort the leaves in the correct order, and then, using rotations, compute a router that matches the shape the signal function expects.

Computing the sorting transformation is fairly complicated. Indeed, it isn't clear how a transformation that exchanges two leaves in a binary tree can be expressed using the operations at our disposal. There is one case where swapping is easy however: when the tree is list shaped, i.e. when for every node, the left branch points to a leaf, and when the leaves that need to be swapped are consecutive. In this case, the transformation is simple: a right-rotation, followed by swapping on the node created on the left, followed by a right transformation. This is illustrated on figure 3.2. This corresponds to the following router:

```
CompRout RightRot (CompRout Swap LeftRot)
```

To sort a whole list shaped tree, the desugarer can then simply used bubble-sort, since bubble-sort only requires the ability to swap two adjacent elements.

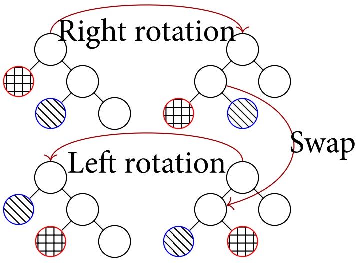


Figure 3.2: Swapping two leaves in a list-shaped tree

To sort any binary tree is then only a matter of transforming it to a list shaped tree using tree rotations, applying the sorting method devised above and then matching the shape of the tree to the desired shape using tree rotations. In the example above, the input signal was under the form `{b, {a, a}}`. This tree is already list shaped, so nothing needs to be done to turn it into a list shaped tree. In the desired input for `si`, the order of signals `a`, `b`, `a`. To have the same order with the current input requires applying a router to swap the first two leaves, yielding:

```
{a, {b, a}}
```

Then, matching the shape of to `{{a,b},a}` can simply be done with a single right-rotation.

The transformations obtained from each step are then composed together in one router that makes the glue. For the example, the final router is therefore given by:

```
CompRout r1 (CompRoute r2 r3)
  where r1 =
          CompRout DelFst (CompRout DelFst
```

[139] Sleator et al., 'Rotation Distance, Triangulations, and Hyperbolic Geometry'. 1986

```
                          (AppLeft IdRout)
                          (AppRight DupRout))
        r2 = IdRout
        r3 = RightRot
```

This algorithm is not expected to be optimal in any way. In practice, however, one should recall that this transformation runs at compile-time inside ghc, where one can expect programs to be fairly small. In the case of frp networks, this is less than 100 lines (more than what we expect to encounter). Even at around 100 lines, there is no significant difference in compilation times from equivalent Yampa programs. In terms of the optimality of the router being generated, although the transformation produces fairly large ones, simple simplifying pass performing obvious optimisations, such as deleting composition of identity routers, is already enough to obtain a much cleaner result. Regardless, §3.4 will show how routers incur small runtime overhead in sfrp.

3.3.1 — *Missing support for feedback at the syntax level*

The handling of feedback at the *proc*-notation syntax level is left as future work, purely for lack of time. Recall that there is no `loop` in sfrp, only delayed loops, so it is unclear how `rec` blocks would be desugared.

In ghc, `rec` blocks are desugared by identifying every signal that appears both as an input and an output, and defining these signals as the ones being fed back. For instance, recall the following example:

```
simpleCircuit :: SF a Double
simpleCircuit = proc _ → do
  t ← time ⤙ ()
  u ← arr sourceWave ⤙ t
  rec ur ← arr (\(u, uc) → u - uc) ⤙ (u, uc)
      ir ← arr (/ r) ⤙ ur
      let ic = ir
      uc ← arr (/ c) <<< integral ⤙ ic
  returnA ⤙ uc
  where sourceWave t =
          u0 * sin(2 * pi * f * t + phi)
```

In the `rec` block defining that signal function, `uc` and `ur` appear as both input and output. Therefore both of them are getting fed back using `loop`. When they are used as input, ghc makes use of the fed back value. In the example, this would lead to the following partial desugaring of the `rec`-block in the `aux` auxiliary signal function:

```
simpleCircuit :: SF a Double
simpleCircuit = proc _ → do
  t ← time ⤙ ()
```

```
u <- arr sourceWave -< t
uc <- loop aux -< (t, u)
returnA -< uc
where
  sourceWave t =
    u0 * sin(2 * pi * f * t + phi)

  aux = proc ((t, u), (fedback_ur, fedback_uc)) -> do
    ur <- arr (\(u, uc) -> u - uc) -< (u, fedback_uc)
    ir <- arr (/ r) -< fedback_ur
    let ic = ir
    uc <- arr (/ c) <<< integral -< ic
    returnA -< (uc, (ur, uc))
```

As mentioned in §3.2.1.3, rec blocks can be quite confusing and are error-prone. Without any mechanism to help debug instantaneous cycles, an argument can be made that the extra convenience that rec block offer is worth the price in debugging time. It is my opinion that, if such an extension to the *proc*-notation were to be implemented, some mechanism would need to exist to provide clear diagnostics to the user, in the style of what exists for synchronous dataflow languages for instance [46].

3.4 — Compiling Scalable FRP

Over the previous sections, a new description for signal functions was introduced, making use of signal descriptors to more precisely describe the kind of signals they operate on. The resulting library offers a similar level of expressivity as Yampa but with additional constraints on signals. Great care has been taken to explicitly represent routing and define what interactions between signals of different kinds are allowed. The custom *proc*-notation allows for clearly and conveniently expressing FRP networks with this library.

In this section, a translation for the precise description of FRP networks previously proposed is presented. Like the compilation methods of synchronous dataflow languages [17], it consists of representing each signal function using one (or several) input references, one (or several) output references and a stepping action that updates the output from the input.

The goal in doing so is to be able to compile away the wiring, now explicitly represented by **`Router`**, by only wiring references at the point the network is compiled. Thus avoiding the need to wire values while the networks run. In doing so however, we will wish to retain some of the pleasant properties of the current implementations, outlined in §2.4.4, most specifically self- and dynamic-simplification.

46. Cuoq et al., 'Modular Causality in a Synchronous Stream Language'. 2001

17. Biernacki et al., 'Clock-Directed Modular Code Generation for Synchronous Data-flow Languages'. 2008

```
data SF (i :: SD p) (o :: SD q) where
  Jump :: s -> (s -> a -> (Maybe b, s)) -> SF (S a) (E b)
  Edge :: s -> (s -> a -> (Maybe b, s)) -> SF (C a) (E b)
  AccumBy :: s -> (s -> a -> s) -> SF (E a) (S s)

  Arr :: (a -> b) -> SF (k a) (k b)
  Arr2 :: (a -> b -> c) -> SF (k a `P` k b) (k c)

  Tag ::(a -> b -> c) -> SF (E a `P` C b) (E c)

  Switch :: SF a (E c `P` b) -> (c -> SF a b) -> SF a b
  DLoop :: StepSignal c
        => SDVal c
        -> SF (a `P` c) (b `P` c)
        -> SF a c
  IntegralLoop :: Continuous c
               => SDVal c
               -> SF (a `P` c) (b `P` c)
               -> SF a c

  Router :: Router i o -> SF i o

  (:>>>:) :: SF a b -> SF b c -> SF a c
  (:***:) :: SF a b -> SF c d -> SF (a `P` c) (b `P` d)
  Const :: SDVal o -> SF i o
```

Figure 3.3: Complete definition of the **SF** type

This section is organized as to guide the reader from a very simple implementation, suitable for static FRP, to implementations of growing complexity as it progresses to support more complicated structures, such as dynamic switching.

Benchmarks will be presented in §3.5 that show a significant improvement in performance compared to Yampa.

3.4.1 — *Signal references*

In the same way that the type **SDVal** defined was defined, it is possible to define the type of *references* parametrized over a signal descriptor.

```
data SDRef a where
  CRef :: IORef a -> SDRef (C a)
  SRef :: IORef Bool -> IORef a -> SDRef (S a)
  ERef :: IORef (Event a) -> SDRef (E a)
  PRef :: SDRef a -> SDRef b -> SDRef (P a b)
```

This type is indeed very similar to **SDVal**. Note that, as was repeated many times, pairs of signals are represented as two distinct references.

Similar to how the function routeVal was used to rearrange a **SDVal** signal representation using a router, a function routeRef is defined to rearrange references, with the following type:

```
routeRef :: Router i o → SDRef i → SDRef o
```

In addition, a function newSDRef for constructing an **SDRef** and two functions readSDRef and writeSDRef for reading and writing to an **SDRef** are provided, with the following types:

```
newSDRef   :: SDVal i → IO (SDRef i)
readSDRef  :: SDRef i → IO (SDVal i)
writeSDRef :: SDRef i → SDVal i → IO ()
```

Their definition is given below:

```
readSDRef :: SDRef i → IO (SDVal i)
readSDRef (CRef r) = do
  v ← readIORef r
  pure (CVal v)
readSDRef (SRef r) = do
  v ← readIORef r
  pure (SVal v)
readSDRef (ERef r) = do
  v ← readIORef r
  pure (EVal v)
readSDRef (PRef p q) = do
  pv ← readSDRef p
  qv ← readSDRef q
  pure (PVal pv qv)
```

Using readSDRef and writeSDRef, a function syncSDRef that reads the values in the **SDRef** passed as the first argument and writes it into the reference in the second argument is also defined:

```
syncSDRef :: SDRef i → SDRef i → IO ()
syncSDRef ir or = do
  iv ← readSDRef ir
  writeSDRef or iv
```

3.4.2 — *Compiling static signal functions*

Let's start by considering a compilation function only for simple static signal function. There are two sensible signatures for this function:

— either the compilation function receives both an input and an output reference and produces the stepping action:

```
compile :: SF i o → SDRef i → SDRef o → IO (IO ())
```

— or the compilation function only receives the input reference and produces both the output reference and the stepping action:

```
compile :: SF i o → SDRef i → IO (SDRef o, IO ())
```

Here, the stepping action is simply of type `IO ()`. When executed, it advances the state of the network by one step and updates the output reference accordingly. Executing it many times advances the network by that many steps.

In this simple case, the second `compile` function is more interesting. Indeed, passing both input and output references forces the stepping action to update the content of the output reference at each time step. But precisely some signal functions, most notably routers, do not need to perform that action, if the input reference is routed from the output reference. Using the second formulation, it is possible to compile a **SFRouter** signal function like so:

```
compile (SFRouter router) ir = pure (routeRef rr ir, pure ())
```

At runtime, the stepping action for this signal function has nothing to do! By virtue of the output reference being derived from the input reference, and thus being automatically 'synchronized' with the input reference. This formulation also allows for the arrow combinators to be quite neatly expressed, like serial composition:

```
compile (sf :>>>: sg) ir = do
  (orf, stprSF) ← compile sf ir
  (or, stprSG) ← compile sg orf
  pure (or, stprSF >> stprSG)
```

>> is the sequencing operator for two **IO** actions. Notice how the output reference of `sf` can be reused as the input reference of `sg`, as one would expect. Note that some of the simplifications outlined in §2.4.4 can be implemented in this function, albeit not all. For instance, the composition of two `arr` defined functions can still be optimised to a single `arr`, like so:

```
compile (Arr f :>>>: Arr g) = compile (Arr (g . f))
```

However, when the handling of dynamism will be discussed in the next section, ways of simplifying dynamic networks will also be necessary. Similarly, the parallel composition also benefits from the more precise representation:

```
compile (sf :***: sg) (PRef ir1 ir2) = do
  (or1, stpr1) ← compile sf ir1
  (or2, stpr2) ← compile sg ir2
  pure (PRef or1 or2, stpr1 >> stpr2)
```

The dispatching of the input and output references to both subordinate signal functions can be done at the compilation step, and there is no need to repeat the process at each iteration.

Working in an imperative context allows for some things that the continuation-based interface allowed for, most notably maintaining state, simply by using references. For instance, compiling the stateful signal function **AccumBy** is done like so:

```
compile (AccumBy init acc) (ERef r) = do
  cr ← newIORef True
  sr ← newIORef init
  let step = do
    evt ← readIORef r
    case evt of
      Nothing →
        writeIORef cr False
      Just s → do
        a ← readIORef sr
        let a' = acc s a
        writeIORef cr True
        writeIORef sr a'
  pure (SRef cr sr, step)
```

3.4.3 — *Handling dynamic changes*

Dynamic change is the biggest difficulty when generalising the proposed scheme. While it is possible to implement switching using the simple `compile` function, it would lead to space- and time-leaks in lots of cases. To see this, consider the following implementation of `compile` for switch. Its principle is simple: since both the output reference and the stepping action will change while the switch runs, what the current output reference and what the current stepping action are can be maintained within references of their own. The stepping action for the whole switch then runs the stepping action for the currently active signal function by retrieving it from the reference and retrieves the output value by retrieving first what the current output reference is, and then getting the output value from it. This is illustrated by the following code:

```
compile (Switch sf next) ir = do
  (PRef firstOutRef (ERef evtRef), firstStepper) ←
    compile sf ir
  outRef ← newIORef or
  stepperRef ← newIORef firstStepper
  switchOr ← newSDRef =<< readSDRef or
  let switchStepper = do
        stepper ← readIORef stepperRef
```

```
9          stepper
10         evt <- readIORef evtRef
11         case evt of
12           Nothing -> do
13             currentOutputRef <- readIORef switchOr
14             syncSDRef currentOutputRef switchOr
15           Just c -> do
16             (nextOutRef, nextStepper) <- compile (next c) ir
17             writeIORef outRef nextOutRef
18             writeIORef stepperRef nextStepper
19             writeIORef evtRef Nothing
20             switchStepper
21   pure (switchOr, switchStepper)
```

On line 2 and 3, the first signal function to run is compiled. This produces an output reference `firstOutRef`, an event reference `evtRef` and a stepping `firstStepper`. On line 4, the reference `outRef` of type **`IORef`** `(`**`SDRef`** `o)` is allocated and contains initially `firstOutRef`. Similarly, `stepperRef` of type **`IORef`** `(`**`IO`** `())` contains the firstStepper for the signal function. The output reference for the whole switch, `switchOr`, of type **`SDRef`** `o` is then created and contains the initial value of `firstOutRef`. With these, it's possible to construct `switchStepper`, the switch's stepping action. `switchStepper` starts by retrieving the stepping action from the currently running signal function and runs it (line 8 and 9). Then, it checks whether an event occurred: if not (line 12), it retrieves the current output reference and uses its content to set the content reference for the switch's output reference `switchOr`; if an event occurred (line 15), it then compiles the next signal function (by applying the continuation `next` to the value carried by the event), which gives a new output reference and a new stepper. These are used to set the content of the `outRef` and `stepperRef`. Since the only action ever writing to `evtRef` is `firstStepper`, and that action will never be run again, `evtRef` must be set to contain **`Nothing`** (line 19), otherwise it will forever contain an event value and `switchStepper` will always switch[a]. Finally, since switching is instantaneous, `switchStepper` runs itself again so that it can run the newly computed stepper.

Unfortunately, this implementation has a subtle, but fatal, flaw. A common pattern in Yampa is to have switches switching back on themselves (or generally on another switch). This is the case for instance when defining the `bouncing` signal function, which models an object in free fall bouncing on the ground. The full definition was given in §2.3.3, but essentially boils down to the following, since the definition of `bounceAux` is irrelevant here:

```
bouncing (y0, v0) = switch bounceAux bouncing
```

[a] Alternatively, one could make sure that `switchStepper` does not read the value of `evtRef` after a switch occurred.

What happens when bouncing is ran after having been compiled with the method above? Initially, the switchOr reference contains a reference to the output reference of bounceAux and similarly, stepperRef contains a reference to the stepping action for bounceAux. After the first switch however, switchOr now contains a reference to the output reference of the new instance of bouncing and similarly for stepperRef. When the switchStepper actions runs for the outermost switch, it has to dereference its stepperRef, runs the action it contains, which is also the stepper for a switch, which will dereference its own stepperRef. A similar chain of operations also exist for setting the output reference of the outermost switch. As bouncing keeps switching, this chain becomes longer: leading both to a time-leak and a space-leak (since there are more and more references being kept alive in memory).

The two problems that need solving are therefore the problem of having variable output references, and variable stepping actions. This is the subject of the next two sections.

3.4.3.1 — *Changing output references* There are two fixes to the problem of varying the output references. The first one is to introduce a notion of variable references. Instead of using **IORef**s directly in **SDRef**, we could use a type like so:

```
data Ref a = FixRef (IORef a)
           | VarRef (IORef Bool) (IORef (Ref a))
```

A reference is either a fixed reference or a variable reference, pointing on another reference. In the latter case, there is also a reference on a Boolean flag that, when **True**, indicates that the variable reference will always point to the same **Ref**: it has become non-variable. This means that it has become an unnecessary indirection and can be 'skipped': another variable reference pointing to it can simply point at the next one. This can be implemented in the function reading or writing to the reference and is simply a form of path compression, like what is typically done when implementing type-checkers with destructive unification [49] or in implementations of union-find data structures [2]. Using such a variable reference in bouncing allows for the output (variable) reference of the outermost switch to always refer to the output (variable) reference of the innermost switch, itself directly pointing at the output reference of bounceAux. This brings down the number of indirections between signal references to a constant amount and, if bouncing is used within another signal function, its variable reference may also be skipped.

This technique has the advantage of allowing to not change the signature of the compile function. The only downside is added indirection to access the content of references [a]. Another solution, which is pursued here, is to instead keep the type of references unchanged, but modify the compile function. Recall in §3.4.2 that another signature

49. Duggan et al., 'Explaining Type Inference'. 1996

2. Aho et al., *Data Structures and Algorithms*. 1983

[a] In practice, this extra complexity shows in terms of performance. Benchmarking showed that reading from an **IORef** is 2 times faster than reading from a **FixRef**. Reading from a **VarRef** pointing to a **VarRef** pointing to a **FixRef**, like what happens with a recursive switch, is 4 times slower than reading from an **IORef**. The machine on which the benchmarks were run was the same machine used for the benchmarks in §3.5 using the same benchmarking library.

for the `compile` function was considered where both the input and output reference were passed in, and the `compile` function only produced a stepping action. While it is not practical to do so in general, as it neutralizes the benefits of making routing explicit, this is not generally a good solution, because of routing, it is a good solution for switching. Indeed, providing the output reference to the `compile` function allows to share it between the initial signal function that the switch runs and the function that the switch computes after an event occurred. There is no need for doing any synchronisation between several references and nothing special has to happen at switching time with regard to the output reference of the switch. The solution therefore, is to make a compromise by optionally passing the output reference to the compilation function, that will optionally produce the output reference. The following type would suffice:

```
SDRef i → Maybe (SDRef o) → IO (Maybe (SDRef o), IO ())
```

However, it would be better to encode the relation that when one of the `Maybe (SDRef o)` is `Nothing`, then the other must be `Just`. This constraint can be encoded in GHC's type system, with the same tools that were used to constrain signals. Let's introduce a tagged-optional type that witnesses with a type-level Boolean which constructor it is made from:

```
data Opt (b :: Bool) a where
  None ::      Opt False a
  Some :: a → Opt True a
```

Type families [38, 39] are type-level functions that are treated as synonyms by GHC's type-checker. This allows to define a type-level `Not` function:

```
type family Not (b :: Bool) :: Bool where
  Not False = True
  Not True  = False
```

and use it to encode the desired invariant:

```
compile :: forall b. SDRef i
        → Opt b (SDRef o)
        → IO (Opt (Not b) (SDRef o), IO ())
```

When passing no output reference to compile, with the `None` data constructor, this causes `b` to be unified to `False` and `Not b` to be unified to `True`. Hence the output of compile must be of type `Opt True (SDRef o)`, which is necessarily constructed from the `Some` data constructor and must contain an output reference. And vice-versa when an output reference is passed to the `compile` function, no output reference can be produced by it.

38. Chakravarty et al., 'Associated Type Synonyms'. 2005 — 39. Chakravarty et al., 'Associated Types with Class'. 2005

compile must work in both cases. Consider the implementation of the compilation of a router with this type. In the case where no output reference is passed, then the references can be routed like they were originally; however, in the case where an output reference is given, it will be necessary to synchronize both references at runtime:

```
compile (SFRouter rr) ir None =
  pure (Some (routeRef rr ir), pure ())
compile (SFRouter rr) ir (Some or) =
  pure (None, syncSDRef (routeRef rr ir) or)
```

A possibly simpler solution could have been to have two compile functions: one where the output reference is provided and one where it is not. However, having a single function is advantageous as in many cases the code for the compile function is identical in both cases. The present approach avoids duplicating this code. For instance, serial composition can be compiled without knowing if it has been passed an output reference:

```
compile (sf :>>>: sg) ir or = do
  (Some orf, stprF) ← compile sf ir None
  (nor, stprG) ← compile sg orf or
  pure (nor, stprF >> stprG)
```

Like in the simpler setting, it is possible to reuse the output reference of the first signal function as the input to the second.

The definition of switch in this new setting will be given in the next section, after the problem of representing variable stepping actions efficiently has been addressed.

3.4.3.2 — *Changing stepping actions* So far, the type of stepping actions has been limited to `IO ()`. To solve the problem of variable stepping actions, a type of steppers is introduced which allows for the existence of variable stepping actions, which are actions that return a new stepper to execute at the next time step:

```
data Stepper = FStpr (IO ())
             | VStpr (IO Stepper)
```

Some functions to execute steppers are given below, also returning the stepper to execute at the next step:

```
execStepper :: Stepper → IO Stepper
execStepper (FStpr stpr)  = stpr >> pure (FStpr stpr)
execStepper (VStpr vstpr) = vstpr
```

and to sequence two steppers:

```
seqStepper :: Stepper -> Stepper -> Stepper
seqStepper (FStpr s1) (FStpr s2) = FStpr (s1 >> s2)
seqStepper (VStpr vs1) (FStpr s2) = VStpr $ do
  s1 <- vs1
  () <- s2
  pure (seqStepper s1 (FStpr s2))
seqStepper (FStpr s1) (VStpr vs2) = VStpr $ do
  () <- s1
  s2 <- vs2
  pure (seqStepper (FStpr s1) s2)
seqStepper (VStpr vs1) (VStpr vs2) = VStpr $ do
  s1 <- vs1
  s2 <- vs2
  pure (seqStepper s1 s2)
```

Note how, in the case of a variable stepper, the recursive call to `seqStepper` allows the simplification of the stepper as parts of the network evolve over time, exactly how the continuation-based embedding allowed for such simplifications. In the same spirit, it would be possible to extend the **`Stepper`** type with additional constructors to represent steppers for applications of `arr` or the identity signal functions to open up more dynamic optimisations opportunities. The latter can also be used also to denote steppers that do nothing, like the ones needed for routers.

With this new machinery and the new way references are handled from the previous section, `switch` can be implemented in the following way:

```
compile (Switch sf mknext) ir or = do
  (Some (PRef orf (ERef evtRef)), stprSF) <- compile sf ir None
  let switchStpr stpr = VStpr $ do
        stpr <- execStepper stpr
        evt <- readIORef evtRef
          case evt of
            Nothing -> pure (VStpr (switchStpr stpr))
            Just c -> do
              (None, stprNext) <- compile (mknext c) ir (Some orf)
              stprNext <- execStepper stprNext
              pure stprNext
  case or of
    None -> pure (Some orf, switchStpr stprSF)
    Some or -> do
      let updateRef = readSDRef orf >>= writeSDRef or
          stpr = seqStepper (switchStpr stprSF) (FStpr updateRef)
      pure (None, stpr)
```

3.4.4 — *Loops*

The absence of a generic loop combinator makes the compilation of SFRP a lot simpler than it would have been otherwise. Let's look at the compilation process for the **IntegrLoop** constructor. It assumes an output reference is passed in, the case when it is not is not much different but presenting just this one case avoids some distracting noise in the code:

```
compile (IntegrLoop init lf) ir (Some or) = do
  memir ← lift $ newSDRef init
  memor ← lift $ newSDRef init
  (None, stpr) ←
    compile lf (ir `CP` memir) (Some (or `CP` memor))
  let integrateStep = do
        x  ← readSDRef memir
        x' ← readSDRef memor
        dt ← getDTime
        let nx = integrate dt x x'
        writeSDRef memir nx
  pure (None, stpr `seqStepper` FStpr integrateStep)
```

The implementation uses to auxiliary references `memir`, which contains the currently computed value of the integral and `memor` which will contain the value of the derivative produced by the signal function `lf`. The stepper for the whole signal function simply consists of running the inner signal function's stepper first, which will make use of the current value for the integral stored in `memir`; and then update the value stored in `memir` using the `integrate` function (defined in §3.2.1.3).

This compilation scheme could form the basis of more involved compilation method, for instance by allowing it to run the inner signal function at various points in time, thus enabling the use of more advanced numerical integration technique.

3.5 — Evaluation

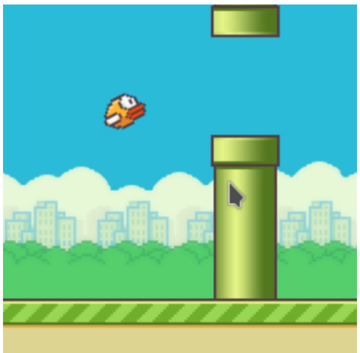
In this section, SFRP is evaluated. Both in terms of ease of use compared to Yampa and in terms of performance, to verify that the scalability objective is attained. The implementation can be found on Gitlab[a].

3.5.1 — *Ease of writing*

In order to evaluate how different SFRP and Yampa are from a user perspective, a modified version of the game Flappy bird written using Yampa [93] was modified to use SFRP[b].

For a large part of the code, the port was as straightforward as the port of the falling object example from §3.3, mostly a matter of surrounding the code written in *proc*-notation by quasi-quote brackets

Figure 3.4: A version of the Flappy bird game running with SFRP



[a] `https://gitlab.com/chupin/scalable-frp`

93. Mahuet, *Flappy Haskell*. 2015

[b] The SFRP version can be found at `https://gitlab.com/chupin/flappy-haskell/tree/scalable`

and modifying the few parts where the syntax is different. The only difficult point was handling inputs to the network. The original game used, as is common, a record of several fields, essentially containing information coming from the Graphical User Interface (GUI) such as the mouse position, mouse clicks, etc. In the new setting, this record is a collection of heterogeneous signals: the mouse position is a continuous signal while the mouse clicks are events. However the only collection of signals SFRP supports is nested-pairs. The choice was then between getting rid of the input record in favour of nested pairs of signals, which is inelegant and tedious to write or read; or slightly 'lying' and keeping the input record as a continuous signal (even if its components are not all continuous signals) and writing accessor signal functions that convert each field into the relevant signal with the right kind. The latter solution was used in this case.

Flappy bird, being a light game, did not directly benefit from an increase in performance from SFRP, since the GUI was, by far, the bottleneck. Turning off the GUI, we were able to measure that, although the Yampa version was already running at a respectable 30 000 iterations per second, the SFRP version was running at 90 000 iterations per second, thus a three-fold improvement. This particular measurement was done on a PC with a 4-core Intel i3-7100T @ 3.40 GHz with 8 GB of memory.

3.5.2 — *Precise performance measurements*

To test the relative performance of SFRP and Yampa in a more meaningful and systematic way, networks with various sizes and characteristics were auto-generated. These networks, in their Yampa and SFRP versions, were then benchmarked using the Criterion library[a].

3.5.2.1 — *Random network generation* The networks were generated in *proc*-notation from a little dedicated Haskell program. Doing so is much easier than generating networks in 'combinator form' and allows to measure the hidden impact of wiring resulting from the *proc*-notation desugaring over the network in a clearer way.

The generated network has one input and one output, both continuous signals of doubles as are all intermediate signals in the network. Using continuous signals means that their representation is not optimized in any way, unlike step signals, meaning that any improvement over Yampa can be attributed to the difference in routing. The generator makes sure that every signal is used at least once in the computation of the output, as to not have parts of the network removed. The network is made of three basic blocks: integrals, with one input and one output; sums and negations of two input signals in one output and switches, with one input and one output and a subnetwork.

The network is generated with a target size. Each block counts for 1 except for switches, that count for the size of their subnetwork.

[a] http://hackage.haskell.org/package/criterion

The generator controls the proportion of switches as well as the average number of iterations at which a switch should occur. It also makes sure that not all switches switch in the same iteration by adding some random noise to the number of iteration a switch should switch after [b]. When a switch is generated, a subnetwork is generated with a size randomly chosen between 1 and the target size of the network it is in. The switch switches back on itself when an event occur (like `bouncing`). This is both in order to model a very common pattern in this kind of programs and to prevent switches from disappearing entirely from the network, leading to dynamic simplifications. An example for a generated network is given in figure 3.5.

Once a network has been generated, the time taken for it to run 100 000 iterations is benchmarked, with SFRP and YAMPA. The measurements were done on a PC with a 8-core Intel Xeon W-2123 @ 3.60 GHz and 32 GB of memory [a]. Results are presented in figure 3.6. The graphs show the average time a network of a given size has taken to execute 100 000 iterations for each library. The average is taken over 1000 trials. Each trial attempts to average out measurement artifacts, such as the computer's clock precision, potentially by running the program to benchmark several times [113]. The ratio of the speedup of SFRP network over the YAMPA network is plotted on the same graph, as the ratio of the two average running times. The general observation is that SFRP is significantly faster in all cases, except in one, rarely encountered in practice, where it is on par with YAMPA.

3.5.2.2 — *Performance of static networks* Benchmarks for static networks (with no switches) can be found in figure 3.6a. They show that networks are consistently faster when using SFRP, by an order of magnitude for networks of size larger than 10. This confirms that wiring is a major cost in YAMPA compared to SFRP. In particular, one can observe that YAMPA's running time is quadratic in the size of the network, while SFRP's is linear.

This can be explained by the following observation: as the network grows, the number of combinators introduced for the purpose of wiring grows linearly, since there has to be at least one per *proc*-notation. But, the work these routers perform also grows linearly, since there is a linearly increasing number of signals to route. This observation will be confirmed in §3.5.2.4, which will focus on the cost of switching on SFRP and shows that it exhibits the same behaviour when it has to switch on large networks.

3.5.2.3 — *Evaluation of performance with the number of switches* Figure 3.6b and 3.6c show two of the benchmarks for networks that were generated with respectively 10% and 25% of each statement being a switch and then run by triggering each switch every 50 iterations on average. These benchmarks, by their very nature cannot be used to precisely predict a trend. Indeed, only the proportion of switches in the net-

[b] If a switch should switch every n iterations on average, the noise is chosen between $\pm \min\left(10, \frac{n}{2}\right)$

[a] This amount of memory was not necessary to run the benchmarks, but to compile them, as it seems GHC is very memory hungry when it encounters programs with large literals, as is the case with these auto-generated networks.

113. O'Sullivan, *Criterion, a New Benchmarking Library for Haskell.* 2009

```
goYampa = proc a → do
  b  ← arr (uncurry (+)) ⤙ (a,a)
  c  ← sw1               ⤙ b
  h  ← sw2               ⤙ c
  o  ← integral          ⤙ b
  p  ← arr (uncurry (+)) ⤙ (h,h)
  q  ← integral          ⤙ b
  r  ← arr (uncurry (-)) ⤙ (b,c)
  s  ← arr (uncurry (+)) ⤙ (p,p)
  t  ← sw3               ⤙ s
  w  ← integral          ⤙ h
  x  ← arr (uncurry (-)) ⤙ (c,p)
  y  ← arr (uncurry (+)) ⤙ (x,w)
  z  ← arr (uncurry (+)) ⤙ (r,t)
  aa ← arr (uncurry (-)) ⤙ (y,z)
  bb ← arr (uncurry (-)) ⤙ (aa,q)
  cc ← arr (uncurry (+)) ⤙ (o,o)
  dd ← arr (uncurry (+)) ⤙ (bb,bb)
  ee ← arr (uncurry (+)) ⤙ (cc,dd)
  returnA                ⤙ ee
  where trigger stop = proc _ → do
          t ← integral ⤙ 1.0
          evt ← up ⤙ t - stop
          returnA ⤙ evt

        sw1 = switch (fun1 &&& trigger 6) (const sw1)
        fun1 = proc ff → do
          d ← integral          ⤙ ff
          e ← arr (uncurry (+)) ⤙ (d,d)
          f ← integral          ⤙ ff
          g ← arr (uncurry (+)) ⤙ (f,e)
          returnA               ⤙ g

        sw2 = switch (fun2 &&& trigger 12) (const sw2)
        fun2 = proc gg → do
          i ← arr (uncurry (-)) ⤙ (gg,gg)
          j ← arr (uncurry (-)) ⤙ (gg,i)
          k ← integral          ⤙ gg
          l ← integral          ⤙ i
          m ← arr (uncurry (+)) ⤙ (j,k)
          n ← arr (uncurry (-)) ⤙ (l,m)
          returnA               ⤙ n

        sw3 = switch (fun3 &&& trigger 7) (const sw3)
        fun3 = proc hh → do
          u ← arr (uncurry (+)) ⤙ (hh,hh)
          v ← arr (uncurry (+)) ⤙ (u,hh)
          returnA               ⤙ v
```

Figure 3.5: Example Yampa signal function generated for benchmarking

This particular network was generated with a target size of 25 statements, with a proportion of switches of 10%, switching on average every 10 iterations. The `trigger` signal function is in charge of triggering an event after a certain time as elapsed by integrating a signal of slope 1. The network has been simplified by hand to make it readable.

work is controlled, not the size, which is picked randomly between just one statement and the target size. However, as the number of switches increases, the gain from using SFRP consistently shrinks. Not only because SFRP becomes slower, but also because YAMPA becomes faster.

The reason for this gain is quite subtle, and has to do with the fact that, when a network contains a switch, its structure is simpler with regard to routing. This is eminently beneficial to YAMPA but not particularly to SFRP. Consider the case of a network of size *n*, without switches. To produce its output, it must wire all *n* signals together. Now consider a network of the same size, but made of two switch blocks, each of size $\frac{n}{2}$. Since each switch is made to take one input and produce one output, to produce their output, both subnetworks must wire $\frac{n}{2}$ signals, and then the two output signals must be wired together. Since the cost of routing is quadratic in the size of the network, it is more efficient to do the latter that the former, as witnessed by the benchmark results [a].

This is also why YAMPA exhibits somewhat better behaviour in practice than in the first benchmarks, since most networks are made in a modular fashion, from small networks linked together, the overall cost of routing is reduced. However, the performance advantage of using SFRP is always significant. It can only increase as the network grows in size, remaining constant at worse.

3.5.2.4 — *Evaluation of the cost of switching* To get a more meaningful idea of the cost of switch, it is more interesting to study networks in a different form. Instead of networks containing some switches, it is easier to generate a purely static network and enclose it in a switch, switching at a certain rate. By that we mean that, if the generated network is n, the network we benchmark is of the form:

```
n' = switch (n &&& r) (\ _ → n')
```

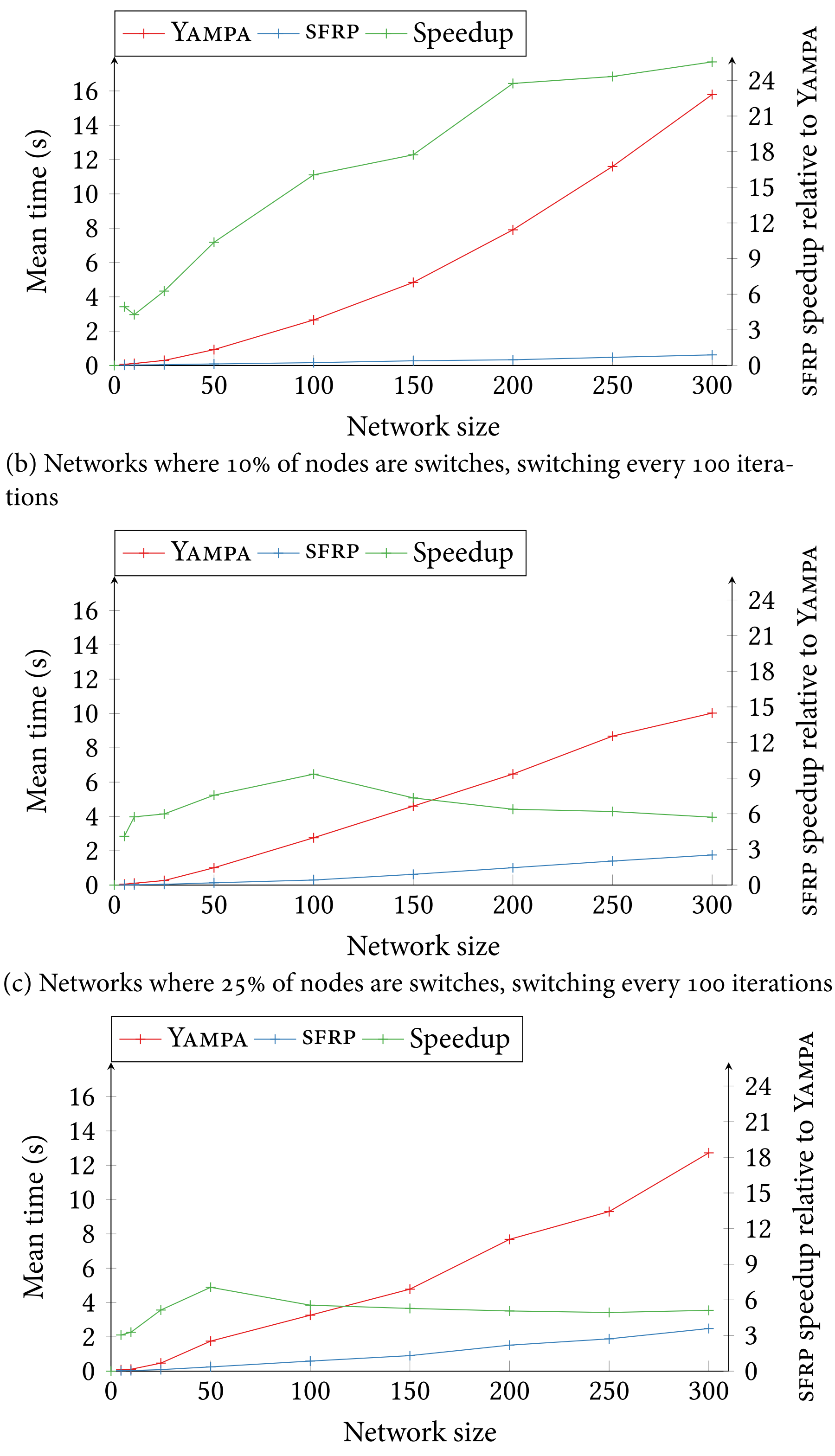
where r is the signal function generating the event for the switch to occur. Some results are shown in figure 3.7. Unlike in the previous case, the switch is made to switch at a set time, since there is only one, there is no point in preventing several switches from switching at the same time.

Overall, YAMPA does not suffer a lot from having to handle infrequent switching events. There is no significant difference in runtime whether a switch occurs every 50 iterations (figure 3.7a), 10 iterations (figure 3.7b) or when no switch occur at all (figure 3.6a). After all, switches from the point of YAMPA are not very different from other signal functions. This is not the case for SFRP which significantly slows down when switches get more frequent. This is expected since its main advantage resides in avoiding the cost incurred with routing, which must be paid every time there is a switch. In the extreme case where a switch occurs at every iteration (figure 3.7c), SFRP is as slow as YAMPA and clearly exhibits the same quadratic time complexity in the size of

[a] The example program in figure 3.5 gives another illustration of that phenomenon. Even though it has, in total, around 30 statements, there are at most 19 signals to wire together in the main signal function.

the network. This confirms that routing is the cause of the quadratic behaviour: since SFRP is forced to compile a new network at every iteration, it must route references at every iteration, much like Yampa does normally. Fortunately, networks written this way are rarely seen in practice and remain small.

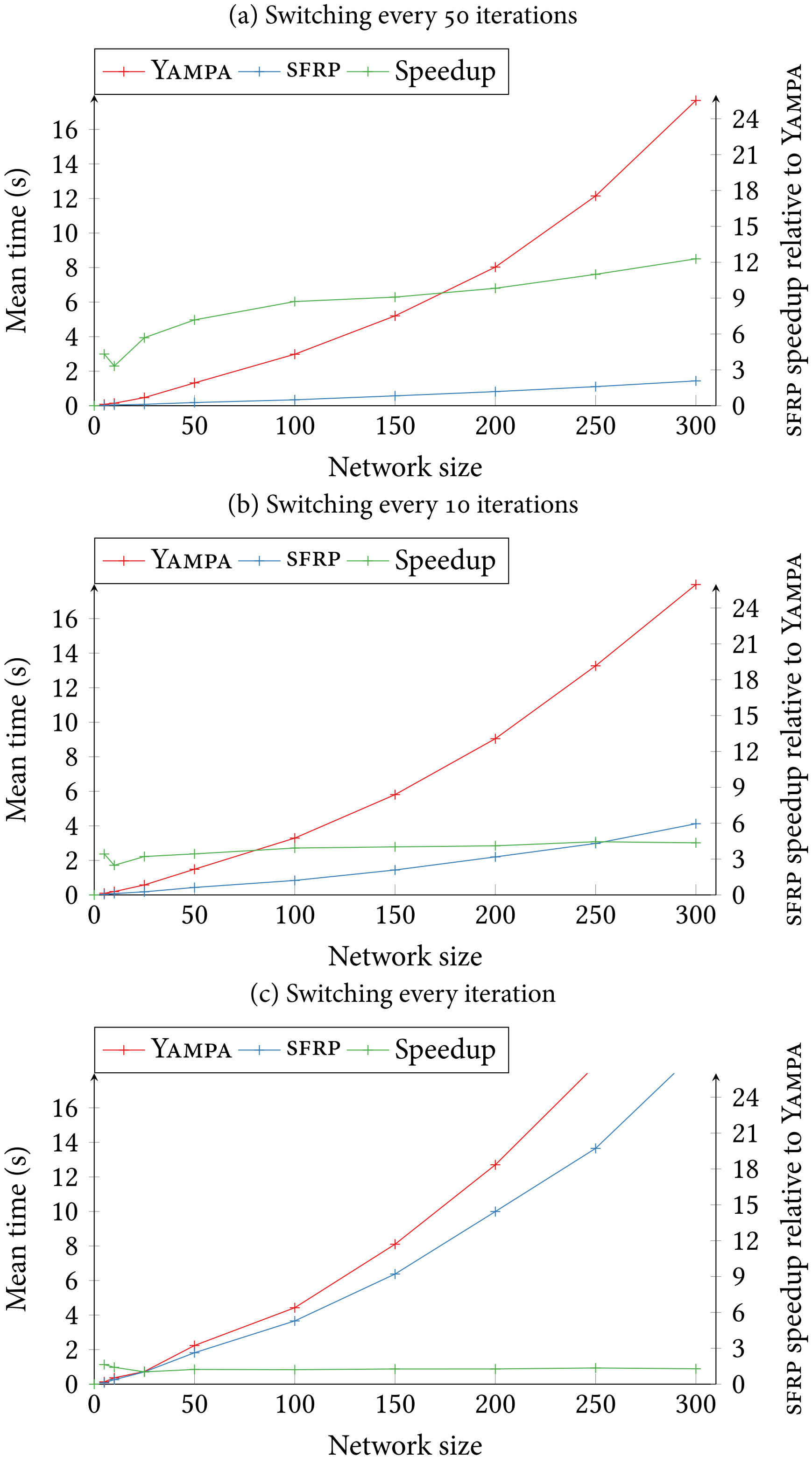
Figure 3.6: Benchmark results for arbitrarily generated networks. Average runtime and speedup over 100 000 iterations


(a) No switches

(b) Networks where 10% of nodes are switches, switching every 100 iterations

(c) Networks where 25% of nodes are switches, switching every 100 iterations

Figure 3.7: Benchmark results for networks with a single switch enclosing an otherwise switchless network. Average runtime and speedup over 100 000 iterations

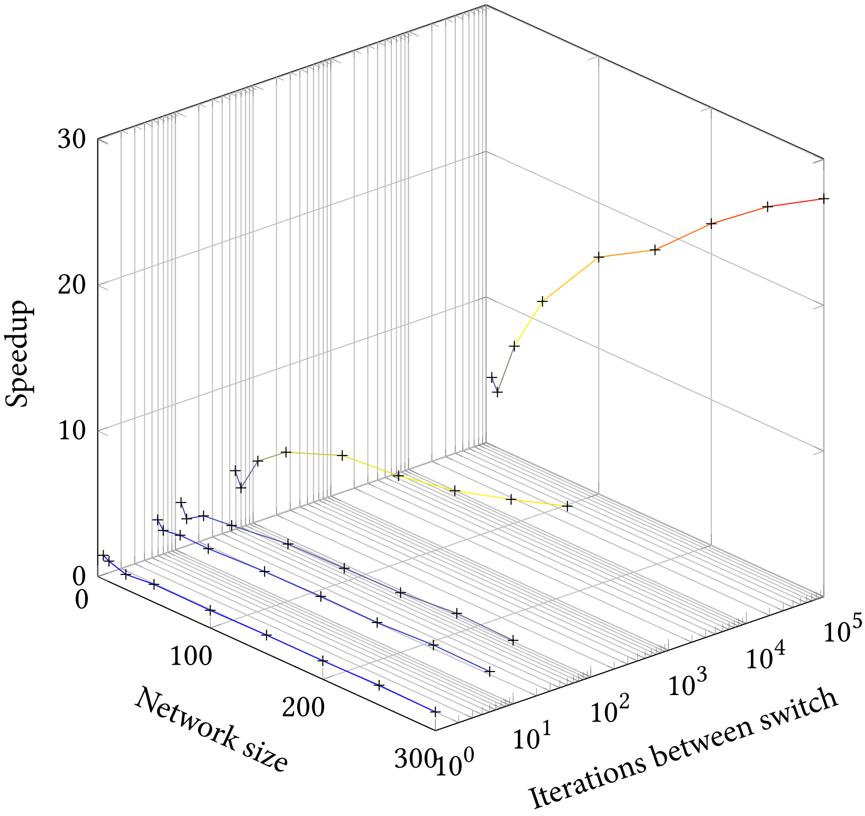


(a) Switching every 50 iterations

(b) Switching every 10 iterations

(c) Switching every iteration

Figure 3.8: 3-dimensional visualisation of the dependency between size, switching frequency and speedup



Related works & conclusions 4

4.1 — Related work

4.1.1 — *Causal modelling languages*

Simulink Simulink [96] is a graphical block-diagram environment embedded within Matlab, first introduced in 1984. It is widely used in industry, where it is a standard tool for the design of embedded control systems [19]. The numerical simulation capabilities of Simulink are meant both as a help for designing the model, and as a tool for checking the validity of generated code from the model. Simulink suffers from several shortcomings. It is based on fairly weak semantics foundations, which make it easy to write nonsensical models or whose meaning is unclear. While some will result in runtime errors, which in this case is the lesser evil, many will simply produce erroneous results [21]. Simulink also uses graphical rules to determine the meaning of otherwise ambiguous models: this means that the meaning of a model can depend on the coordinates of a block on the diagram and not only on the topology of the diagram [133].

96. Mathworks, *Simulation and Model-Based Design*. 2020

19. Bouissou et al., 'An Operational Semantics for Simulink's Simulation Engine'. 2012

21. Bourke et al., 'A Synchronous Look at the Simulink Standard Library'. 2017

133. Scaife et al., 'Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre'. 2004

Zélus Zélus [20] is a synchronous language conservatively extended with ode. It descends from Lucid Synchrone [124], from which it inherits support for higher-order functions. Its approach as a hybrid language heavily influenced the design of sfrp. Zélus imposes a strict boundary between the synchronous (discrete) world and the continuous world, which is checked at compile-time through a type-system [10]. It is built on strong semantics foundations [12] based on non-standard analysis [130]. The intention of Zélus is, like Simulink, to provide a unified framework for the conception of discrete systems (such as control systems), using the synchronous dataflow part of Zélus, and the simulation of these systems in their environment, modelled with differential equations. As such, Zélus is more oriented towards safety than expressivity, like synchronous languages in general. For instance, it doesn't support unbounded structural dynamism, since it would make it impossible to generate code with strong memory and timing guarantees.

20. Bourke et al., 'Zélus, a Synchronous Language with ODEs'. 2013

124. Pouzet, *Lucid Synchrone*. 2006

10. Benveniste et al., 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. 2017

12. Benveniste et al., 'Non-Standard Semantics of Hybrid Systems Modelers'. 2012

130. Robinson, *Non-Standard Analysis*. 1974

Ptolemy Ptolemy[53] is a modelling environment based on an actor-oriented paradigm. In Ptolemy, a model consists of a set of actors which execute concurrently and interact together through a given set of rules called a domain. The choice of domain dictates, among other things, the model of time used by the model, either discrete or continuous [144]. What is interesting is that different domains, with

53. Eker et al., 'Taming Heterogeneity - the Ptolemy Approach'. 2003

144. The Ptolemy Project, *System Design, Modeling, and Simulation Using Ptolemy II*. 2014

different notions of time (and even different requirements for the rate of change of time) can also be made to interact, creating heterogeneous models. To support this, Ptolemy uses multiform time, which allows for the passing of time at different rate across multiple domains, while still providing a globally coherent notion of time. Particularly relevant to hybrid modelling languages is Ptolemy's representation of infinitesimal time-steps, with the notion of superdense time. Superdense time allows to divide an instant in time into many microsteps. Each microstep takes an infinitesimally short time to pass but they can still be ordered in time. This was used to propose a formal approach at defining the semantics of hybrid modellers [91].

91. Lee et al., 'Operational Semantics of Hybrid Systems'. 2005

4.1.2 — *Functional Reactive Programming approaches*

55. Elliott, 'Push-Pull Functional Reactive Programming'. 2009

Push-pull frp Elliott, in his 2009 paper [55], revisits frp. He provides a modernised monadic and applicative interface, but the main motivation is to resolve the tension between supporting continuously changing values efficiently, which calls for a pull-based implementation, and events, which calls for a push-based implementation. Push- and pull-base refer to the way the network runs to produce results: push-based implementations wait for an event to be produced as an input, and produce a result that pushes through the network; while pull-based wait for a result to be demanded by the user of the network and work backward through the network to produce it. The answer is to combine both, and the paper achieves this through an elegant derivation from a denotational semantics. However, the basic behaviours [a] are represented by piece-wise constant behaviours (like the step signals of sfrp) with overlaid *known* functions, accounting for the behaviour between the points of discrete changes. For example, the time behaviour is in essence a single step with an overlaid identity function. This means that the basic behaviours cannot account for the case where the behaviour is not predictable until the next discrete change, such as is the case with integration (in general). A separate interface is provided for this kind of behaviour, but the paper does not give many details and it is thus difficult to comment on the efficiency. It is also unclear to what extent feedback is supported as the paper does not provide the required primitives, neither for behaviours nor for events. As to the implementation, it relies on an 'unambiguous choice' operator that works by spawning threads, using `unsafePerformIO` to provide a pure interface, and then picking the first available result. The specific setup is such that this indeed is safe and does not violate referential transparency. The implementation is thus very different from that of sfrp, and its ultimate efficiency hinges on how well the runtime can handle lots of light-weight threads and how efficiently the sophisticated machinery required to keep everything pure from a user perspective can be compiled. In contrast, the code that ultimately is executed when sfrp programs are run is in essence just conventional, single-threaded

[a] In some implementations, continuous-time signals are called *behaviours*. We keep this terminology if it is used in the work being discussed.

imperative code. The paper does not provide any performance evaluation.

Stream-based FRP Patai [118, 119] proposes a representation of reactive programs as infinite streams of values, where streams are constructed using stream generators in a way that make them suitable to be represented as an stepping action in the **IO** monad, the elements of the stream being produced by repeatedly performing the action. The implementation retains a high degree of dynamicity, as streams can be higher-order and recursively defined, to account for feedback for instance. By nature, there is no notion of continuous time in this implementation. Behind the scenes, the implementation is imperative. A stream network is constructed, where streams may depend on others in arbitrary ways. Carefully coordinated sampling and updating is then carried out, with each stream being represented by an imperative variable that indicates that the stream either is being ready to be sampled, or that it has been sampled, along with its current value, to ensure the results of computations are shared and break cycles. The traversal of this graph at each time step thus amounts to dynamic scheduling of the computations, unlike in SFRP where sequential code for reading and writing the imperative signal variables in an appropriate order is constructed once and for all.

118. Patai, 'Efficient and Compositional Higher-Order Streams'. 2011 — 119. Patai, 'Eventless Reactivity from Scratch'. 2009

Ultrametric FRP Krishnaswami and Benton [87] propose a denotational semantics for reactive programs represented as programs over streams, like in Patai's work. Also like Patai's work, time is discrete. The authors propose a language abiding to this semantics and an efficient translation to a low-level imperative dataflow graph that preserves the high-level semantics. The dataflow graph is a network of imperative references containing thunks over streams. The evaluation of these thunks is dynamically scheduled as the values of the head of each stream is required. Upon evaluation, each references is updated to point to a new thunk to the tail of the stream. The flexibility of the stream representation means that dynamic higher-order recursive FRP programs can be expressed in this setting. However, unlike Patai's work, the well-foundedness of recursively defined streams as well as the causality is guaranteed.

87. Krishnaswami et al., 'Ultrametric Semantics of Reactive Programs'. 2011

FRPNow! Ploeg and Claessen [123] propose a new monadic FRP interface, close to the original FRP interface, that by careful construction rules out space leaks and allows monadic **IO** actions to be performed directly from FRP code. They give a denotational semantics and derive an implementation from this, proving that the stated freeness from leaks indeed holds. While the interface provides both behaviours and events, the value of a behaviour can only change at discrete points in time, by switching, and the only assumption regarding time is that there is a total order between time points. Behaviours in this

123. Ploeg et al., 'Practical Principled FRP'. 2015

system are thus akin to step signals: there is no direct support for continuous signals in the SFRP sense. Feedback is limited to behaviours. The implementation is effectful, using Haskell's **IO** monad. Imperative variables are used to share the latest version of events and behaviours, with `unsafePerformIO` used to ensure that those variables are properly shared even though the provided interface is pure, not mentioning the **IO** context. **IO** actions generating primitive events are spawned using threads, and the resulting events are batched into so called rounds to give the illusion of **IO** actions not taking any time. The system keeps track of what computations are ready to run in response to primitive or derived events, ensuring they are run as soon as possible but at most once. Also, computations of which results are no longer needed and that have no observable side effects are also removed. The scheduling of computations is thus done dynamically, and the overhead for frequently changing signals is consequently substantial. Again, we note that this is different from SFRP where the scheduling is done statically.

Causal commutative arrows Causal Commutative Arrows (CCA) [152] are a particular class of arrows that can be put into causal commutative normal form, meaning either a pure function or a single loop containing one pure function and one initial state value. Use of causal commutative arrows has been shown to lead to great improvements in performance over the continuation-based representation presented in §2.4, in part by using an imperative representation. In this representation, the actions modifications to the internal state are scheduled statically, however it does not tackle the problem of wiring in the way we do, which remains fully dynamic. Note that, although feedback is supported, it always come with an implicit delay. Unlike SFRP or YAMPA, CCA do not support dynamic changes to the network structure.

152. Yallop et al., 'Causal Commutative Arrows Revisited'. 2016

Reactive Banana Reactive Banana [4] is a first-class FRP library, oriented around programming an event network between signals, which is then compiled into a stepping action. Although Reactive Banana supports notionally continuous behaviours, the implementation is completely event driven, meaning the network only steps forward when an event occurs. Reactive Banana networks can be dynamic in a way similar to other implementations, by having events carrying new behaviours to switch to.

4. Apfelmus, *Reactive-Banana*. 2011

4.2 — Future work and conclusions

This work showed how the performance of arrowized FRP programs could be greatly improved by making use of an imperative representation, similar to how synchronous dataflow languages are compiled, and helped by a precise type-level description of a network. Quantitative evidence of this improvement was presented. The approach is also mature enough to, in many cases, be used as a (mostly) drop-in replacement for YAMPA.

The current implementation of SFRP is however lacking in some respects compared to Yampa. In particular, SFRP does not yet support collection-based switches and 'freezing' of running signal functions [110]. Also, the arrow `loop` combinator for feedback is not fully supported. The problem here is that `loop` calls for an instantaneous feedback edge which makes static scheduling more difficult. In principle this can be solved by analysing the dataflow graph: there must be some decoupling somewhere in a feedback loop to make it well-defined, but the decoupling need not necessarily be along the back edge itself. The solution used here, to have the feedback signal always decoupled through a delay, is quite pragmatic and also fairly common. Indeed, many of the systems considered in the section on related work also do not support instantaneous feedback edges for various reasons.

The new interface of SFRP makes it amenable to other implementation techniques, such as compilation to lower level languages (either ahead-of-time or just-in-time); and opens the door to use more advanced integration techniques. Since this was not the original goal of this work, which was focused on providing better performance, this has not yet been investigated, but would provide a very exciting research avenue for future work.

Finally, to further improve the scalability, it would be interesting to explore systematic incremental evaluation. As discussed briefly in section 2.4.5, the SFRP implementation has an appropriate structure, thanks to direct communication between producers and consumers through shared variables.

[110] Nilsson et al., 'Functional Reactive Programming, Continued'. 2002

Part II

Modular compilation for Functional Hybrid Modelling

Introduction to non-causal modelling 5

The previous part focused on work done on causal modelling languages and causal reactive languages. Causal modelling languages allow for the manipulation of a limited set of differential equations: directed equations, or Ordinary Differential Equation (ODE). This approach has many advantages: implementations are simple and there is a clear mapping from the model to the implementation. Conceptually, it is easier to define precise semantics for causal modelling languages as well.

However, transforming an arbitrary set of undirected differential equations into a causal model is a somewhat involved and tedious task. This is particularly true in a hybrid setting or in the presence of non-linear equations wherefrom extracting a causal equations is sometime impossible, at least without significant symbolic manipulation. Overall, this approach hurts modularity. Since the way an equation depends on the context in which it is used, it is difficult to write reusable model fragments.

The alternative is then to allow for the manipulation of undirected equations directly, which is the approach taken by non-causal modelling languages. This implies the existence of simulation techniques for systems of undirected differential equations. This chapter serves as an introduction to these techniques, showcasing particularly some of their limitations which have consequences for the implementation based around them.

The chapter is organised as follows. §5.1 focuses on listing the shortcomings of the causal modelling approach which can be solved by directly manipulating undirected equations. §5.2 goes over the numerical and symbolic methods used for the simulation of undirected differential equations. §5.3 explains the specific difficulties associated with higher-index systems, a class of equation system which require additional symbolic processing before their simulation can be undertaken. §5.4 introduces some of the problems associated with the initialisation of systems of undirected equations. Finally, §5.5 introduces Functional Hybrid Modelling (FHM), a paradigm that allows for modelling using undirected equations, and HYDRA, a language based on these ideas.

5.1 — Limits of causal modelling

Chapter 2 gave an introduction to causal modelling. It showed how it was possible to extract directed differential equations by hand that

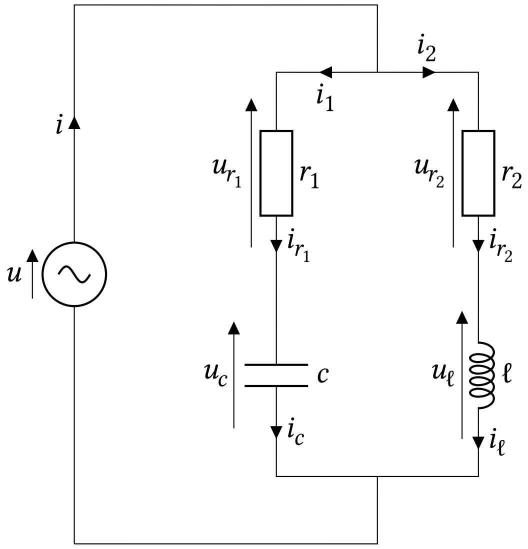


Figure 5.1: A more complicated electrical circuit

could be translated, either by hand again or using a causal modelling language to perform simulation.

This section discusses the limits of this approach. In particular, it shows how supporting only directed equation leads to modularity issues, especially in the context of hybrid models. It also demonstrates more subtle difficulties that arise in the presence of non-linearity, when extracting a causal model is made much harder.

The aim of this section is to motivate the need for languages able to handle *undirected* equations directly.

5.1.1 — *Modularity*

Let's consider the electrical circuit in figure 5.1. The circuit is modelled with the following 12 equations of 12 unknowns:

$$u = u_0 \sin(2\pi f t + \varphi) \tag{5.1a}$$

$$u_{r_1} = r_1 i_{r_1} \tag{5.1b}$$

$$c u_c' = i_c \tag{5.1c}$$

$$u_{r_2} = r_2 i_{r_2} \tag{5.1d}$$

$$\ell i_\ell' = u_\ell \tag{5.1e}$$

$$i = i_1 + i_2 \tag{5.1f}$$

$$i_1 = i_{r_1} + i_c \tag{5.1g}$$

$$i_{r_1} = i_c \tag{5.1h}$$

$$i_2 = i_{r_2} + i_\ell \tag{5.1i}$$

$$i_{r_2} = i_\ell \tag{5.1j}$$

$$u = u_{r_1} + u_c \tag{5.1k}$$

$$u_{r_1} + u_c = u_{r_2} + u_\ell \tag{5.1l}$$

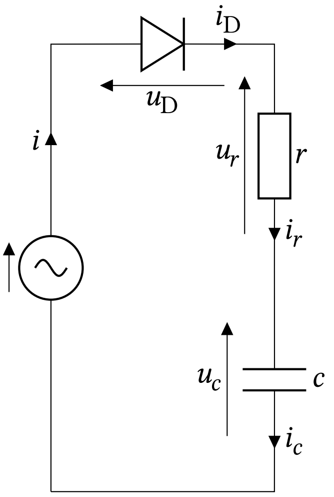
The circuit uses two resistors so, when implementing the circuit in a causal modelling language, it would be desirable if a single model of a resistor could be written and reused. However, this is not possible in this case. Indeed, the causal version of the system of equations above is as follows:

$$\begin{align}
u &= u_0 \sin(2\pi f t + \varphi) \tag{5.2a}\\
u_{r_1} &= u - u_c \tag{5.2b}\\
i_{r_1} &= \frac{1}{r} u_{r_1} \tag{5.2c}\\
i_c &= i_{r_1} \tag{5.2d}\\
u_c' &= \frac{1}{c} i_c \tag{5.2e}\\
i_1 &= i_{r_1} + i_c \tag{5.2f}\\
i_{r_2} &= i_\ell \tag{5.2g}\\
i_2 &= i_{r_2} + i_\ell \tag{5.2h}\\
u_{r_2} &= r_2 i_{r_2} \tag{5.2i}\\
u_\ell &= u_{r_1} + u_c - u_{r_2} \tag{5.2j}\\
i_\ell' &= \frac{1}{\ell} u_\ell \tag{5.2k}\\
i &= i_1 + i_\ell \tag{5.2l}
\end{align}$$

Notice that equations 5.2c and 5.2i, both derived from Ohm's law, are different: the first is used to compute the current through r_1 and the second is used to compute the voltage across r_2.

This is unfortunate since, in general, a differential equation is merely a constraint and it doesn't prescribe an actual order of evaluation. In Ohm's law's case, there is nothing in the physics stating that u determines i or that i determines u. It may be the case in a particular use of the resistor that the causality holds one way or another, but it is not true in general. For that reason, a description of a resistor that is *independent of the context in which it is used* cannot be written in a causal modelling language.

Figure 5.2: A simple electrical circuit with a diode



The final limitation, arising specifically in hybrid models, is that causality must be fixed. Consider the circuit in figure 5.2 featuring an ideal diode. An ideal diode behaves like a wire with no resistance when the current through it is positive and like an open switch when the voltage through it is negative. This translates to these equations:

$$\begin{cases} u_{\mathrm{D}} = 0 & \text{when } i_{\mathrm{D}} > 0 \\ i_{\mathrm{D}} = 0 & \text{when } u_{\mathrm{D}} < 0 \end{cases}$$

Writing a model for the diode in a language like FRP is not possible, since the model only determines one of u_{D} or i_{D} at any given instant. However, more annoyingly, whether the diode determines u_{D} or i_{D} influences how other equations in the system must be used. Indeed, a description as directed equation of circuit 5.2 goes as follow:

$$u = u_0 \sin(2\pi f t + \varphi) \tag{5.3a}$$

$$u_\mathrm{D} = 0 \tag{5.3b}$$

$$u_r = u - u_\mathrm{D} - u_c \tag{5.3c}$$

$$i_r = \frac{1}{r} u_r \tag{5.3d}$$

$$i_c = i_r \tag{5.3e}$$

$$i_\mathrm{D} = i_r \tag{5.3f}$$

$$i = i_\mathrm{D} \tag{5.3g}$$

$$u_c' = \frac{1}{c} i_c \tag{5.3h}$$

On the other hand, when the diode is closed (the current through it is zero), it is:

$$u = u_0 \sin(2\pi f t + \varphi) \tag{5.4a}$$

$$i_\mathrm{D} = 0 \tag{5.4b}$$

$$i_r = i_\mathrm{D} \tag{5.4c}$$

$$i_c = i_r \tag{5.4d}$$

$$i = i_\mathrm{D} \tag{5.4e}$$

$$u_r = r i_r \tag{5.4f}$$

$$u_c' = \frac{1}{c} i_c \tag{5.4g}$$

$$u_\mathrm{D} = u - u_r - u_c \tag{5.4h}$$

The equation derived from Ohm's law for the resistor is used to compute the current in one mode (equation 5.3d) and the voltage in the other (equation 5.4f). The same is true for equations 5.3g and 5.4c; and equations 5.3c and 5.4h.

5.1.2 — *Non-linearity*

So far, the only system of equations that have been considered were simple linear systems which are easy to transform to directed equation systems by symbolic manipulation. But this is sometimes not possible in the presence of non-linear equations. In the case of electrical circuit, relations can be found when modelling some non-linear electrical components, such as non-linear resistors [146] in which the resistance depends on the intensity passing through it [a]. Depending on how the resistance depends on the intensity, extracting an explicit expression for i_r may not be obvious.

Furthermore, while the solution to a linear equation is unique, there are potentially many solutions to a non-linear problem or none at all. Take the equation stating that two variables x and y describe a circle of radius ℓ in 2-dimensions: $x^2 + y^2 = \ell^2$. Using this equation

146. Trajković, *The Electrical Engineering Handbook*. 2005

[a] The resistance of an element generally depends on the temperature at which it is at. Since the resistor dissipates energy into heat, the resistor's temperature can change as the circuit runs, causing changes in its resistance. This is the case for light bulbs or electric heaters for instance.

to compute x in terms of y gives two solutions: either $x = \sqrt{\ell^2 - y^2}$ or $x = -\sqrt{\ell^2 - y^2}$. In the presence of these equations, insisting on using directed equations can therefore be problematic.

5.1.3 — *Algebraic loops*

Consider the two equations below:

$$\begin{aligned} x &= 2y \\ y &= x + 1 \end{aligned}$$

These equations cannot be rearranged in a way that a value for either x or y is produced first. However, this system has an obvious solution. Replacing x by $2y$ in the second equation gives $y = 2y + 1$, for which $y = -1$ is an obvious solution, giving $x = -2$ as solution for the first equation.

These equations form an algebraic loop: a set of equations which cannot be made causal. Algebraic loops are not allowed in the causal paradigm: in FRP, they correspond to unbounded loops that were discussed briefly in §2.3.2. Sometimes, these loops indeed have no solution, like the following:

$$\begin{aligned} x &= y + 1 \\ y &= x + 1 \end{aligned}$$

which, when replacing x in the second equation gives $y = y + 2$. But in practice, they tend to appear in the description of many physical systems. To express such systems in a causal modelling language requires manually modifying the system so that it is without loops, like what was done above by replacing x by $2y$ in the second equation. While it was easy in this case, doing so for large scale systems can be difficult, simply due to the number of variables and equations that can make up the loop. In the presence of non-linearity, it may not be possible to find an explicit expression for a variable from the equation system.

5.2 — Differential Algebraic Equation

A system of undirected differential equation is named a Differential Algebraic Equations (DAE). As the name indicates, it is a mix of differential equations and algebraic constraints. In general, a DAE takes the following form:

$$F(x', x, y, t) = 0$$

In this context, x represents the set of time-varying functions solved by integrating their derivative x', while y refers to the set of variables only present non-differentiated. The first set are called the *state variables* while the second set are the *algebraic variables*. Looking back on equation system 5.1, i_ℓ and u_c are the set of state variables while all the other variables are algebraic. Like for an ODE, the function F which characterises the DAE is called its residual function.

Simulating a DAE is generally much harder than an ODE, due to the extra algebraic constraints. In general, a solver for a DAE can be built by first numerically computing the value of the algebraic variables and of the derivatives of the state variables from the DAE. Then, a numerical integration method, like the ones used for ODE, is used to compute the values of the state variables from the computed values of the derivatives. A more detailed discussion on the simulation of continuous systems described by ODE or DAE can be found in [37]. A historical perspective on simulation code for DAE systems is given in [121].

37. Cellier et al., *Continuous System Simulation*. 2006

121. Petzold, 'A Description of DASSL: A Differential/Algebraic System Solver'. 1982

5.2.1 — *Causalisation*

The most conceptually straightforward solution is causalisation, which consists of converting the equation system to an ODE by symbolic manipulation like was demonstrated earlier. This process can be mechanised [142] in order to free the modeller from having to perform this transformation.

142. Tarjan, 'Depth-First Search and Linear Graph Algorithms'. 1972

Causalisation is quite an attractive process for solving DAE. Its main advantages are that it allows to reuse all the techniques used for solving ODE while only paying a price at compile-time for causalising the system. In contrast to implicit methods, which will be presented in the next section, it allows for using simpler, more efficient, integration methods.

However, it has two drawbacks. The first has to do with its main advantage: the price at compile time to pay is potentially large for large equation systems. It also prevents (or at least makes more complicated, see [155]) code generation for partial DAE, as they could be causalised in many ways [a].

155. Zimmer, 'Module-Preserving Compilation of Modelica Models'. 2009

The second drawback, as was mentioned before is the issue with non-linearity and algebraic loops. Problems with non-linearity can be mitigated by using differentiation. Indeed, a non-linear equation of the form:

$$f(x_1, x_2, \dots, x_n) = 0$$

can be turned into an ODE in any of the variable x_1 to x_n by differentiation of both sides of the equation, yielding a linear equation in the derivatives x_1' to x_n':

$$x_1' \frac{\partial f}{\partial x_1} + \cdots + x_n' \frac{\partial f}{\partial x_n} = 0$$

Algebraic loops can be removed by using tearing algorithms [37, §7.4]. Tearing is the process of choosing one variable in the algebraic loop, and replacing that variable by its expression as given by an equation in the loop. It is the same process that was used to solve the simple

37. Cellier et al., *Continuous System Simulation*. 2006

[a] In general, there are as many possible causalisations as there are variables appearing in the equation.

loops of §5.1.3: tearing algorithms simply allow for its automation in a tool.

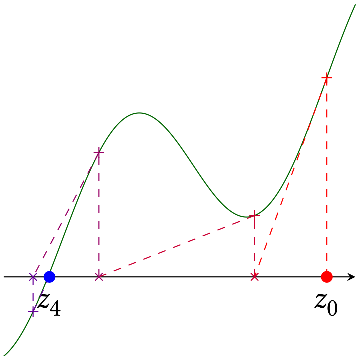
5.2.2 — *Using algebraic solvers*

The alternative to causalisation is to use a dedicated DAE solver. To simplify the discussion, a DAE solver first operates by solving the algebraic system with unknowns x' and y formed by the DAE:

$$F(x', x, y, t) = 0$$

and then using a numerical integration method to integrate x'.

Figure 5.3: Illustration of the Newton-Raphson method in the one dimensional case



To solve this algebraic system, the Newton-Raphson method [127, 150] is the most commonly used method. Its principle is explained here.

Given a problem of the form $G(z) = 0$ and an initial guess for a root of G, z_0, it is possible to compute an approximate value for a solution to the problem by approximating G as a linear function around z_0. If G were truly linear then its graph would be a straight line[a] and therefore its root should be at the point were that straight-line intersects the horizontal axis. This point can be computed as the slope of that line is given by the derivative of G at z_0. If G is truly linear, then that process terminates in one step; if it is not, it repeats until the current guess z_n is such that $\|G(z_n)\| < \epsilon$, where ϵ is a predefined tolerance.

In the one dimensional case (illustrated in figure 5.3), the next guess x_{n+1} is computed from x_n using the following formula:

$$x_{n+1} = x_n - \frac{G(x_n)}{G'(x_n)}$$

In higher-dimensions, when $G : \mathbb{R}^n \to \mathbb{R}^n$, the notion of derivative is generalised by the Jacobian matrix [82]:

$$x_{n+1} = x_n - \mathrm{Jac}_G(x_n)^{-1} G(x_n)$$

where the Jacobian matrix is the square matrix of size n containing all the partial derivatives of G:

$$\mathrm{Jac}_G = \left(\frac{\partial g_i}{\partial z_j}\right)_{i,j\in[1,n]}$$

The Newton-Raphson method is not infallible. It is important that the initial guess be close to the true root, which is not always easy to guarantee [35]. When a function has multiple roots, it can happen that the method computes an approximation for a root which is not the closest one to the guess. This tends to happen when the derivative of G is small, giving an intersection point with the horizontal axis very far away. The method will fail if the derivative of G is zero at the point of the guess, which graphically translates to a horizontal line that never

127. Raphson, ‘Analysis Æequationum Universalis’. 1697 — 150. Wallis, ‘A Treatise of Algebra, Both Historical and Practical’. 1685

[a] Or a plane or hyper plane in higher dimensions.

82. Jacobi, ‘De formatione et proprietatibus Determinatium’. 1841

35. Casella et al., ‘On the Choice of Initial Guesses for the Newton-Raphson Algorithm’. 2021

crosses the horizontal axis. In higher-dimensions, this occurs when the Jacobian matrix is singular, meaning it is not invertible.

DAE solvers often combine the solving of the algebraic system and the integration step into one step. Indeed, suppose that integration of x is performed using a forward Euler scheme. Then the value computed for x' by solving the algebraic system will be used to compute an approximation for x like so:

$$x(t+\delta t) = x(t) + \delta t x'(t)$$

This is equivalent to stating that:

$$x'(t) = \frac{x(t+\delta t) - x(t)}{\delta t}$$

and that therefore, it is enough to solve the following algebraic system:

$$\mathrm{F}\left(\frac{x(t+\delta t) - x(t)}{\delta t}, x(t), y(t), t\right) = 0$$

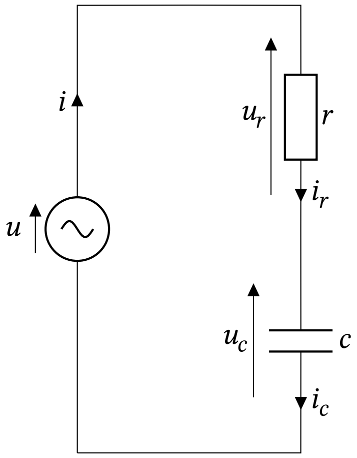
The technique generalises to other integration methods.

5.3 — Higher-index systems

5.3.1 — *Definition*

Whether solving a DAE relies on causalisation (with a mix of ODE solver and algebraic solvers) or uses a DAE solver directly, both schemes rely on the system not being structurally singular. That is, it must be possible to determine, from the equation system, the values of the highest-order derivative of all the variables from the algebraic system. For the state variable, this is their first-order derivative; for the algebraic variable, it is the variable itself (its zeroth-order derivative).

Figure 5.4: A simple RC-circuit



It turns out, however, that many systems do not readily exhibit this property. Let's consider the simple RC-circuit in figure 5.4. The equations for this circuit were given in §2.1. In that section, the voltage source was assumed to be known and the voltages across other components was derived from it. Suppose now that a modeller is interested in a different problem: they want to know what voltage source should be used in order for the voltage across the capacitor to conform to a known function $f(t)$. This translates to the following equation system:

$$u_c = f(t) \tag{5.5a}$$

$$cu_c' = i_c \tag{5.5b}$$

$$u_r = ri_r \tag{5.5c}$$

$$u = u_r + u_c \tag{5.5d}$$

$$i = i_r \tag{5.5e}$$

$$i = i_c \tag{5.5f}$$

There are 6 equations for 6 unknowns: u_c', i_c, u_r, i_r, u and i, but there is no 1-to-1 mapping between these variables and the equations, since no unknown appears in equation 5.5a.

Equation 5.5a is a *constraint* equation: an equation that can only be assigned to solve for a lower-order derivative of an unknown, in this case u_c instead of u_c'. In general, a constraint equation cannot be used directly to compute a solution, rather it restricts the set of solutions of the DAE. For system 5.5, one can recover a solvable system using some simple symbolic manipulation. Since equation 5.5a provides an explicit expression for u_c, one can simply remove it and replace u_c by $f(t)$ in equation 5.5d.

In the presence of more complicated equations (e.g. non-linear equations), a general approach to this problem is to eliminate constraint equations by means of differentiation. Indeed, if $u_c = f(t)$ holds at all time, then $u_c' = f'(t)$ holds too. Replacing equation 5.5a by this new, latent, equation yields a DAE which is not structurally singular. If the initial conditions of the variables in the system satisfy both the latent equations and the constraint equations, then the solution for the new DAE is the same as the one for the original DAE [97].

The process of discovering the set of equations that must be differentiated for a DAE to be non-singular is referred to as index-reduction. The index refers to the number of times that parts of or the whole of the DAE must be differentiated to transform it into an ODE. A DAE that is not structurally singular and can be simulated with a DAE solver is index-1. It is sometime referred to by the term implicit ODE since it can be put under the form:

$$\begin{aligned} x' &= f(x, y, t) \\ 0 &= g(x, y, t) \end{aligned}$$

where x is the set of state variables and y the set of algebraic variables. Differentiating the algebraic constraints once indeed yields an index-0 ODE, although this step is not necessary for simulation.

Since equation system 5.5 needed one of its equations to be differentiated, it is index-2.

5.3.2 — *Index-reduction algorithms*

Index-reduction can be performed solely from structural information about the DAE [29, 116, 126]. That is, it merely requires knowing which variables appear in which equations but doesn't require more precise information about each equation.

One such method is Pryce's signature method (or Σ-method). It is an alternative to the more commonly used Pantelides algorithm [116]. It is presented here since it is somewhat easier to understand and is

[97] Mattsson et al., 'Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives'. 1993

[29] Caillaud, 'Implicit Structural Analysis of Multimode DAE Systems'. 2020 — 116. Pantelides, 'The Consistent Initialization of Differential-Algebraic Systems'. 1988 — 126. Pryce, 'A Simple Structural Analysis Method for DAEs'. 2001

[116] Pantelides, 'The Consistent Initialization of Differential-Algebraic Systems'. 1988

[126] Pryce, 'A Simple Structural Analysis Method for DAEs'. 2001

used in the implementation work described in later chapters. For all the details and proofs of correctness, the original paper [126] is easy to read.

The signature method relies on the signature matrix of the DAE. The signature matrix Σ is a square matrix such that the coefficient σ_{ij} is a value of $\mathbb{N} \cup \{-\infty\}$ and gives information on how the i-th variable of the DAE appears in its j-th equation. Specifically, if the variable doesn't appear in this equation, $\sigma_{ij} = -\infty$; if it appears differentiated n times (at most), then $\sigma_{ij} = n$. The signature matrix for equation system 5.5 is given by ($\cdot$ is used to represent $-\infty$):

$$\Sigma = \begin{array}{cccccc|l} u_c & i_c & u_r & i_r & u & i & \\ \hline 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 5.5\text{a} \\ 1 & 0 & \cdot & \cdot & \cdot & \cdot & 5.5\text{b} \\ \cdot & \cdot & 0 & 0 & \cdot & \cdot & 5.5\text{c} \\ 0 & \cdot & 0 & \cdot & 0 & \cdot & 5.5\text{d} \\ \cdot & \cdot & \cdot & 0 & \cdot & 0 & 5.5\text{e} \\ \cdot & 0 & \cdot & \cdot & \cdot & 0 & 5.5\text{f} \end{array}$$

The Σ-method then specifies index-reduction as a linear programming problem known as the assignment problem [15]. The first task is to find an assignment (if it exists) of variables to equations, such that a variable is uniquely assigned to an equation in which it appears (and vice-versa, an equation cannot be assigned to multiple variables). To choose which variable is assigned to which equation, equations where the variable appears differentiated more times are preferred. This problem can be understood better with a worker/task analogy, where variables are tasks that need to be performed and workers are equations able to perform these tasks. The signature matrix then quantifies the proficiency of a worker (equation) to perform a certain task (solve for the variable). When a worker is unable to perform a task, their proficiency is $-\infty$. The goal is to maximise the total productivity. This can be formulated as the following linear programming problem, where one searches for the $\xi_{ij} \in \mathbb{N}$ such that the quantity:

[15] Bertsekas, *Linear Network Optimization*. 1991

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \xi_{ij} \sigma_{ij}$$

is maximized, with the additional constraints that:

$$\forall i \in [1, n], \sum_{j=1}^{n} \xi_{ij} = 1$$

$$\forall j \in [1, n], \sum_{i=1}^{n} \xi_{ij} = 1$$

Loosely speaking, the quantity being maximised can be viewed as the total productivity of a given assignment of workers to tasks. The constraints state that exactly one worker must be assigned to a task and exactly one task must be assigned to a worker.

From the assignment given by ξ_{ij}, finding the number of differentiations needed to reduce the system consists of finding the minimal difference between the number of times the variable assigned to an equation appears differentiated in the equation originally and the highest-order derivative of that equation in the final *reduced* DAE. Since differentiating an equation can make other variable appear differentiated, it is important to not simply consider the highest-order derivative of a variable in the original, higher-index, DAE.

Fortunately, the solution to that problem is simple to compute from the assignment. It is formulated as the dual problem of the linear programming problem above. Given two n-tuples C and D, where C collects the number of differentiation for each equation and D collects the order of the highest-order derivative for each variable, the problem consists of minimising the following quantity:

$$\sum_{j=1}^{n} d_j - \sum_{i=1}^{n} c_i$$

with the added constraints that:

$$\begin{aligned} \forall i, j \in [1, n], d_j - c_i &\geq \sigma_{ij} \\ \forall i \in [1, n], c_i &\geq 0 \end{aligned}$$

When formulated as such, there are many optimal solutions to that problem. Indeed, if C and D are solutions, then C + 1 and D + 1 are also solutions. However, there exists a minimal (or canonical) set of offsets that can easily be found by a simple fixed-point procedure.

5.3.3 — *State selection*

The DAE system obtained by simply replacing constraint equations is sometime referred to as the Underlying Ordinary Differential Equation (UODE). Simulating the UODE directly theoretically provides a valid solution for the original DAE, provided the initial conditions used for simulating the UODE also satisfy the DAE. Practically, however, the computed solution of the UODE tends to 'drift-off' from the original constraints. Indeed, the constraint equations are completely removed from the UODE and only hold implicitly. However, due to numerical errors, the computed solution may not truly satisfy these constraints.

The solution proposed in [97] is to not only use the differentiated equations but also the constraint equations to simulate the system [a]. Consider the following index-2 DAE:

$$u_c = f(t) \tag{5.6a}$$

$$u_{r_1} = r_1 i_{r_1} \tag{5.6b}$$

[97] Mattsson et al., 'Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives'. 1993

[a] If an equation is differentiated multiple times, both the original equation and all the differentiated versions of that equation are to be used.

$$cu_c' = i_c \tag{5.6c}$$
$$u_{r_2} = r_2 i_{r_2} \tag{5.6d}$$
$$\ell i_\ell' = u_\ell \tag{5.6e}$$
$$i = i_1 + i_2 \tag{5.6f}$$
$$i_1 = i_{r_1} + i_c \tag{5.6g}$$
$$i_{r_1} = i_c \tag{5.6h}$$
$$i_2 = i_{r_2} + i_\ell \tag{5.6i}$$
$$i_{r_2} = i_\ell \tag{5.6j}$$
$$u = u_{r_1} + u_c \tag{5.6k}$$
$$u_{r_1} + u_c = u_{r_2} + u_\ell \tag{5.6l}$$

It is derived from figure 5.1 and equation system 5.1. However equation 5.1a has been replaced by a constraint equation over u_c. Differentiating equation 5.6a yields an index-1 DAE but simply adding it to system 5.6 produces 13 equations and only 12 unknowns:

$$u_c = f(t) \tag{5.7a}$$
$$u_c' = f'(t) \tag{5.7a$'$}$$
$$u_{r_1} = r_1 i_{r_1} \tag{5.7b}$$
$$cu_c' = i_c \tag{5.7c}$$
$$u_{r_2} = r_2 i_{r_2} \tag{5.7d}$$
$$\ell i_\ell' = u_\ell \tag{5.7e}$$
$$i = i_1 + i_2 \tag{5.7f}$$
$$i_1 = i_{r_1} + i_c \tag{5.7g}$$
$$i_{r_1} = i_c \tag{5.7h}$$
$$i_2 = i_{r_2} + i_\ell \tag{5.7i}$$
$$i_{r_2} = i_\ell \tag{5.7j}$$
$$u = u_{r_1} + u_c \tag{5.7k}$$
$$u_{r_1} + u_c = u_{r_2} + u_\ell \tag{5.7l}$$

Notice however that there is some redundancy in this overdetermined system. between equation 5.7a and 5.7a′. For instance, the fact that u_c' is the derivative of u_c is implicitly present in the fact that both $u_c = f(t)$ and $u_c' = f'(t)$ hold. In other words, if one were to replace u_c' by an algebraic variable $\dot{u}_c$, this information would not be lost, only holding implicitly[a]. Performing this transformation yields the following index-1 DAE, with 1 state variable i_ℓ and 12 algebraic variables, including u_c and $\dot{u}_c$:

[a] In the same way that the constraint equations held only implicitly in the UODE.

$$u_c = f(t) \tag{5.8a}$$
$$\dot{u}_c = f'(t) \tag{5.8a$'$}$$

$$u_{r_1} = r_1 i_{r_1} \tag{5.8b}$$

$$c\dot{u}_c = i_c \tag{5.8c}$$

$$u_{r_2} = r_2 i_{r_2} \tag{5.8d}$$

$$\ell i'_\ell = u_\ell \tag{5.8e}$$

$$i = i_1 + i_2 \tag{5.8f}$$

$$i_1 = i_{r_1} + i_c \tag{5.8g}$$

$$i_{r_1} = i_c \tag{5.8h}$$

$$i_2 = i_{r_2} + i_\ell \tag{5.8i}$$

$$i_{r_2} = i_\ell \tag{5.8j}$$

$$u = u_{r_1} + u_c \tag{5.8k}$$

$$u_{r_1} + u_c = u_{r_2} + u_\ell \tag{5.8l}$$

$\dot{u}_c$ is a *dummy-derivative*: a placeholder algebraic variable for what is really the derivative of another variable. The dummy-derivative algorithm [97] gives a systematic method to select a valid set of state variables and dummy-derivatives. For instance in the above example, choosing u_c as a state i'_ℓ as a dummy-derivative would not have helped with solving the system.

It is sometimes necessary to change the set of dummy-derivatives during the simulation. In the presence of non-linear equations, the DAE can become singular in the neighbourhood of particular points, leading to solvability issues or inconsistencies such as discontinuities. The subtleties associated with selecting a 'good' set of dummy-derivatives and when to change it is outside the scope of this work. The interested reader can refer to [99], which provides a very thorough overview of the dummy-derivative algorithm and ways to efficiently implement dynamic state selection.

[97] Mattsson et al., 'Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives'. 1993

[99] McKenzie et al., 'Structural Analysis Based Dummy Derivative Selection for Differential Algebraic Equations'. 2017

5.3.4 — *Concluding remarks on index-reduction*

This section introduced the concept of higher-index DAE and the difficulties associated with their simulation. Higher-index systems arise when a DAE contains constraint equations, that only constrain the solution but cannot be used directly for simulating the equation. Structural algorithms can be used to recover an index-1 DAE from a higher-index DAE by differentiating these constraint equations, potentially several times. The resulting DAE can then be simulated: either by directly simulating the UODE, obtained by replacing the constraint equations with their differentiated versions; or by keeping the constraint equations but removing some state variables from the DAE to avoid an overdetermined system.

Let's conclude this summary by noting that index-reduction algorithms are not modular in the sense that, in general, it is not possible to know, from a partial DAE, how many times each equation might appear differentiated when that partial DAE is used as part of a larger one. In cases where parts of the DAE change during simulation, the analysis must be performed after each such change on the whole DAE, not simply on the part that was modified. This is not surprising since index-reduction can be performed by finding a 1-to-1 assignment between equations and variables [a], but §5.1.1 showed how such assignments typically did not remain valid after a structural change in a DAE and generally completely depend on the context in which a particular set of equation is used inside a larger equation system. Note that recent work [29] has been proposed that addresses this particular issue for fully-assembled hybrid DAE with finite number of modes; however it is not usable for partial DAE nor DAE with unbounded number of modes, in the style of FRP [b].

5.4 — Initialisation

Initialising an ODE only requires the specification of the initial values of the state variables. The initial values of all other variables and the derivatives of the state variables can trivially be determined from the ODE itself. However, initialising a DAE is harder, since the initial conditions for the state variables and all other variables must satisfy the DAE, thus requiring solving an algebraic system at initialisation time.

Higher-index systems pose additional difficulties, namely that some state variables may not need an initialising equation at all. Consider system 5.6 again. Even if u_c appears differentiated, its initial value is constrained and providing an explicit equation for it would result in an over-determined system. The number of additional equations to correctly initialise a DAE corresponds to its number of *degrees of freedom* [126]. In the case of system 5.6, the system has 0 degrees of freedom since it can be initialised without the need for any additional equations. By contrast, system 5.1 had 2 degrees of freedom since it requires an initial value for both i_ℓ and u_c.

The number of degrees of freedom can be computed from the output of index-reduction as the following quantity:

$$\sum_{j=1}^{n} d_j - \sum_{i=1}^{n} c_i$$

This makes it particularly easy to verify that there are enough additional equations to initialise the system, although it can sometimes be difficult for the modeller to clearly identify which are needed in complex equation systems. This problem can be somewhat alleviated by allowing the user to state an over-determined system, identifying and eliminating redundant information and then verifying the consistency of the computed solution with regards to these eliminated constraints [114].

[a] See §5.3.2

[29] Caillaud et al., 'Implicit Structural Analysis of Multimode DAE Systems'. 2020

[b] See §2.3.4

[126] Pryce, 'A Simple Structural Analysis Method for DAEs'. 2001

[114] Ochel et al., 'Symbolic Initialization of Over-Determined Higher-Index Models'. 2014

5.5 — Introduction to Functional Hybrid Modelling

Functional Hybrid Modelling (FHM) [112] has been proposed as an expressive approach for designing a non-causal modelling language. FHM is inspired by FRP, in particular by Yampa. Like FRP, FHM is often realised as an embedding in a host functional language. Previous work [65] used Haskell as the host language.

This section provides an introduction to FHM. It uses a new implementation of the concept of FHM called Hydra. While the implementation reuses the name of previous implementations [63], it is entirely new[a]. In particular, the functional host language, in which the modelling language FHM is embedded, are implemented by the same compiler. This section is purely meant as an introduction. A precise definition of the language will be given in the next chapter. The goal of the work will be to present the modelling of circuit 5.1 as a *signal relation*, as introduced below, and show how treating relations as first-class enables great flexibility and modularity in the modelling process.

5.5.1 — *Signal relations*

§2.3.1 introduced the concept of signals as time-varying values and signal functions as functions between signals. This proves to be useful abstraction in the context of FRP and causal modelling. To capture the semantics of differential equations as constraints, or predicates, over signals, the notion of *signal relation* **SR** α is defined as a predicate on signals of type α, conceptually:

$$\mathbf{SR}\ \alpha \approx \mathbf{Signal}\ \alpha \rightarrow \text{Predicate}$$

Where the idea of FRP is to program with signal functions as first-class values, Functional Hybrid Modelling (FHM) has first-class signal relations.

5.5.2 — *Individual equations as signal relations*

The first task for modelling circuit 5.1 is to model each individual component. In particular, one must provide the equations governing the behaviour of the resistor, capacitor, inductance and voltage source appearing in the circuit. Each component of the circuit can be modelled as a signal relation between the voltage across it and the current that flows through it. For instance, the behaviour of the resistor in Hydra is captured by the following:

```
let resistor r =
  sigrel u, i {
    u = r * i
  }
```

112. Nilsson et al., ‘Functional Hybrid Modeling’. 2003

65. Giorgidze et al., ‘Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems’. 2009

63. Giorgidze, ‘First Class Models’. 2012

[a] The source code for the compiler can be found at: `https://gitlab.com/chupin/hydra-v2`.

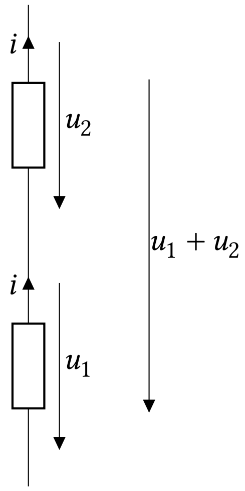
`resistor` is a function parametrised by a real number `r`. Its type is `real → SR (real, real)` The keyword `sigrel` introduces a signal relations between the signals `u` and `i`, called the *interface variables* of the relation. Similarly to the *proc*-notation, these signals only exist inside the body of the signal relation. Signal relations modelling a capacitor and inductance can be written like so:

```
let capacitor c =
  sigrel u, i {
    c * der u = i
  }

let inductance l =
  sigrel u, i {
    l * der i = u
  }
```

Notice the use of the `der` keyword that allows to refer to the derivative of a signal. Note that the argument to `der` needs not be an identifier but can be any arbitrary signal of real type.

Figure 5.5: Two components connected in series



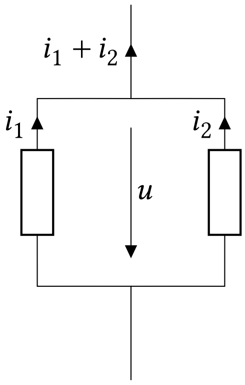
5.5.3 — *Higher-order signal relations*

The voltage through two components connected either in series or in parallel is given by Kirchhoff's laws. Previously, these laws were used in ad-hoc ways when modelling in a causal setting. The non-causal setting allows to state them once and then reuse them at will.

Connecting two components 1 and 2 in series (illustrated in figure 5.5) results in a component where the voltage across and current through follows:

$$
\begin{aligned}
u &= u_1 + u_2 \\
i &= i_1 \\
i_1 &= i_2
\end{aligned}
$$

Figure 5.6: Two components connected in parallel



while connecting two components in parallel (illustrated in figure 5.6) results in the following:

$$
\begin{aligned}
u &= u_1 \\
u_1 &= u_2 \\
i &= i_1 + i_2
\end{aligned}
$$

Since these laws are independent of the components being connected together, it is possible to implement them in Hydra by parametrising over the signal relations describing each component, like so:

```
let serial c1 c2 =
  sigrel u, i {
    let u1, i1, u2, i2 {
      c1 <> u1, i1;
      c2 <> u2, i2;
      u  =  u1 + u2;
      i  =  i1;
      i1 =  i2
    }
  }
```

serial has type:

```
SR (real, real) → SR (real, real) → SR (real, real)
```

c1 and c2 are the signal relations describing the components. The <> symbol is a signal relation application: the equation c1 <> u1,   i1 states that u1 and i1 are related through signal relation c1. The signals u1, i1, u2 and i2 are local signals declared with the **let** keyword.

The parallel signal relation, modelling two components connected in parallel can be implemented in the same way:

```
let parallel c1 c2 =
  sigrel u, i {
    let u1, i1, u2, i2 {
      c1 <> u1, i1;
      c2 <> u2, i2;
      u = u1;
      u1 = u2;
      i  = i1 + i2;
    }
  }
```

Using these two relations, it is possible to construct, by function application, a signal relation which represents the right part of circuit 5.1, containing the resistors, capacitor and inductance:

```
let rlc =
  parallel (serial (resistor r1)
                   (initialised_capacitor uc0 c))
           (serial (resistor r2)
                   (initialised_inductance il0 l))
```

In the above, initialised_capacitor and initialised_inductance are used instead of capacitor and inductance. These relations are defined as:

```
let initialised_capacitor uc0 c = sigrel u, i {
  capacitor c <> u, i;
  init u = uc0;
}
```

```
let initialised_capacitor il0 l = sigrel u, i {
  inductance l <> u, i;
  init i = il0
}
```

An `init` equation is an equation that only holds at the very first instant the signal relation starts to hold. Such equations are necessary to set the initial values of state variables in the system. Hydra supports arbitrary initialisation equations, although simple equations like the ones above are given special treatment by the implementation and are not solved by using a numerical solver but simply by translating them to assignments.

To complete the circuit, a last signal relation is needed to model the closed circuit formed by the voltage source and the rest of the components. Again, this can be implemented as a higher-order signal relation:

```
let close c1 c2 = sigrel () {
  let u1, i1, u2, i2 {
    c1 <> u1, i1;
    c2 <> u2, i2;
    u1 = u2;
    i1 = i2;
  }
}
```

`close` has no interface signal, since it is intended to represent a complete and fully-determined circuit. The final circuit can now be assembled. This final signal relation can be parametrised with a voltage source, thus allowing its simulation with many different sources:

```
let circuit source = close source rlc
```

Results of the simulation of `circuit` with a constant voltage source and a sine-wave voltage source are shown in figure 5.7 and 5.8.

5.5.4 — *Hybrid signal relations*

Hydra possesses a switch construct, similar to the one introduced for FRP. In this version of Hydra, the switch is however more limited that what exists in Yampa or in previous versions of Hydra [63]. Suppose one wanted to simulate the behaviour of `circuit` with more interesting voltage sources. For instance, a triangle wave function. A triangle wave function could be implemented as the composition of the asin and sin function. Indeed, since asin is the inverse of sin on $[-1, 1]$, a triangle wave function of amplitude A and frequency f is given by:

$$\text{Triangle}(t) = \frac{2\text{A}}{\pi} \arcsin\left(\sin\left(2\pi f t\right)\right)$$

[63] Giorgidze, ‘First Class Models’. 2012

Figure 5.7: Simulation results for `circuit` under a constant voltage source

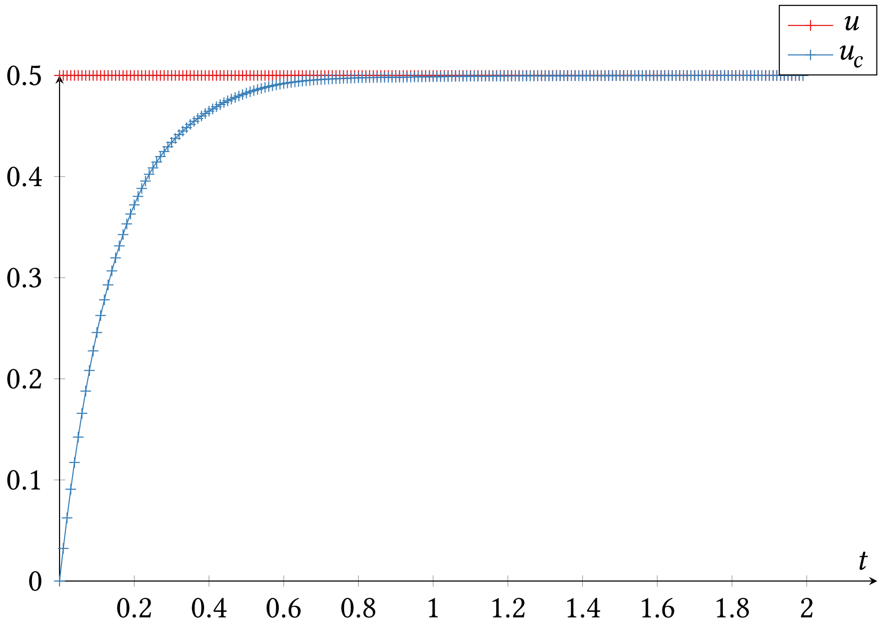


Figure 5.8: Simulation results for `circuit` under a sine-wave voltage source

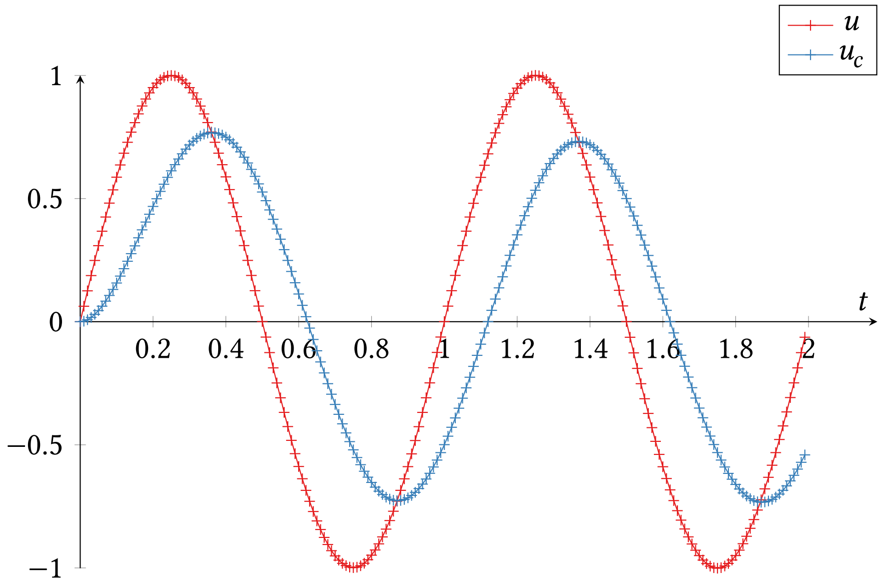
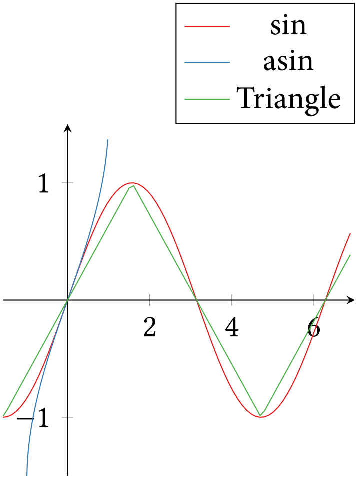


Figure 5.9: Plot of the asin, sin and Triangle functions (with $A = 1$ and $f = \frac{1}{2\pi}$)



However, although it is correct, this definition has several problems:

— if evaluated with floating-point, the precision of the result can decrease as time passes,

— the resulting function has a discontinuous derivative, which is problematic if the derivative is needed, e.g. in the context of a higher-index system,

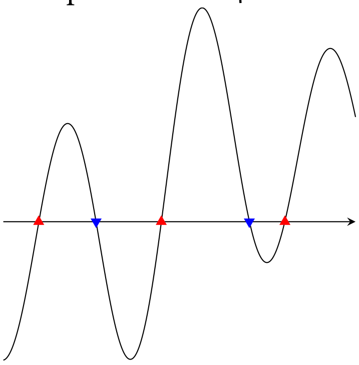
— regardless of the previous points, the technique used here doesn't generalise well to other discontinuous or otherwise piecewise-defined functions.

A better solution is to define the triangle function as a piecewise function made of two alternating affine functions with slope $\pm Af^{a}$. This can easily be encoded using a `switch` relation, in the following way:

```
let triangle_wave a f =
  sigrel w {
    init w = 0.0;
    switch init Up
      mode Up -> der w = a * f
        when up(w - a) -> Down
      mode Down -> der w = - a * f
        when down(w - a) -> Up
  }
```

[a] The functions grows by $\pm 2A$ over half a period length $\frac{T}{2}$. Hence the rate of change is given by $\pm\frac{2A}{\frac{T}{2}}$, which simplifies to $\pm Af$.

Figure 5.10: Illustration of the zero-crossing of a signal ▲ represents the time at which events are reported by applying `up` to the given signal, while ▲ represents the events reported by applying `down`. Both are reported with `updown`.



A switch relation consists of several modes whose names start with a capital letter. At any given point in time only one mode is active, meaning that the relations it specifies hold. By default, the first mode is active when the relation is first activated. Optionally, however, the initial mode can be specified with `init`, like it is in the example. This switch relation is made of two modes: **`Up`**, during which the equation:

```
der w = a * f
```

is active and **`Down`**, during which the equation:

```
der w = - a * f
```

is active. The transitions between modes occurs at events specified by **`when`** clauses. A **`when`** clause specifies an event and the mode that the switch will switch into when the event occurs. The events are specified by means of three combinators: `up`, `down` and `updown`. These combinators take a real valued signal as argument and produce an event when that signal crosses zero. The `up` (resp. `down`) combinator produces an event only when the signal crosses zero from a negative to a positive value (resp. from a positive to a negative value), while `updown` produces an event anytime the signal crosses zero. In the case of `triangle_wave`, the transition occurs when the interface variable w reaches a (w - a crosses 0 going up) or -a (w - a crosses 0 going down).

Modes can be given arguments. As both modes in the above model are very similar, this allows merging them into a single one:

```
let triangle_wave a f =
  sigrel w {
    init w = 0.0;
    switch init Triangle(1)
      mode Triangle(c) → der w = c * a * f
        when up(w - a) → Triangle(-1)
        when down(w - a) → Triangle(1)
  }
```

The new **Triangle** uses its argument to encode the direction of the derivative of w: alternating going up and down using the zero-crossing signals.

More interestingly, models such as the diode can also be expressed in a modular fashion, something not possible in a causal language, as explained in §5.1.1. The following is a model for an diode which is, initially, closed:

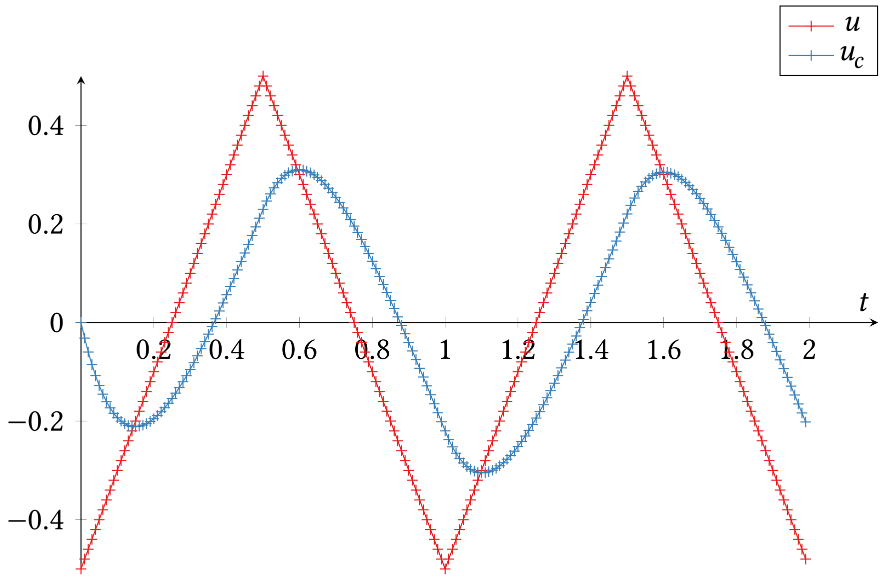
```
let ic_diode =
  sigrel u, i {
    switch init Closed
      mode Closed → u = 0
        when down(i) → Open
      mode Open → i = 0
        when up(u) → Closed
  }
```

Note that there is no way currently to specify programmatically what the first mode of a switch should be. For the diode, this would allow to specify the initial mode of the diode depending on the values of i and u at the initial instant. While it would be extremely useful, this feature is currently left as future work and will be discussed in chapter 9.

The simulation of signal relation circuit with a triangle wave signal source is given in figure 5.11.

In triangle, the signal w is continuous through the mode change. In general, Hydra assumes that any state variable is continuous through a mode change. The definition of a state for that matter is any signal which appears differentiated at least once in the signal relation, meaning that whether or not it has been selected to be an actual state during simulation is not relevant. Note further that this property is non local: an interface variable may not appear differentiated in a particular signal relation, but can appear differentiated when used outside of that particular signal relation. This property may also change depending

Figure 5.11: Simulation results for `circuit` under a triangle-wave voltage source



on the mode a signal relation is in, since a variable may appear differentiated in one mode and not in another.

Considering all state variables to be continuous is sometime problematic. Recall the bouncing ball example from §2.3.4. One would want to implement it like so in Hydra:

```
let falling y0 vy0 = sigrel y, vy {
  der y = vy;
  der vy = -9.81;
  init (y, vy) = (y0, vy0);
}

let bouncing y0 vy0 = sigrel y, vy {
  switch init Bounce(y0, vy0)
    mode Bounce(y0, vy0) →
      falling y0 vy0 <> y, vy;
      when down(y) → Bounce(y, -vy)
}
```

Both `y` and `vy` are states in these relations. However, although `y` can be considered continuous, `dy` explicitly is not: it needs to be reinitialised. Hydra can be made to consider a signal as reinitialised by marking it with the `reinit` keyword:

```
let bouncing y0 vy0 = sigrel y, vy {
  switch init Bounce(y0, vy0)
    mode Bounce(y0, vy0) reinit y, vy →
      falling y0 vy0 <> y, vy;
      when down(y) → Bounce(y, -vy)
}
```

Marking y and vy as reinitialised informs Hydra that it should expect init equations for these two variables, and must assume that all their derivatives are unknown at the time of switching.

5.5.5 — *Modelling by constraints*

Hydra permits simulating models expressed with constraints. For instance, one can model the same electrical circuit, this time forcing the voltage through the capacitor to follow a set function. Such a 'constrained capacitor' model can be implemented like so:

```
let constrained_capacitor constraint c =
  sigrel u, i {
    capacitor c <> u, i;
    constraint <> u;
  }
```

It can then replace the initialised_capacitor relation in the rlc signal relation:

```
let constrained_rlc constraint =
  parallel (serial (resistor r1)
                   (constrained_capacitor constraint c))
           (serial (resistor r2)
                   (initialised_inductance il0 l))
```

Note how in this signal relation there is no initialisation equation for the voltage across the capacitor, since it is expected that the constraint determines its value at every instant (including the initial instant). The full circuit can then assembled with the voltage source left unconstrained using the empty signal relation noop:

```
let noop = sigrel x {}

let constrained_circuit constraint =
  close noop (constrained_circuit constraint)
```

Results for simulating this circuit with different forcing functions are given in figure 5.12 and 5.13.

5.5.6 — *Delimitations*

The goal of this section is to give a clear account of what kinds of model the current implementation of Hydra can effectively simulate. The limitations to the capabilities of Hydra are in part due to limitations of the techniques Hydra implements (such as the way it performs state selection or initialisation) and partly due to the lack of

Figure 5.12: Simulation results for `constrained_circuit` with a sine-wave constraint

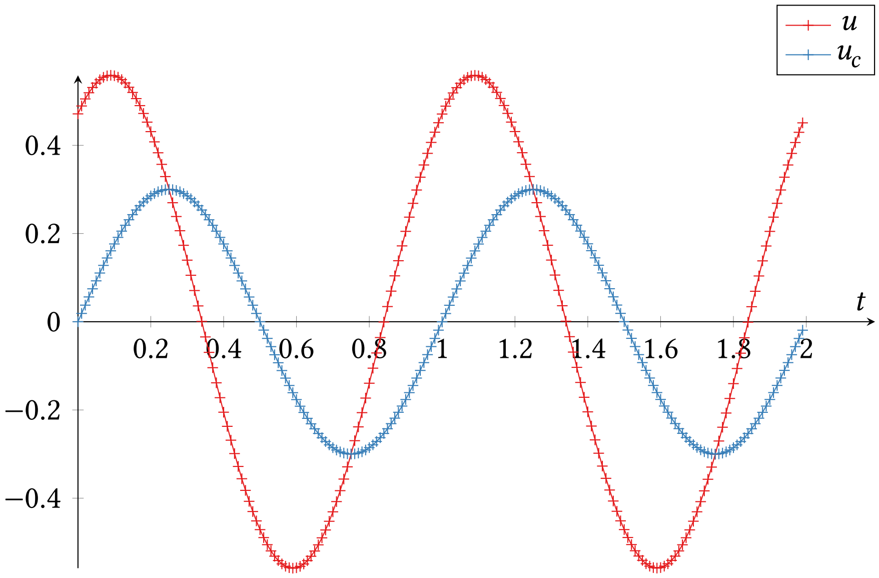
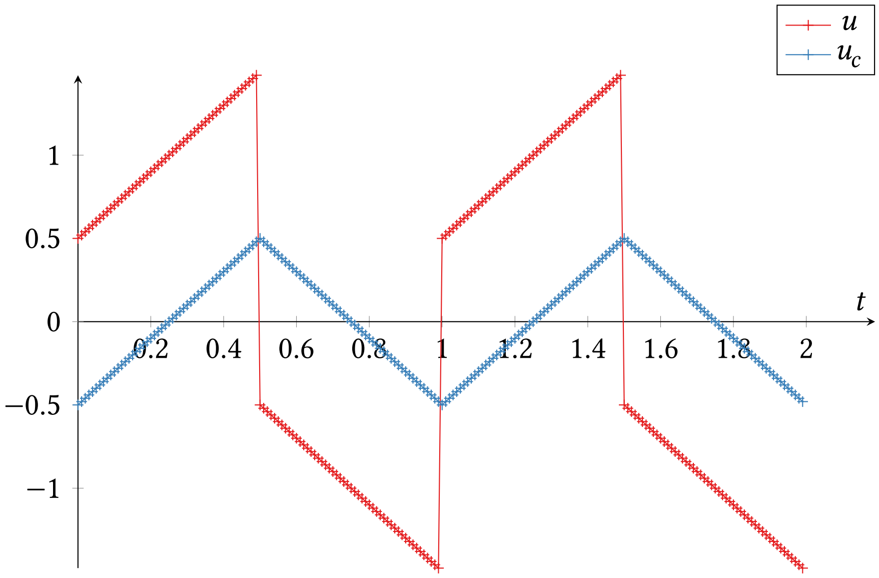


Figure 5.13: Simulation results for `constrained_circuit` with a triangle-wave constraint



some features that Hydra would need to implement to allow the modeller to assist the implementation during the simulation. Note however that these limitations are not inherent to the Hydra or fhm as an approach to implementing modelling languages, rather they stem from the need to prioritise some features over other, given the limited resources available for its development.

Currently, Hydra is perfectly capable of simulating higher-index systems consisting only of linear equations. It is able to handle some hybrid systems with varying index between modes, although this can require substantial extra work from the modeller, for instance in the form of additional reinitialisation equations to avoid underdetermined or overdetermined systems. Hydra has support for non-linear equations although it may have some difficulties initialising and simulating systems with non-linear equations. The difficulty with initialising a system with non-linear equation is, in part, due to the need to find a good initial guess to initialise a system of algebraic equations. While it is not a problem for a linear system of algebraic equations[a], it is for non-linear systems. Finding a guess is often considered domain-specific knowledge and therefore it is the responsibility of the modeller to provide such a guess. Hydra does not currently have a way to let the user specify a guess and instead chooses the same initial guess for every signal, with the exception of signals for which there is a trivial initial equation of the form:

```
init x = c
```

where `c` is some constant; or, in the case of a mode change, signals whose value was known in the previous mode. These limitations may make Hydra fail to initialise a system, particularly ones with non-linear equations, if the true initial value of a signal is not close to `0`, the value it uses by default. To avoid the problem, the easiest method would be to allow for annotations either on variable declaration or at mode change to allow to specify a guess for that variable.

The difficulty with simulating a system with non-linear equations with Hydra is due to its state selection method. As hinted to in §5.3.3, the choice of dummy-derivative is crucial to perform the simulation of a higher-index system. Hydra's implementations however is quite crude: it implements a static dummy-derivative algorithm[b] and selects states based on a 'first comes, first served' basis: if a state variable can be a dummy-derivative in such a way that this forms a valid set of states, then it is selected, but that may not lead to a system that can be effectively simulated. This can be alleviated by implementing better state selections algorithms that take into account the values of signals at mode change to perform state selection and by allowing the modeller to suggest or impose that a variable be used as a state. This can be done by using special purpose annotations, like what is done in Modelica or Dymola [56, 103].

[a] For which one iteration of Newton's method is always sufficient to find a solution, see §5.2.2.

[b] That is, states are selected once at the start of the simulation of a given mode.

56. Elmqvist, *Object-Oriented Modeling of Hybrid Systems.* 1993 — 103. Modelica Association, *Modelica Language Specification.* 2021

6 *Detailed specification of Hydra*

This chapter provides a detailed specification of Hydra at the surface level. It introduces both the functional host language and modelling language embedded within it, with a specific focus on the latter. The implementation is left for the next chapter.

§6.1 presents the abstract syntax of Hydra. §6.2 introduces its type system, which presents similarities with the type system presented in part I for sfrp. §6.3 introduces a small core language, which is a simplified version of the surface language and is easier to use for compilation. It is this core language that will be used later to discuss the implementation. Finally, §6.4 shows an interpretation of Hydra's signal relations in terms of dae and shows a simple simulation method for based on this interpretation. The difficulties with mapping this interpretation to a concrete implementations are then discussed to serve as a base for the contributions of the next chapter.

6.1 — Surface syntax

The surface syntax of the language is given in figure 6.1 in Backus-Naur form. The language is a two-staged language, it consists of a functional part, similar to ML-style language and a language for signal relations. Most features have already been introduced with the previous examples, with the exceptions of patterns and anonymous functions.

The functional host language for Hydra is arguably very primitive, in comparison to mainstream functional languages. This work was not an exercise in the compilation of efficient and expressive functional languages. The choice of using a custom language for this work was made in an attempt to give greater flexibility for experimenting with new ideas, without being limited by the capabilities of the host language. This has been instrumental in the development of the ideas that will be presented in chapter 7.

A Hydra program consists of a series of declarations. A declaration may either be a value declaration, which declares a name associated with an expression; or a type declaration. Currently, Hydra only supports declarations of type aliases or records. In particular, support for sum types is left as future work.

6.2 — Type system

Hydra is statically typed and supports polymorphism. Types are inferred at compile-time automatically, although binders and expres-

Figure 6.1: Hydra abstract syntax

```
‹⊕›            ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’                                        (Binary operators)

‹unop›         ::= ‘sin’ | ‘cos’ | ‘tan’ | ‘sinh’ | ‘cosh’ | ‘tanh’                  (Unary operators)
                 |  ‘asin’ | ‘acos’ | ‘atan’ | ‘asinh’ | ‘acosh’ | ‘atanh’
                 |  ‘sqrt’ | ‘exp’ | ‘log’

‹signal›       ::= ‹expr›                                                            (Constants)
                 |  identifier
                 |  ‘der’ ‹signal›
                 |  ‹signal› ‹⊕› ‹signal›
                 |  ‹unop› ‹signal›
                 |  ‹signal› ‘.’ label                                               (Field selection)
                 |  ‘(’‹signal›‘)’
                 |  ‹signal› ‘,’ ‹signal›

‹initMode›     ::= Identifier ‘(’ ‹expr› ‘)’
‹targetMode›   ::= Identifier ‘(’ ‹signal› ‘)’
‹patternMode›  ::= Identifier ‘(’ ‹pattern› ‘)’

‹equation›     ::= ‹signal› ‘=’ ‹signal›
                 |  ‘init’ ‹signal› ‘=’ ‹signal›
                 |  ‹expr› ‘<>’ ‹signal›                                             (Signal relation application)
                 |  ‘let’ ‹pattern› ‘{’ ‹equation› ‘}’
                 |  ‹equation› ‘;’ ‹equation›
                 |  ‘switch’ ‘init’ ‹initMode› ‹branch*›

‹branch›       ::= ‘mode’ ‹modePattern› ‘->’ ‘reinit’ identifier * ‹equation› ‹condition›*

‹condition›    ::= ‘when’ ‹event› ‘->’ ‹targetMode›

‹event›        ::= ‘up’ ‹signal›
                 |  ‘down’ ‹signal›
                 |  ‘updown’ ‹signal›

‹pattern›      ::= ‘_’                                                               (Wildcard pattern)
                 |  identifier
                 |  ‘(’‹pattern›‘)’
                 |  identifier ‘@’ ‹pattern›                                         (Alias pattern)
                 |  ‹pattern› ‘,’ ‹pattern›                                          (Product pattern)
                 |  ‘{’ label ‘:’ ‹pattern› ‘}’                                      (Record pattern)

‹expr›         ::= constant
                 |  identifier
                 |  ‹expr› ‹expr›                                                    (Function application)
                 |  ‘fun’ ‹pattern› ‘->’ ‹expr›                                      (Anonymous functions)
                 |  ‘sigrel’ ‹pattern› ‘{’ ‹relation› ‘}’
                 |  ‹signal› ‹⊕› ‹expr›
                 |  ‹unop› ‹expr›
                 |  ‹expr› ‘.’ label                                                 (Field selection)
                 |  ‘(’‹expr›‘)’

‹type›         ::= ‘real’                                                            (Real type)
                 |  identifier                                                       (Expression type variable)
                 |  ‘~’ identifier                                                   (Signal type variables)
                 |  Identifier                                                       (Type constructor)
                 |  ‹type› ‹type›                                                    (Type application)
                 |  ‹type› ‘,’ ‹type›                                                (Product type)

‹typeDecl›     ::= ‹type›                                                            (Type synonym)
                 |  ‘{’ label ‘:’ ‹type› ‘}’                                         (Record declaration)

‹declaration›  ::= ‘let’ i ‹pattern› ‘=’ ‹expr›                                      (Value declaration)
                 |  ‘type’ i ‘=’ ‹typeDecl›                                          (Type declaration)
```

Figure 6.2: Hydra types syntax

$$
\begin{array}{rcl}
\langle\tau_s\rangle & ::= & \alpha_s \\
& | & \text{‘}\texttt{\textbf{real}}\text{’} \\
& | & \text{‘}\texttt{\textbf{(}}\text{’}\ \langle\tau_s\rangle *\ \text{‘}\texttt{\textbf{)}}\text{’} \\
& | & \text{‘}\texttt{\textbf{\{}}\text{’}\ \ell\ \text{‘}\texttt{\textbf{:}}\text{’}\ \langle\tau_s\rangle\ \text{‘}\texttt{\textbf{\}}}\text{’} \\
\\
\langle\tau_e\rangle & ::= & \alpha_e \\
& | & \langle\tau_s\rangle \\
& | & \text{‘}\texttt{\textbf{(}}\text{’}\ \langle\tau_e\rangle *\ \text{‘}\texttt{\textbf{)}}\text{’} \\
& | & \text{‘}\texttt{\textbf{\{}}\text{’}\ \ell\ \text{‘}\texttt{\textbf{:}}\text{’}\ \langle\tau_e\rangle\ \text{‘}\texttt{\textbf{\}}}\text{’} \\
& | & \langle\tau_e\rangle\ \text{‘}\boldsymbol{\rightarrow}\text{’}\ \langle\tau_e\rangle \\
& | & \text{‘}\texttt{\textbf{SR}}\text{’}\ \langle\tau_s\rangle
\end{array}
$$

sions can be annotated with a type for clarity or to restrict the use of a particular function or value.

The syntax of types is given in figure 6.2. Like the language itself, types have two levels. There are types for signals, denoted τ_s in figure 6.2 and types for expressions, denoted τ_e. This distinction stems from the need to limit the type of signals to types which can be given a representation as a continuously varying value, as touched upon in the previous part[a]. In the case of Hydra, a signal type may only be a real or a product (tuple or record) of signal types.

Likewise, there are two kinds of type variables, expression type variables α_e and signal type variables α_s, where signal type variables may only unify to signal types. In type annotations, a signal variable is denoted with `~a` whereas an expression type variable is denoted with `a`. This distinction allows for safe polymorphic signal relations. Consider the following signal relation:

```
let equal_constant c =
  sigrel x {
    x = c
  }
```

Naïve type inference would infer the following type:

```
equal_constant : a -> SR a
```

This function could then be applied to any expression, including expressions which do not have signal types, such as the identity function `id`. This would result in a signal relation which relates a signal of type `b -> b` which, arguably, wouldn't make much sense. The two kinds of type variables allows to instead infer the type:

```
equal_constant : ~a -> SR ~a
```

[a] See §2.4.5

Figure 6.3: Definition of $\mathcal{S}$

$$\textsc{Real}\ \frac{}{\mathcal{S}(\mathbf{real})} \qquad \textsc{Signal variable}\ \frac{}{\mathcal{S}(\alpha_s)} \qquad \textsc{Tuple}\ \frac{\mathcal{S}(\tau_1) \quad \cdots \quad \mathcal{S}(\tau_n)}{\mathcal{S}((\tau_1, \ldots, \tau_n))} \qquad \textsc{Record}\ \frac{\mathcal{S}(\tau_1) \quad \cdots \quad \mathcal{S}(\tau_n)}{\mathcal{S}(\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\})}$$

When applied to `id`, the program is rejected since the type `b -> b` cannot unify to a signal type variable.

This method is directly adapted from Standard ML's handling of equality type variables [101] which solves a similar problem, namely having a polymorphic equality function that only works for types for which equality is well-defined (excluding functions, most notably). Equality type variables are a special kind of type variables which can only unify to types which have a well-defined equality, in the same way that signal type variables in Hydra can only unify to types for which it makes sense to consider as continuously varying values. An alternative would have been to implement a constraint system in the style of Haskell's type classes [149]. While being more general[a], it is more complicated to implement and the current approach happens to fit the problem at hand particularly well.

101. Milner et al., *The Definition of Standard ML*. 1997

149. Wadler et al., 'How to Make Ad-Hoc Polymorphism Less Ad Hoc'. 1989

[a] Type classes were precisely introduced as an approach to generalising ad-hoc rules like the equality type variables of Standard ML.

The typing rules of the language are given as sets of inference rules for each syntactical construct. These rules make use of environments denoted Γ_e and Γ_s which map variables to types. Γ_e is the environment of expression variables and Γ_s is the environment of signal variables; ϵ denotes an empty environment. Figure 6.4 defines the relation $\Gamma_e, \Gamma_s \vdash s : \tau$ which states that signal s has type τ under environment Γ_e and Γ_s. Figure 6.5 defines the relation $\Gamma_e \vdash_e e : \tau$ which states that expression e has type τ under environment Γ_e. Finally figure 6.6 defines the relation $\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \mathrm{E}$ which states that equation E is well-typed. This relation is overloaded to type branches (see the Branch rule) and conditions (see the Condition rule). These relations make use of additional relations: $\mathcal{S}(\tau)$ (figure 6.3) which states that type τ is a signal type; $\mathrm{P}(\Gamma, p, \tau)$ which extends an environment Γ with variables bound in pattern p where pattern p has type τ (its definition is omitted) and $\mathscr{L}(\Gamma_e, \Gamma_s, \mathrm{B}, \mathrm{M}, \tau)$ which, given the label M of a mode, states that M is a mode defined by the set of branches B and that τ is the type of the argument of mode M under environments Γ_e and Γ_s.

6.3 — Core language

After type-checking, the surface language is transformed into a core language, which is a simplified version of the surface language. Note that this transformation essentially concerns the language of signal relation, the functional part of the language is mostly unchanged,

Figure 6.4: Hydra's signals typing rules

Expr
$$\frac{\Gamma_e \vdash e : \tau_e \qquad \mathcal{S}(\tau_e)}{\Gamma_e, \Gamma_s \vdash e : \tau_e}$$

Variable
$$\frac{}{\Gamma_e, \Gamma_s \vdash v : \Gamma_s[v]}$$

Der
$$\frac{\Gamma_e, \Gamma_s \vdash s : \mathbf{real}}{\Gamma_e, \Gamma_s \vdash \mathbf{der}\, s : \mathbf{real}}$$

BinOp
$$\frac{\Gamma_e, \Gamma_s \vdash e_1 : \mathbf{real} \qquad \Gamma_e, \Gamma_s \vdash e_2 : \mathbf{real}}{\Gamma_e, \Gamma_s \vdash e_1 \oplus e_2 : \mathbf{real}}$$

UnOp
$$\frac{\Gamma_e, \Gamma_s \vdash e : \mathbf{real}}{\Gamma_e, \Gamma_s \vdash \mathit{unop}\, e : \mathbf{real}}$$

FieldSel
$$\frac{\Gamma_e, \Gamma_s \vdash e : \{\ell_1 : \tau_1, \dots, \ell_i : \tau_i, \dots, \ell_n : \tau_n\}}{\Gamma_e, \Gamma_s \vdash e.\ell_i : \tau_i}$$

Tuple
$$\frac{\Gamma_e, \Gamma_s \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma_e, \Gamma_s \vdash e_n : \tau_n}{\Gamma_e, \Gamma_s \vdash (e_1, \dots, e_n) : (\tau_1, \dots, \tau_n)}$$

Figure 6.5: Hydra's expression typing rules

Constant
$$\frac{}{\Gamma_e \vdash c : \mathbf{real}}$$

Variable
$$\frac{}{\Gamma_e \vdash v : \Gamma_e[v]}$$

BinOp
$$\frac{\Gamma_e \vdash e_1 : \mathbf{real} \qquad \Gamma_e \vdash e_2 : \mathbf{real}}{\Gamma_e \vdash e_1 \oplus e_2 : \mathbf{real}}$$

UnOp
$$\frac{\Gamma_e \vdash e : \mathbf{real}}{\Gamma_e \vdash \mathit{unop}\, e : \mathbf{real}}$$

App
$$\frac{\Gamma_e \vdash f : \tau \to \tau' \qquad \Gamma_e \vdash s : \tau}{\Gamma_e \vdash f s : \tau'}$$

Lambda
$$\frac{\mathrm{P}\,(\Gamma_e, p, \tau) \vdash e : \tau'}{\Gamma_e \vdash \mathbf{fun}\; p \to e : \tau \to \tau'}$$

SigRel
$$\frac{\Gamma_e, \mathrm{P}\,(\epsilon, p, \tau) \vdash_{\mathrm{E}} \mathrm{E} \qquad \mathcal{S}(\tau)}{\Gamma_e \vdash \mathbf{sigrel}\; p\,\{\,\mathrm{E}\,\} : \mathbf{SR}\;\tau}$$

with the exception that all polymorphic functions are made monomorphic. That is, when a polymorphic expression is used and instantiated with a type in which no type variable (signal or expression) appears, the expression is duplicated and replaces the original expression at the use site. Naturally, when an expression is instantiated several time with the same type, a single shared duplicate is created.

At the signal level, all signals are flattened in such a way that every signal identifier has type **real**. Complex patterns are flattened to lists of identifiers [a]. Equations between product types of signals are converted into list of equations between the constituent. Finally, the derivative of signals on which `der` is applied is computed, such that `der` is only applied to variables. This can lead to a variable being differentiated multiple times, so the core language supports a notion of *n* differentiated identifiers. The identifier itself is then represented as its 0-th derivative. When the derivative of a signal appears in an event, a dummy-variable equal to said derivative is introduced and replaces the derivative in the event. This is to avoid cases like the following:

```
let bar = sigrel x {
```

[a] This is always possible since, after monomorphisation, signal types can only be nested products types of real.

Figure 6.6: Hydra's equations typing rules

$$\textsc{Equation}\quad \frac{\Gamma_e, \Gamma_s \vdash s_1 : \tau \qquad \Gamma_e, \Gamma_s \vdash s_2 : \tau}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} s_1 = s_2}$$

$$\textsc{InitEquation}\quad \frac{\Gamma_e, \Gamma_s \vdash s_1 : \tau \qquad \Gamma_e, \Gamma_s \vdash s_2 : \tau}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \textbf{init}\; s_1 = s_2}$$

$$\textsc{Local}\quad \frac{\Gamma_e, \mathrm{P}(\Gamma_s, p, \tau) \vdash_{\mathrm{E}} \mathrm{E}}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \textbf{let}\; p \{\, \mathrm{E} \,\}}$$

$$\textsc{Sequence}\quad \frac{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \mathrm{E}_1 \qquad \Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \mathrm{E}_2}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \mathrm{E}_1 ; \mathrm{E}_2}$$

$$\textsc{App}\quad \frac{\Gamma_e \vdash e : \textbf{SR}\, \tau \qquad \Gamma_e, \Gamma_s \vdash s : \tau}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} e \diamond s}$$

$$\textsc{Switch}\quad \frac{\mathscr{L}(\Gamma_e, \Gamma_s, \mathrm{M}, \mathrm{B}, \tau) \qquad \Gamma_e \vdash e : \tau \qquad \Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \mathrm{B}}{\Gamma_e, \Gamma_s \vdash_{\mathrm{E}} \textbf{switch init}\; \mathrm{M}(e)\; \mathrm{B}}$$

$$\textsc{Branch}\quad \frac{\mathscr{L}(\Gamma_e, \Gamma_s, \mathrm{M}, \mathrm{B}, \tau) \qquad \mathrm{P}\,(\Gamma_e, p, \tau)\,, \Gamma_s \vdash_{\mathrm{E}} \mathrm{E} \qquad \mathrm{P}\,(\Gamma_e, p, \tau)\,, \Gamma_s, \mathrm{B} \vdash_{\mathrm{E}} \mathrm{C}}{\Gamma_e, \Gamma_s, \mathrm{B} \vdash_{\mathrm{E}} \textbf{mode}\; \mathrm{M}\,(p) \rightarrow \mathrm{E}\; \mathrm{C}}$$

$$\textsc{Condition}\quad \frac{\Gamma_e, \Gamma_s \vdash e : \textbf{real} \qquad \mathscr{L}(\Gamma_e, \Gamma_s, \mathrm{M}, \mathrm{B}, \tau) \qquad \Gamma_e, \Gamma_s \vdash e' : \tau}{\Gamma_e, \Gamma_s, \mathrm{B} \vdash_{\mathrm{E}} \textbf{when up}(e) \;\rightarrow \mathrm{M}(e')}$$

```
  switch init Foo(0)
    mode Foo(y) -> x = y
      when up(der x) -> Foo(x)
}
```

In bar, the derivative of x is not necessarily defined, since x might be an algebraic variable. During the translation to core, a dummy-variable is introduced as follows:

```
let bar = sigrel x {
  let dx {
    dx = der x;
    switch init Foo(0)
      mode Foo(y) -> x = y
        when up(dx) -> Foo(x)
  }
}
```

The added equation and variable force the existence of der x in the signal relation and avoid any unpleasant surprises later on, such as the the derivative of x being required for detecting an event but never being properly computed by the numerical solver.

The only extension to the functional layer of the language is the ability to select tuple fields, with a syntax similar to record label selectors. This is useful when flattening signal relations of that form:

```
let ex (c : (real, real)) =
  sigrel x, y {
    (x, y) = c
  }
```

which can translate to the following:

```
let ex (c : (real, real)) =
  sigrel x, y {
    x = c.0;
    y = c.1;
  }
```

without the need to flatten c.

Figure 6.7 shows the abstract syntax of equations and signals in the core language.

In the following discussions, in particular chapter 7, the core language will be used rather than the surface language.

6.4 — A simple simulation method

In this section, a simple translation from the core language to a hybrid DAE is presented. Here a hybrid DAE is defined as a DAE paired with a set of events to monitor and a transition function, which produces a new hybrid DAE when an event is triggered. A simple simulation method is then presented for such a hybrid DAE. Finally, the feasibility and shortcomings of this representation are discussed. In particular, the difficulties associated with the translation being highly non-modular are discussed; their origin is analysed and will lead to the modular compilation scheme for Hydra which will be presented in the next chapter.

6.4.1 — *Extracting DAE from signal relations*

Given a signal relation and the modes in which each switch currently is, it is possible to extract a DAE, that is, a set of variables and undirected equations. It is in general only partial: the modular approach promoted by Hydra makes it so that most signal relations do not fully specify the value of all signals in the system.

In addition, it is possible to extract a set of additional equations that hold at the very first time instant and the set of variables to be reinitialised. For this step, it is important to know whether a given switch has just been activated through a transition. This is necessary for deciding whether a given initialisation equation is active.

Figure 6.7: Core language abstract syntax

```
  ‹signal›    ::=  ‹expr›
               |   ‘der_n’ identifier                    (n-th derivative, n statically known)
               |   ‹signal› ‹⊕› ‹signal›                            (⊕ ∈ ‘+’, ‘-’, ‘*’, ‘/’, ‘□’)
               |   ‹unop› ‹signal›                       (‹unop› ∈ ‘sin’, ‘cos’, ‘sqrt’, … )
               |   ‘(’‹signal›‘)’
               |   ‹signal› ‘,’ ‹signal›

‹condition›   ::=  ‘when’ ‹event› ‘->’ identifier‘(’identifier * ‘)’

 ‹relation›   ::=  ‹signal› ‘=’ ‹signal›
               |   ‘init’ ‹signal› ‘=’ ‹signal›
               |   ‹expr› ‘<>’ ‹signal›
               |   ‘let’ ‹pattern› ‘{’ ‹relation› ‘}’
               |   ‹relation› ‘;’ ‹relation›
               |   ‘switch’ ‘init’ ‹initMode› ‹branch*›

   ‹branch›   ::=  ‘mode’ Identifier‘(’identifier * ‘)’ ‘->’
                   ‘reinit’ identifier *  ‹equation› ‹condition›*

‹condition›   ::=  ‘when’ ‹event› ‘->’ Identifier‘(’identifier * ‘)’

    ‹event›   ::=  ‘up’ identifier
               |   ‘down’ identifier
               |   ‘updown’ identifier


  ‹pattern›   ::=  i
               |   ‘(’‹pattern›‘)’

     ‹expr›   ::=  i
               |   c
               |   ‹expr› ‹expr›
               |   ‘fun’ ‹pattern› ‘->’ ‹expr›
               |   ‘sigrel’ ‹pattern› ‘{’ ‹relation› ‘}’
```

Assuming all initialisation equations are active whenever a switch occurs would be problematic. For instance, in the example circuit that was used in this chapter, it is clear that the initialisation equation which sets the voltage through the capacitor to 0 at the first time instant should not be used if a structural change somewhere else in the circuit occurs (e.g. in the definition of the voltage source). A similar rule is in place for reinitialisation statements in branches: they only hold when the branch has just been entered through a transition, or if the branch is the first branch of the switch and the switch has just become active [a].

These informations can be encoded in a state type, which gives the state a given switch is in (containing the label of the active mode, along with the values of the mode's arguments), whether it is the first time it is active, and the state of any switch appearing inside the active branch. No other state is needed for other constructs of Hydra, in the absence of signal relation applications after inlining. The state of a list of equations can simply be represented as the list of the states of each equations.

An alternative would be to use a similar combinator as the one used in frp: switching then amounts to applying a function, which can generate initialisation equations and reinitialisation information at the point of application and then the rest of the dae. The difficulties with efficiently representing such switches have already been discussed in part I. Currently Hydra does not use this representation to avoid some complexity in the code generation, preferring the representation with explicit labels, which essentially amount to the representation discussed at the start of §3.4.3. The implementation of a general switch combinator like the one in frp is left as future work, but I would expect the work presented in part I to adapt well to the compiled setting of fhm.

Extraction functions can then be implemented that produce the list of equations, initialisation equation, reinitialisation information, etc. The advantage of explicitly representing the state outside of the equation system is that this allows a transition function to be defined simply as producing a new state, and not a whole equation system, which maps better to an implementation. Figure 6.9 gives 3 examples of such functions: R, which extracts the list of active equations from a signal relation, I, which produces the list of additional initialisation equations and Reinit, which extracts a list of reinitialised signals from the signal relation.

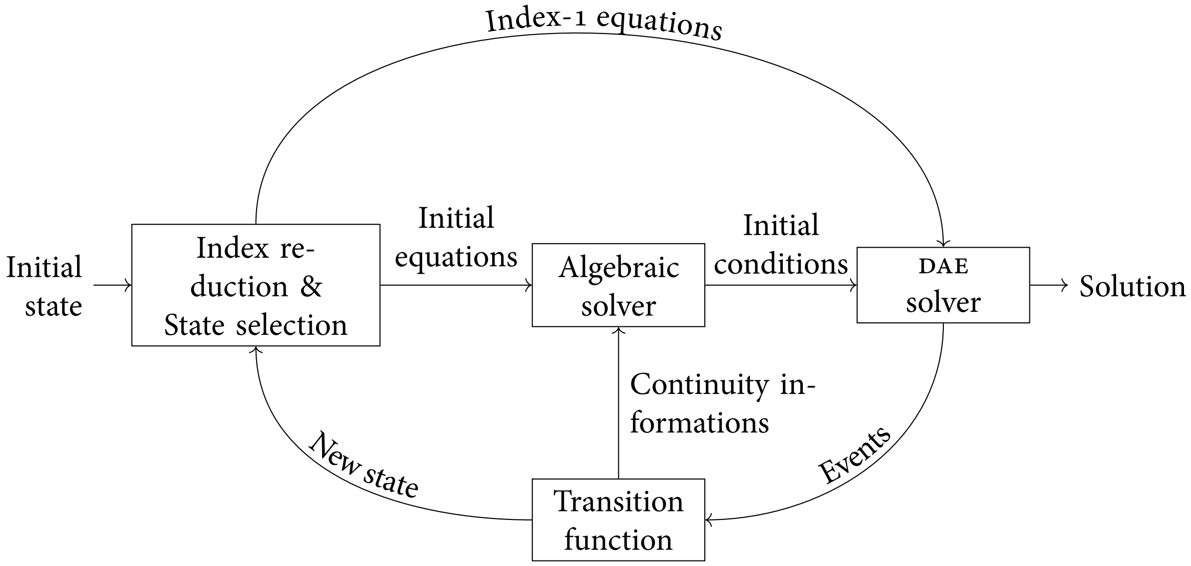
This translation of Hydra to systems of differential equations can be seen as an informal semantics for signal relations. This method is used by other non-causal languages, for instance Modelica, which gives meaning to models by specifying their translation to systems of equations [103]. A similar approach was used in [32] to propose a

[a] When a transition is taken that doesn't change the mode a switch is in, e.g. in the definition of `triangle_wave`, the switch is still considered to have been newly entered.

103. Modelica Association, *Modelica Language Specification.* 2021

32. Capper, 'Semantics Methods for Functional Hybrid Modelling'. 2014

Figure 6.8: Simulation process of a Hydra signal relation



semantics for FHM models. In [63], the author instead proposed an ideal semantics based on viewing signal relations as second-order logic predicates: signal relations were then viewed through the lens of the ideal definition of functions from time-varying value to logic properties given in §5.5.

[63] Giorgidze, 'First Class Models'. 2012

6.4.2 — *Simulation*

After the equations have been extracted, the resulting DAE, extended with its initial conditions, can be solved with the techniques presented in chapter 5. Index-reduction is performed, in order to produce a DAE of index-1, eventually differentiating some equations in the system. Initialisation can then be performed using an algebraic solver and the dynamic simulation proceeds using a DAE solver. Events are monitored by the solver and when one occurs, the transition function produces a new state, which is used to construct a new hybrid DAE, and the process repeats.

The simulation loop is summarized with the diagram in figure 6.8.

6.4.3 — *Interpretation or compilation strategies*

From the list of equations representing an index-1 DAE, it is possible to extract a residual function to provide to a numerical solver. This can be done either by providing an interpreter or by generating machine code from the list of equations.

When generating code, there are some advantages to this whole-program compilation approach, particularly for optimisations; both of the original equation systems (e.g. equalities and otherwise trivial equations can easily be eliminated) and of its representation in code:

there is usually no conditionals or jumps in the code nor function calls (apart to elementary mathematical functions), which gives the backend a lot of room to optimise.

However, it also has obvious disadvantages. As models become large, many equations appear multiple times and get duplicated unnecessarily [34, §2.6]. This causes long compilation time, which cannot be shortened by partially compiling code. [77] also cites legal and economic concerns, since it is impossible to distribute a library for a non-causal language as a compiled artifact. In the context of hybrid models, the problem is made worse, since a new DAE is generated at each mode change. For that reason, implementations of hybrid non-causal languages have usually been either interpreted [154] or made use of a JIT compiler [57, 65].

It would be much better if partial models (partial signal relations in Hydra's case) could be compiled independently. That is, if code would be generated for the residual function corresponding to the list of equations that form a signal relation once and reused wherever the signal relation is applied.

While it is possible in the context of causal modelling, it is not so with non-causal models due to higher-index systems. In such cases, additional equations must be provided that correspond to differentiated versions of existing ones. Existing index-reduction algorithms are not modular, in the sense that it is not possible to determine which equation must be differentiated without having a complete view of the system. This is also a problem for hybrid systems since the DAE of one mode may have a different index from the DAE in the other mode or require different equations to be differentiated. Determining the complete set of equations that needs to be differentiated requires enumerating every possible mode, which is not generally possible in the presence of unbounded structural dynamism. Although recent work [29] has been proposed to compute the set of latent equations for a dynamic model, the model still has to be complete at the moment the algorithm is applied.

The solution to this problem would be to generate code that can compute an arbitrary derivative of a Hydra signal. When a differentiated version of an equation is needed, there would then be no need to generate new code. However, commonly used techniques for performing differentiation of equations do not usually allow for this. The next chapter will then explore an alternative approach, called *order-parametric differentiation*, which allows for the generation of code able to compute any derivative of the expression it represents efficiently. This allows to compile Hydra ahead-of-time and in a modular fashion.

34. Casella, *Simulation of Large-Scale Models in Modelica*. 2015

77. Höger, 'Operational Semantics for a Modular Equation Language'. 2013

154. Zimmer, 'Equation-Based Modeling of Variable-Structure Systems'. 2010

57. Elmqvist et al., 'Systems Modeling and Programming in a Unified Environment Based on Julia'. 2016 — 65. Giorgidze et al., 'Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems'. 2009

29. Caillaud et al., 'Implicit Structural Analysis of Multimode DAE Systems'. 2020

Figure 6.9: Extracting a DAE automata from a signal relation

$$
\begin{array}{rcl}
\mathrm{State} & ::= & \epsilon \\
 & | & \mathrm{State} :: \mathrm{State} \\
 & | & \mathrm{Bool} \times \mathrm{Mode} \times \mathrm{State} \\
\\
\mathrm{R}(\_, x = y) & = & [x - y] \\
\mathrm{R}(\_, \mathbf{init}\ x = y) & = & [] \\
\mathrm{R}(\mathrm{S}, \mathbf{let}\ p\ \mathbf{in}\ e) & = & \mathrm{R}(\mathrm{S}, e) \\
\mathrm{R}((\_, \ell(e), \mathrm{S}), \mathbf{switch}\ \mathrm{B}) & = & \mathrm{R}(\mathrm{S}, \mathrm{B}!\ell(e)) \\
\mathrm{R}\,(\mathrm{S}_1 :: \mathrm{S}_2, e_1; e_2) & = & \mathrm{R}(\mathrm{S}_1, e_1) \mathbin{+\!\!+} \mathrm{R}(\mathrm{S}_2, e_2) \\
\\
\mathrm{I}(\_, \_, x = y) & = & [x - y] \\
\mathrm{I}(\_, \mathbf{true}, \mathbf{init}\ x = y) & = & [x - y] \\
\mathrm{I}(\_, \mathbf{false}, \mathbf{init}\ x = y) & = & [] \\
\mathrm{I}(\mathrm{S}, \mathit{new}, \mathbf{let}\ p\ \mathbf{in}\ e) & = & \mathrm{I}(\mathrm{S}, \mathit{new}, e) \\
\mathrm{I}((\mathit{new}, \ell(e), \mathrm{S}), \mathbf{switch}\ \mathrm{B}) & = & \mathrm{I}(\mathrm{S}, \mathit{new}, \mathrm{B}!\ell(e)) \\
\mathrm{I}\,(\mathrm{S}_1 :: \mathrm{S}_2, \mathit{new}, e_1; e_2) & = & \mathrm{I}(\mathrm{S}_1, \mathit{new}, e_1) \mathbin{+\!\!+} \mathrm{I}(\mathrm{S}_2, \mathit{new}, e_2) \\
\\
\mathrm{V}(\_, x = y) & = & [] \\
\mathrm{V}(\_, \mathbf{init}\ x = y) & = & [] \\
\mathrm{V}(\mathrm{S}, \mathbf{let}\ p\ \mathbf{in}\ e) & = & p\mathrm{V} :: (\mathrm{S}, e) \\
\mathrm{V}((\_, \ell(e), \mathrm{S}), \mathbf{switch}\ \mathrm{B}) & = & \mathrm{V}(\mathrm{S}, \mathit{new}, \mathrm{B}!\ell(e)) \\
\mathrm{V}\,(\mathrm{S}_1 :: \mathrm{S}_2, e_1; e_2) & = & \mathrm{V}(\mathrm{S}_1, e_1) \mathbin{+\!\!+} \mathrm{V}(\mathrm{S}_2, e_2) \\
\\
\mathrm{R}e(\_, \_, x = y) & = & [] \\
\mathrm{R}e(\_, \mathbf{true}, \mathbf{init}\ x = y) & = & [] \\
\mathrm{R}e(\_, \mathbf{false}, \mathbf{init}\ x = y) & = & [] \\
\mathrm{R}e(\mathrm{S}, \mathit{new}, \mathbf{let}\ p\ \mathbf{in}\ e) & = & \mathrm{I}(\mathrm{S}, \mathit{new}, e) \\
\mathrm{R}e((\mathbf{true}, \ell(e), \mathrm{S}), \mathbf{switch}\ \mathrm{B}) & = & \mathrm{Reinit}\,(\mathrm{B}!\ell(e)) \mathbin{+\!\!+} \mathrm{R}e(\mathrm{S}, \mathit{new}, \mathrm{B}!\ell(e)) \\
\mathrm{R}e\,(\mathrm{S}_1 :: \mathrm{S}_2, \mathit{new}, e_1; e_2) & = & \mathrm{R}e(\mathrm{S}_1, \mathit{new}, e_1) \mathbin{+\!\!+} \mathrm{I}(\mathrm{S}_2, \mathit{new}, e_2)
\end{array}
$$

7 Modular compilation of signal relations

In the previous chapter, the HYDRA language was introduced. At the end of the chapter, the interpretation of of signal relations as Differential Algebraic Equation (DAE) was presented and implementation strategies were discussed. In particular, the fact that, because of higher-index systems, which potentially requires arbitrary differentiation of part of the equation system, non-causal languages are typically not compiled modularly. The drawbacks of this whole-program approach was also touched upon.

In this chapter, to address these problems, we explore the possibility of generating code capable of computing an arbitrary derivative of a formula. Computation of arbitrary derivatives of an expression has been explored in the automatic differentiation literature. For instance, Karczmarczuk [84] showed how, exploiting lazy evaluation, derivatives of arbitrary order can be computed on demand. However, questions of efficiency aside, the runtime support needed for lazy evaluation is considerable and not a particularly natural fit with the DAE solvers and other components typically used for implementing this class of languages. Instead, this chapter presents an approach called *order-parametric differentiation* where the code generated for an expression is parametrised on the order of the derivative. It can thus be used to compute any derivative at any point (including the undifferentiated value of the expression for order 0). The objective is to generate modular code in the usual, programming-language sense of the term. Compilers for programming languages, like C, JAVA or HASKELL, compile the code for a function once and then simply use a symbol to jump into the body of the function when it is being called. This allows separately compiled modules to be joined by a linker, without further compilation as such. Simulation code for models should also be compiled into a function and use the same mechanism when one model is used in another, allowing separately compiled models to be linked in the conventional sense. The ideas presented here are applicable generally and not limited to HYDRA or FHM. Further, there are pros and cons of the proposed approach, and as such it should be viewed as a complement to existing implementation techniques for cases where true modular compilation is particularly important, not necessarily as a complete replacement for existing techniques.

84. Karczmarczuk, 'Functional Differentiation of Computer Programs'. 2001

This chapter is organised as follows. In §7.1, an intermediate form

Figure 7.1: IIR's abstract syntax

‹program›	::=	'**program**' *‹input›* *‹expr›*	
‹input›	::=	identifier '**:**' *‹τ›*	
	|	*‹input›* '**,**' *‹input›*	
‹expr›	::=	*‹atom›*	
	|	*‹atom›*'**[**'*‹atom›*'**]**'	(*Pointer offset*)
	|	*‹atom›*'□'*‹atom›*	
	|	'**unop**' *‹atom›*	
	|	*‹atom›* '**=**' *‹atom›*	(*Integer equality*)
	|	*‹atom›* '**mod**' *‹atom›*	(*Integer modulo*)
	|	*‹expr›* '**;**' *‹expr›*	
	|	'**let**' identifier '**:=**' *‹expr›* '**in**' *‹expr›*	
	|	'**allocate**' *‹τ›* '**[**'*‹atom›*'**]**'	(*Uninitialised memory allocation*)
	|	'**!**' *‹atom›*	(*Pointer derefencing*)
	|	identifier '**←**' *‹atom›*	(*Set pointer value*)
	|	'**if**' *‹atom›* '**then**' *‹expr›* '**else**' *‹expr›*	
	|	'**sum**'identifier'**from**'*‹atom›*'**to**'*‹atom›* *‹expr›* '**end**'	
	|	'**prod**'identifier'**from**'*‹atom›*'**to**'*‹atom›* *‹expr›* '**end**'	
	|	'**iter**'identifier'**from**'*‹atom›*'**to**'*‹atom›* *‹expr›* '**end**'	
‹atom›	::=	constant	(*Real and integer constants*)
	|	identifier	(*Intermediate identifier*)
	|	'**Diff**' '**(**' signal '**,**' *‹atom›* '**)**'	(*Differentiated signal*)
‹τ›	::=	⊤	(*Unit type*)
	|	$\mathbb{R}$	
	|	$\mathbb{Z}$	
	|	Pointer *‹τ›*	

(the Intermediate Imperative Representation (IIR)) is introduced that represents mathematical expressions as imperative programs. In §7.2, the idea of order-parametric differentiation is then introduced and the compilation of HYDRA equations to order-parametric IIR code is presented. In §7.3, the compilation of signal relations as a whole is then discussed. §7.4 consists of performance benchmarks of order-parametric code against more traditional approaches to the generation of simulation code.

7.1 — The Intermediate Imperative Representation

HYDRA ultimately compiles to LLVM [89], which is a low-level lan-

[89] Lattner et al., 'LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation'. 2004

guage using Static Single Assignment (ssa) form [131]. It is specifically designed to be used by compilers as an intermediate representation between the original language and machine-code. The llvm project is a large effort that provides infrastructure for compiler implementers based on the llvm intermediate representation. It provides compilers (ahead-of-time or just-in-time), optimisers, debuggers, etc. for a wide variety of architectures. For a language like Hydra, where execution speed is of prime importance but with little programmer time available to implement the compiler, llvm is a particularly good fit, as it enables the compilation of Hydra to efficient code with minimum effort.

While it is possible to compile Hydra signals and equations directly to llvm, it would make the discussion in this chapter much harder to follow: llvm is not intended to be written or read by humans and is a fairly large language of which Hydra uses a significant portion of. Having a custom intermediate language which contains exactly what is needed for compilation is therefore a better solution for the clarity of the exposition. In later section, it will also prove useful for some parts of the code generator. In particular §7.3.3 will demonstrate how some operations are in fact easier to perform over this intermediate form than over the original representation of signals.

This new language is called Intermediate Imperative Representation (iir). As its name suggests, it is a small imperative language with just the necessary constructs to compile Hydra signals to efficient machine code. Its abstract syntax is given in figure 7.1. The reader might find it helpful to pay attention to the following points. Firstly, note that signals, represented by a pair of an identifier and differentiation index are separate from mere identifiers, which contain the results of intermediate computations. Note that the differentiation index need not be a constant. Then, operators (unary or binary) can only be applied to atomic expressions. When writing example programs, this constraint will sometimes be relaxed for the sake of clarity. Finally the language supports pointers and explicit memory allocation, the latter being introduced with the keyword '**allocate**'. The way memory allocation is dealt with efficiently will be discussed in §7.3.2. Note that offsets of pointers are computed with the $x[i]$ syntax, which represents a pointer offset by i from x. Unlike in C, the pointer is not automatically dereferenced, this is done using '**!**'. Storing a value in memory location is done with the syntax $p \leftarrow x$, which stores the value x at pointer p. The language is typed, with the type system supporting integers, reals and pointers over types. The unit type $\top$ is used to give a type to the result of expression constructed with $\leftarrow$ or '**iter**'.

There is no notion of program application in the syntax. However for the purpose of clarity, the application of a program p with n arguments to n atomic expression is defined as expanding to the body of the program where the named arguments have been substituted by the

[131] Rosen et al., 'Global Value Numbers and Redundant Computations'. 1988

arguments. This application is denoted by $p(x_1, x_2, \dots, x_n)$.

7.2 — Compiling equations

7.2.1 — *Introduction to automatic differentiation*

Numerical differentiation [27] approximates the value of a derivative in a point from the original function (or program), typically by approximating the derivative with the growth rate between two points next to the point of evaluation of the derivative. It is, in essence, the opposite of numerical integration. Its main disadvantage is the imprecision it suffers from. In a general purpose modelling language, this can be problematic since the compiler is expected to handle a wide variety of system equally well. An alternative is symbolic differentiation, which operates on the analytic representation of the function to produce a representation of the derivative. Although it produces exact results, the size of the representation grows quickly and can lead to inefficiencies if naively translated to machine code, as a lot of subterms appear multiple times.

[27] Buden et al., *Numerical Analysis.* 2011

Automatic differentiation is an alternative approach that does not suffer from either problem. It doesn't have to be applied to a mathematical expression but can even be applied directly to the representation of a computer program. The idea of automatic differentiation is to consider that the derivative of every subterm in a program can be computed independently. If that subterm is named, then its derivative can be computed once and reused at will when it is required again.

Consider for instance the following function:

$$f(t) = \underbrace{\sin(\underbrace{2t}_{x_1})}_{x_2} + \underbrace{t^3}_{x_3}$$

It can be compiled as an IIR program with one input, the real t:

```
Program $t : \mathbb{R}$
    let $x_1 := 2 * t$ in
    let $x_2 := \sin(x_1)$ in
    let $x_3 := t^3$ in
    $x_2 + x_3$
```

Suppose one wished to emit a program that could compute f', given the value of t. By symbolic differentiation, the expression for f' would be:

$$f'(t) = 2\cos(2t) + 3t^2$$

The idea of automatic differentiation is that the program that implements this function, reduces to a composition of elementary operations and functions. This is particularly visible in IIR where operators are only applied to atomic expressions. Consider the program representing *f*. The final expression in the program x_2+x_3 can be easily differentiated knowing the derivative of x_2 and x_3. These derivatives can be computed easily from their definition, which might require further differentiation of other definitions or other definitions to be introduced if the derivative cannot be computed with just one operation, for instance the derivative of the product requires two multiplications and an addition. This procedure describes forward mode automatic differentiation and, applied to the code for *f*, it yields the following program for f':

```
Program $t : \mathbb{R}$
    let $x_1$ $:= 2 * t$ in
    let $x_1'$ $:= 2$ in
    let $x_2$ $:= \sin(x_1)$ in
    let $y_1$ $:= \cos(x_1)$ in
    let $x_2'$ $:= x_1' * y_1$ in
    let $x_3$ $:= t^3$ in
    let $y_2$ $:= t^2$ in
    let $x_3'$ $:= 3 * y_2$ in
    $x_2' + x_3'$
```

One can then repeat the operation to compute higher-order derivative, like the second order derivative of *f* below. Note that it is beneficial in this case to remember whether an identifier binds the derivative of a previous identifier or not, e.g. it is important to remember that x_1' is the first-order derivative of x_1:

```
Program $t : \mathbb{R}$
    let $x_1$ $:= 2 * t$ in
    let $x_1'$ $:= 2$ in
    let $x_1''$ $:= 0$ in
    let $x_2$ $:= \sin(x_1)$ in
    let $y_1$ $:= \cos(x_1)$ in
    let $y_1'$ $:= x_1' * x_2$ in
    let $x_2'$ $:= x_1' * y_1$ in
    let $y_3$ $:= x_1'' * y_1$ in
    let $y_4$ $:= x_1' * y_1'$ in
    let $x_2''$ $:= y_3 + y_4$ in
    let $x_3$ $:= t^3$ in
    let $y_2$ $:= t^2$ in
    let $y_2'$ $:= 2 * t$ in
    let $x_3'$ $:= 3 * y_2$ in
    let $x_3''$ $:= 6 * y_2'$ in
    $x_2'' + x_3''$
```

In contrast with symbolic differentiation, automatic differentiation allows previously computed values of the derivatives of subexpressions to be reused and, in general, only yields code that is a constant factor larger than the original code [70].

70. Griewank et al., *Evaluating Derivatives*. 2008

7.2.2 — *Order-parametric differentiation*

Automatic differentiation is thus an efficient technique to compute the derivative of a function represented by a program. While applying it repeatedly allows for higher-order derivatives to be computed, the requirement to generate modular code is that a single program can compute any derivative of an expression. This clearly cannot be done with the first-order scheme that was presented earlier without resorting to recompilation or symbolic manipulation.

The idea of this work is instead to consider other rules of differentiation, that are parametric in the order of differentiation. The mathematical background is established in this section, which shows out to obtain symbolic formulations for the n-th derivative of mathematical expressions. A generalisation of the automatic differentiation strategy presented earlier is then established; and the full translation of Hydra signals into IIR programs able to compute an arbitrary derivative of a signal is shown.

7.2.2.1 — *Mathematical background* Let's consider this simple function, defined for some constant $k \in \mathbb{R}$:

$$f(t) = t^k$$

The successive derivatives of f are:

$$\begin{aligned} f'(t) &= kt^{k-1} \\ f''(t) &= k(k-1)t^{k-2} \\ f^{(3)}(t) &= k(k-1)(k-2)t^{k-3} \\ &\vdots \\ f^{(10)}(t) &= k(k-1)\cdots(k-9)t^{k-10} \\ &\vdots \end{aligned}$$

A pattern quickly becomes apparent and one can see that $f^{(n)}$ is given by the following formula [a]:

$$f^{(n)}(t) = t^{k-n} \prod_{i=0}^{n-1} (k-i)$$

[a] Note that, when k is a natural number, this formula also gives $f^{(n)}(t) = 0$, when $n > k$, since one of the factors in the product is $k - k$.

Many common functions admit such formulas for their n-th derivatives. In particular this holds for all the functions supported by Hydra and by the C math header library [81], which are given in table 7.1. This table also gives the formulas for multiplication and division by treating it as a product and an exponentiation with exponent -1.

81. ISO, *ISO C Standard 1999*. 1999

Table 7.1: n-th derivative of unary functions

Expression	n-th derivative	Comment
$\exp(t)$	$\exp(t)$	
$\ln(t)$	$\ln(t) \quad \text{if } n = 0$ $\left(\frac{1}{t}\right)^{(n-1)} \quad \text{otherwise}$	
t^k	$t^{k-n} \prod_{i=0}^{n-1} (k - i)$	For any constant $k \in \mathbb{R}$
k^t	$\ln(k)^n k^t$	For any strictly positive real constant k
$p(t)^{q(t)}$	$(\exp(q(t) \ln(p(t))))^{(n)}$	
$\sin(t)$	$\sin(t) \quad \text{if } n = 4k$ $\cos(t) \quad \text{if } n = 4k + 1$ $-\sin(t) \quad \text{if } n = 4k + 2$ $-\cos(t) \quad \text{if } n = 4k + 3$	
$\cos(t)$	$\sin^{(n+1)}(t)$	
$\tan(t)$	$\left(\frac{\sin(t)}{\cos(t)}\right)^{(n)}$	
$\sinh(t)$	$\sinh(t) \quad \text{if } n \text{ is even}$ $\cosh(t) \quad \text{if } n \text{ is odd}$	
$\cosh(t)$	$\sinh^{(n+1)}(t)$	
$\tanh(t)$	$\left(\frac{\sinh(t)}{\cosh(t)}\right)^{(n)}$	
$\text{asin}(t)$	$\text{asin}(t) \quad \text{if } n = 0$ $\left(\frac{1}{\sqrt{1-t^2}}\right)^{(n-1)} \quad \text{if } n > 0$	
$\text{acos}(t)$	$\text{acos}(t) \quad \text{if } n = 0$ $\left(-\frac{1}{\sqrt{1-t^2}}\right)^{(n-1)} \quad \text{if } n > 0$	
$\text{atan}(t)$	$\text{atan}(t) \quad \text{if } n = 0$ $\left(\frac{1}{1+t^2}\right)^{(n-1)} \quad \text{if } n > 0$	
$\text{asinh}(t)$	$\text{asinh}(t) \quad \text{if } n = 0$ $\left(\frac{1}{1+t^2}\right)^{(n-1)} \quad \text{if } n > 0$	
$\text{acosh}(t)$	$\text{acosh}(t) \quad \text{if } n = 0$ $\left(-\frac{1}{\sqrt{t^2-1}}\right)^{(n-1)} \quad \text{if } n > 0$	
$\text{atanh}(t)$	$\text{atanh}(t) \quad \text{if } n = 0$ $\left(\frac{1}{1-t^2}\right)^{(n-1)} \quad \text{if } n > 0$	
$\Gamma(t)$	$\Gamma(t)\mathrm{D}_n(t)$ $\text{where} \begin{cases} \mathrm{D}_1(t) = \psi(t) \\ \mathrm{D}_k(t) = \mathrm{D}'_{k-1}(t) + \psi(t)\mathrm{D}_{k-1}(t) \end{cases}$	ψ is the digamma function. Programs for computing its value can be found in [85].
$p(t)q(t)$	$\sum_{i=0}^{n} \binom{n}{i} p^{(i)}(t) q^{(n-i)}(t)$	
$\frac{p(t)}{q(t)}$	$\left(p(t)q(t)^{-1}\right)^{(n)}$	

Some of these functions do not have a direct formula for their derivatives, but can be expressed in terms of combinations of other functions. This is the case for tan and tanh, which can be expressed as respectively $\frac{\sin}{\cos}$ and $\frac{\sinh}{\cosh}$. The derivatives of some functions, such as the inverse trigonometric and hyperbolic functions, can be computed from their $(n-1)$-th derivative. Some of the formulations given here are not optimal, this will be discussed in section 8.3.1.

A crucial rule is missing from table 7.1: the one for composition. Computing the n-th derivative of a composition of two functions can be done using Faà di Bruno's formula. It is a generalisation of the chain rule, which states that $(f \circ g)'(t) = g'(t)f'(g(t))$. The formula makes use of the notion of partitions of a natural number, which is defined as:

Definition 7.2.1 (Partitions of natural numbers). *The partitions of a natural number n is the set of n-tuple of natural numbers* $(m_1, \dots, m_n)$ *such that:*

$$1 \cdot m_1 + 2 \cdot m_2 + \cdots + n \cdot m_n = n$$

A partition of n corresponds to a way n can be split into smaller numbers. For instance, 4 has 4 partitions $(0,0,0,1)$ $(4 = 1 \times 4)$, $(1,0,1,0)$ $(4 = 1 \times 1 + 2 \times 1)$, $(2,1,0,0)$ $(4 = 2 \times 1 + 1 \times 2)$, $(0,2,0,0)$ $(4 = 2 \times 2)$ and $(4,0,0,0)$ $(4 = 4 \times 1)$. In the following, P_n denotes the set of partitions of n and $|\mathrm{P}_n|$ the number of partitions of n, simply called the *partition number* [71]. From that definition, it is possible to define the n-th derivative of the composition of two functions.

Theorem 7.2.1 (Faà di Bruno's formula). *The n-th derivative of the composition of two* $\mathcal{C}^n(\mathbb{R})$ *functions f and g is a* $\mathcal{C}^n(\mathbb{R})$ *function given by the following formula [5, 60, 61]:*

$$(f \circ g)^{(n)}(t) = \sum_{(m_1,\dots,m_n) \in \mathrm{P}_n} \frac{n!}{m_1!1!^{m_1} m_2!2!^{m_2} \cdots m_n!n!^{m_n}} \cdot f^{(m_1+\cdots+m_n)}(g(t)) \cdot \prod_{j=1}^{n} \left(g^{(j)}\right)^{m_j} \tag{Faà di Bruno's formula}$$

The formula is quite daunting, however one can get a better sense of what it means by defining the following quantities associated with a partition $p = (m_1, \dots, m_n) \in \mathrm{P}_n$:

$$\eta(p) = \frac{n!}{m_1!1!^{m_1} m_2!2!^{m_2} \cdots m_n!n!^{m_n}}$$
$$\sigma(p) = m_1 + m_2 + \cdots + m_n$$

Then the formula becomes:

$$(f \circ g)^{(n)}(t) = \sum_{p \in \mathrm{P}_n} \eta(p) \cdot f^{(\sigma(p))}(g(t)) \cdot \prod_{j=1}^{n-\sigma(p)+1} \left(g^{(j)}\right)^{m_j}$$

71. Hardy et al., *An Introduction to the Theory of Numbers*. 2008

5. Arbogast, 'Du calcul des dérivations'. 1800 — 60. Faà Di Bruno, 'Note sur une nouvelle formule de calcul différentiel'. 1857 — 61. Faà di Bruno, 'Sullo sviluppo delle funzioni'. 1855

It is simply a sum of terms of the form of a derivative of f multiplied by some derivatives of g. Note that in the formula above, the bound on the inner product has been shortened to $n - \sigma(p) + 1$. Indeed, it is easy to show that $m_j = 0$ if $j > n - \sigma(p) + 1$. Another formulation, which will prove useful, consists of factoring the terms of the form $f^{(\sigma(p))}$. $\sigma(p)$ must be between 0 and n, but there are typically many more partitions in P_n. The corresponding factors can be expressed in terms of the partial Bell polynomials [9], which are defined for a given n and k as:

9. Bell, 'Partition Polynomials'. 1927

$$\mathrm{B}_{n,k}(x_1, \dots, x_{n-k+1}) = \sum_{\substack{p \in \mathrm{P} \\ \sigma(p)=k}} \eta(p) \cdot \prod_{j=1}^{n-k+1} \left(x_j\right)^{m_i}$$

Note that the sum spans on those partitions whose coefficients sum to k. It follows that Faà di Bruno's formula can then be rewritten as:

$$(f \circ g)^{(n)} = \sum_{i=1}^{n} f^{(k)}(g(t)) \cdot \mathrm{B}_{n,k}\left(g'(t), g''(t), \dots, g^{(n-k+1)}(t)\right)$$

7.2.2.2 — *A generalised automatic differentiation scheme* The efficiency of the proposed scheme will be discussed in greater details in later sections, in particular the efficiency of Faà di Bruno' formula. However it is enough for now to recognise that the formulas we've discussed in section 7.2.2.1 can be used to express Hydra signals as iir programs. However, like in the first-order case, the translation of symbolic expressions to programs must make use of memoisation as much as possible. In the first-order case, this was done by associating to every identifier in the original iir program, an identifier in the differentiated iir program. In the present case, as there is an unknown number of derivatives being computed, one must then associated an array containing the n derivatives of the identifier. The rules of differentiation or order-parametric differentiation can then be applied in exactly the same way.

The translation from Hydra to order-parametric iir programs is now presented in the form of the function T, which given an Hydra signal produces a program that takes as input the order of differentiation n and a real pointer pointing to a block of reals of size n that will contain the values of the first n derivatives of the compiled signal after the execution of the program. This translation makes use of an intermediate function C, defined in figure 7.3, which produces a program that, given an array containing the n values of $f^{(i)}(g(t))$ and an array containing the values of $g^{(j)}(t)$, computes the first n-th derivative of $f \circ g$ using Faà di Bruno's formula.

7.2.3 — *Computing with Faà di Bruno's formula*

Let's consider the program in figure 7.3 and expand the computation of $f \circ g_n$ for $n = 4$ and $n = 5$. Assuming the needed information

```
$T(D_k(v))$ =
    Program $N : \mathbb{Z}, r : \text{Pointer } \mathbb{R}$
        iter $n$ from $0$ to $N$
            let $r_n := r[n]$ in
            let $npk := n + k$ in
            $r_n \leftarrow \text{Diff}(v, npk)$
        end
```

```
$T(e)$ =
    Program $N : \mathbb{Z}, r : \text{Pointer } \mathbb{R}$
        $r \leftarrow e$
        iter $n$ from $1$ to $N$
            let $r_n := r[n]$ in
            $r_n \leftarrow 0$
        end
```

```
$T(e_1 + e_2)$ =
    Program $N : \mathbb{Z}, r : \text{Pointer } \mathbb{R}$
        let $a_1$ := allocate $\mathbb{R}$ [N] in
        $T(e_1)(N, a_1)$;
        let $a_2$ := allocate $\mathbb{R}$ [N] in
        $T(e_2)(N, a_2)$;
        iter $n$ from $0$ to $N$
            $r[n] \leftarrow !a_1[n] + !a_2[n]$
        end
```

```
$T(\sin(e))$ =
    Program $N : \mathbb{Z}, r : \text{Pointer } \mathbb{R}$
        let $g$ := allocate $\mathbb{R}$ [N] in
        $T(e)(N, g)$
        let $h$ := allocate $\mathbb{R}$ [N] in
        iter $n$ from $0$ to $N$
            let $h_n := h[n]$ in
            let $g_0 := g[0]$ in
            let $f_n$ :=
                let Sign :=
                    if $n/2$ mod $2 = 0$ then
                        $1$
                    else
                        $-1$
                in
                let SinCos :=
                    if $n$ mod $2 = 0$ then
                        $\sin(g_0)$
                    else
                        $\cos(g_0)$
                in
                Sign $*$ SinCos
            in
            $h_n \leftarrow f_n$
        end
        $C(h, g, r)$
```

Figure 7.2: Translation of Hydra signals to order-parametric IIR programs

Only the + is used to illustrate the translation for binary operators and only sin is used to illustrate the translation for unary operators.

```
**Program** N $: \mathbb{Z}, f : \text{Pointer } \mathbb{R}, g : \text{Pointer } \mathbb{R}, r : \text{Pointer } \mathbb{R}$
    **iter** $n$ **from** 0 **to** N
        **let** $p_n := !p[n]$ **in**
        **let** $\eta_n := !\eta[n]$ **in**
        **let** $\sigma_n := !\sigma[n]$ **in**
        **let** $\mathrm{P}_n := !\mathrm{P}[n]$ **in**
        **let** $f \circ g_n :=$
            **sum** $i$ **from** 0 **to** $p_n$
                **let** $\eta_i := !\eta_n[i]$ **in**
                **let** $\sigma_i := !\sigma_n[i]$ **in**
                **let** $\mathrm{P}_i := !\mathrm{P}_n[i]$ **in**
                **let** $\Pi_i :=$
                    **prod** $j$ **from** 1 **to** $n - \sigma_i + 1$
                        **let** $m_j := !\mathrm{P}_i[j]$ **in**
                        $g[j]^{m_j}$
                    **end**
                **in**
                $\eta_i * !f[\sigma_i] * \Pi_i$
            **end**
        **in**
        $r[n] \leftarrow f \circ g_n$
    **end**
```

Figure 7.3: Naïve program for computing the first n-th derivative of a function composition

This function assumes the existence of three global values: p, η, σ and P. p is an integer pointer pointing to an arbitrarily large block of memory, such that $p[n]$ points to the partition number of n. η and σ are pointer to pointers of integers. $\eta[n]$ (resp. $\sigma[n]$) points to a block of memory of size the partition number of n and contains the values for η (resp. σ) for a given partition.

from the partitions of 4 and 5 has already been computed, and after flattening every loop, one gets:

$$\begin{aligned} f \circ g_4 \leftarrow\ & 1*!y_1[1]*!x_1[4] \\ & + 4*!y_1[2]*!x_1[1]*!x_1[3] \\ & + 3*!y_1[2]*!x_1[2]^2 \\ & + 6*!y_1[3]*!x_1[1]^2*!x_1[2] \\ & + 1*!y_1[4]*!x_1[1]^4 \end{aligned} \qquad \begin{aligned} f \circ g_5 \leftarrow\ & 1*!y_1[1]*!x_1[1] \\ & + 10*!y_1[2]*!x_1[2]*!x_1[3] \\ & + 5*!y_1[2]*!x_1[1]*!x_1[4] \\ & + 15*!y_1[3]*!x_1[1]*!x_1[2]^2 \\ & + 10*!y_1[3]*!x_1[1]^2*!x_1[3] \\ & + 10*!y_1[4]*!x_1[3]*!x_1[2] \\ & + 1*!y_1[5] * x_1[1]^5 \end{aligned}$$

Consider further that the exponentiation is implemented as repeated multiplication [a]. It is clear that there are a lot of redundant operations between the two computations. By comparison, repeated first-order differentiation can exploit these redundancies either by simple memoisation or by performing Common Subexpression Elimination (CSE) after the construction of the expression. However, in our case, one cannot perform such elimination easily, as the actual values of the exponents are hidden inside data structures and a clear relationship between them is non-trivial.

[a] LLVM implements exponentiation of floating point number by exponentiation by squaring, which performs O(log(n)) multiplications.

By implementing Faà di Bruno's formula directly, one can see that, indeed, most of the time is spent computing the products of the derivatives of g for all the partitions. This is therefore where progress must be made. In this section, two reformulations of Faà di Bruno's formula are presented that makes the computation of the first n-th derivatives much more efficient. One relies on the structure of partitions, in particular the relationship between a partition of n and its successors. The other relies on a recurrence relation on the partial Bell polynomials.

7.2.3.1 — *Exploiting the structure of partitions* In the following, γ_p denotes the following function associated with a partition p of n:

$$\gamma_p(x_1, \dots, x_n) = \prod_{j=1}^{n-\sigma(p)+1} x_j^{m_j}$$

One could then rewrite Faà di Bruno formula (or the Bell's polynomials) in terms of γ_p. As mentioned in the previous section there is a need to optimise the computation of this function in Faà di Bruno's formula. To do so, let's consider the following lemma [a].

Lemma 7.2.1 (Child partition). *Let $p = (m_1, \dots, m_n)$ be a partition of n. Then for all $j \in [1, n]$, $p_j = \left(m_1, \dots, m_j + 1, \dots, m_n, 0, \dots, 0\right)$ is a partition of $n + j$.*

The partition p_j is the j-child of p. Respectively, p is called the j-ancestor of p_j.

A partition therefore has many children, which differ from it by only one coefficient. More importantly, this means that γ_p is related with γ_{p_j} through the following relation:

$$\gamma_{p_j}\left(x_1, \dots, x_{n+j}\right) = x_j \gamma_p(x_1, \dots, x_n)$$

Hence if one knows, when computing a term in $(f \circ g)^{(n)}$, an ancestor to a partition of n and the value of γ for that partition, then one can avoid having to compute many multiplications.

Lemma 7.2.2 (Existence of an ancestor). *All partitions of n except the partition $(0, \dots, 0, 1)$ have at least one ancestor.*

Naïvely, one could then look at these results and suggest the following scheme to compute the partitions of n from the partitions of its predecessor: simply take the j-child of each partitions of $n - j$, for j running from 1 to $n - 1$. While indeed one would get all the partitions of n this way, except the trivial partition that doesn't have an ancestor, one would also get duplicates since an ancestor isn't unique in general. To use this method, it would be better to have a technique that would consider only some children of the partitions of $n-j$, in such a way that one gets every partition of n exactly ones. The set of ancestors selected is then a set of canonical ancestors for the partitions of n: this set can

[a] To avoid distracting from the discussion, proofs of the lemmas and theorems given in this section are given in appendix A.

be used safely to compute the value of γ from the previously computed values of γ for these ancestor partitions.

A very good canonical ancestor is the *closest* ancestor. By definition, there can be at most one j-child or j-ancestor for any partition. It follows that there exist, for every partition with at least one ancestor, a closest ancestor, a j-ancestor such that the partition has no k-ancestor with $k < j$. The closest ancestor can be determined like so:

Lemma 7.2.3 (Closest ancestor). *If $p = (m_1, \dots, m_n)$ is a partition of n whose first non-zero coefficient is m_j, where $j < n$, then the closest ancestor of p is its j-ancestor.*

which entails:

Theorem 7.2.2 (Closest children). *If p is a partition of n whose first j coefficients are zero, then it is the closest ancestor of its first $j+1$ children.*

In the following, the closest child of a partition p is a child for which p is the closest ancestor. It is now possible to compute the set of partitions of n from the partitions of the predecessors of n.

Theorem 7.2.3 (Recursive constructions of the partitions). *The partitions of n are made of the partition $(0, \dots, 0, 1)$ and of the j closest children of the partitions of $n - j$, for all $j \in [1, n[$. This construction results in no duplicate partition.*

By ordering the partitions of n in anti-lexicographic order[a], the partitions are also ordered in such a way that the partitions that have a closest 1-child appear first, then the partitions that have a closest 2-child, etc. If the data on partitions is stored in such order, it is possible to store an array of n indices, such that the entry at index j points to the the first partition whose j-th coefficient is 0, meaning that this partition and all the following are the closest ancestor of their children.

Example. *Let's try reconstructing the partitions of 6 using ancestry relations with the partitions of previous numbers. Below are the partitions of numbers from 1 to 5, organised in anti-lexicographic order.*

1	*2*	*3*	*4*	*5*
(1)	*(2,0)* *(0,1)*	*(3, 0, 0)* *(1, 1, 0)* *(0, 0, 1)*	*(4, 0, 0, 0)* *(2, 1, 0, 0)* *(1, 0, 1, 0)* *(0, 2, 0, 0)* *(0, 0, 0, 1)*	*(5, 0, 0, 0, 0)* *(3, 1, 0, 0, 0)* *(2, 0, 1, 0, 0)* *(1, 2, 0, 0, 0)* *(1, 0, 0, 0, 1)* *(0, 1, 0, 1, 0)* *(0, 0, 0, 0, 1)*

The lines are used to indicate the number of zero coefficients at the start of the partition (partitions below the first line have their first coefficient equal to 0, partitions below the second line have their first and second coefficients equal to 0, etc.). If a partition is below a total of k lines, it has therefore a 1-closest child, a 2-closest child, etc. and a $k+1$-

[a] The partitions with the largest first coefficient appear first, then the partitions with a first coefficient of zero and the largest second coefficient, etc.

closest child. For instance, the partition (0, 0, 0, 1) of 4 is below 3 lines in total, indicating that its first 3 coefficients are 0.

To construct the partitions of 6 using these informations, we must first take the 1 closest children of the partitions of 5. All the partitions of 5 have a 1-closest child, giving:

(6, 0, 0, 0, 0, 0)
(4, 1, 0, 0, 0, 0)
(3, 0, 1, 0, 0, 0)
(2, 2, 0, 0, 0, 0)
(2, 0, 0, 0, 1, 0)
(1, 1, 0, 1, 0, 0)
(1, 0, 0, 0, 1, 0)

Then, the 2-closest children of the partitions of 4, these corresponds to the partitions of 4 below the first horizontal line (and whose first coefficient is 0), giving:

(0, 3, 0, 0, 0)
(0, 1, 0, 1, 0)

Then, the 3-closest children of the partitions of 3, of which there are only one:

(0, 0, 2, 0, 0, 0)

There are no valid ancestors in the partitions of 2 and 1, since those only have 2 and 1-children respectively. Adding the trivial partition:

(0, 0, 0, 0, 0, 1)

gives the 11 partitions of 6.

An implementation of Faà di Bruno's formula using ancestry relations is given as an IIR program in figure 7.4.

7.2.3.2 — *Exploiting Bell polynomials* Another strategy relies on the following recurrence relation on Bell polynomials [43]:

43. Comtet, *Advanced Combinatorics*. 2012

$$\begin{aligned} \mathrm{B}_{0,0} &= 1 \\ \forall n \geq 1, \mathrm{B}_{n,0} &= 0 \\ \forall k \geq 1, \mathrm{B}_{0,k} &= 0 \\ \mathrm{B}_{n,k}\left(x_1, \dots, x_{n-k+1}\right) &= \sum_{i=1}^{n-k+1} \binom{n-1}{k-1} \cdot x_i \cdot \mathrm{B}_{n-i,k-1}\left(x_1, \dots, x_{n-i-k}\right) \end{aligned}$$

Recall that Faà di Bruno's formula can be expressed using Bell's polynomials as follows:

$$(f \circ g)^{(n)}(t) = \sum_{i=1}^{n} f^{(k)}(g(t)) \cdot \mathrm{B}_{n,k}\left(g'(t), g''(t), \dots, g^{(n-k+1)}(t)\right)$$

Knowing the values of the Bell polynomials applied to g for previous values of n, one can then compute $(f \circ g)^{(n)}$ like so:

$$(f \circ g)^{(n)} = \sum_{k=1}^{n} f^{(k)}(g(t)) \sum_{i=1}^{n-k+1} \binom{n-1}{i-1} g^{(i)}(t) \mathrm{B}_{n-i,k-1}\left(g'(t), \dots, g^{(n-i-k)}(t)\right)$$

```
Program $N : \mathbb{Z}, f : \text{Pointer } \mathbb{R}, g : \text{Pointer } \mathbb{R}, r : \text{Pointer } \mathbb{R}$
    let $G :=$ allocate $\text{Pointer } \mathbb{R}\ [N + 1]$ in
    $r[0] \leftarrow !f[0]$
    iter $n$ from $1$ to $N$
        let $P_n := !P[n]$ in
        let $G_n :=$ allocate $\mathbb{R}\ [P_n]$ in
        $G[n] \leftarrow G_n$
        let $\eta_n := !\eta[n]$ in
        let $\sigma_n := !\sigma[n]$ in
        let $r_1 :=$
            sum $k$ from $1$ to $n$
                let $G_{n-k} := !G[n - k]$ in
                let $A_{n-k} := !A[n - k]$ in
                let $s := !A_{n-k}[k - 1]$ in
                let $P_{n-k} := !P[n - k]$ in
                sum $j$ from $s$ to $P_{n-k}$
                    let $G_{n-k,j} := !G_{n-k}[j]$ in
                    $G_n[j] \leftarrow !g[k] * G_{n-k,j}$
                    let $\eta_j := !\eta_n[j]$ in
                    let $\sigma_j := !\sigma_n[j]$ in
                    $!\eta_j * !f[\sigma_j] * !G[j]$
                end
            end
        in
        $G_n[P_n - 1] \leftarrow !g[n]$
        $r[n] \leftarrow r_1 + !f[1] * !g[n]$
    end
```

Figure 7.4: Program for computing the first n-th derivative of a function composition using ancestry relations

This program assumes the existence of the same global variables as the program in figure 7.3. In addition, it requires the existence of an array A, where $A[n][k]$ returns the index of the first partition of n whose first k coefficients are 0. Unlike the naïve implementation, this one relies on the information on partitions being ordered in the specific way outlined in §7.2.3.1.
In the program, G is an array which contains the memoised values of all the values of γ_p computed so far. It is populated when computing r_1, which corresponds to the value of $(f \circ g)^{(n)}$ minus the value contributed by the trivial partition, which is added on at the end.

To compute the first n derivatives of $f \circ g$, this scheme requires exactly $\frac{(n+1)(n+2)}{2}$ storage space. This is demonstrated on the program in figure 7.5.

7.2.3.3 — *Measuring performance* To measure the performance of Faà di Bruno's formula, the programs in figure 7.3, 7.4 and 7.5 have been translated to C code [a] and benchmarked using Google's benchmark library [68]. The code was compiled with Clang 12 on a PC running ArchLinux with kernel 5.12.9 with a 4-core Intel i3-7100T @ 3.40 GHz processor. The results are presented in figure 7.7.

An additional benchmark is included, which uses the naïve formulation but exploits the fact that partitions are sparse, the larger the number, the more 0 will appear as a coefficient in its partitions. This is illustrated in figure 7.6. While it shows an improvement on the benchmarks, the benefits are not sufficient to compete with the cleverer formulation fleshed-out in the earlier sections.

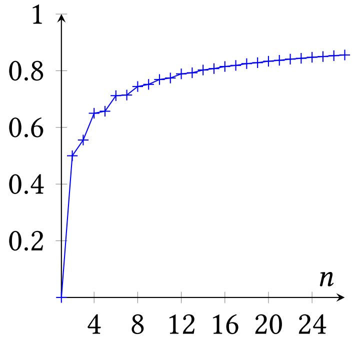
[a] The code can be found at `https://gitlab.com/chupin/fdb_bench`.

68. Google, *Benchmark*. 2021

```
Program N : $\mathbb{Z}$, $f$ : Pointer $\mathbb{R}$, $g$ : Pointer $\mathbb{R}$, $r$ : Pointer $\mathbb{R}$
    let B := allocate Pointer $\mathbb{R}$ [N + 1] in
    $r[0] \leftarrow !f[0]$
    let $\mathrm{B}_0$ := allocate $\mathbb{R}$ [1] in
    $\mathrm{B}[0] \leftarrow \mathrm{B}_0$
    $\mathrm{B}_0[0] \leftarrow 1$
    iter $n$ from 1 to N
        let $\mathrm{B}_n$ := allocate $\mathbb{R}$ [$n$] in
        $\mathrm{B}[n] \leftarrow \mathrm{B}_n$
        let $f \circ g_n$ :=
            sum $k$ from 1 to $n$
                let $\mathrm{B}_{n,k}$ :=
                    sum $i$ from 1 to $n - k + 1$
                        binom$(n - 1, i - 1) * !g[i] * !(!\mathrm{B}[n - i])[k - 1]$
                    end
                in
                $\mathrm{B}[n][k] \leftarrow \mathrm{B}_{n,k}$
                $!f[k] * \mathrm{B}_{n,k}$
            end
        in
        $r[n] \leftarrow f \circ g_n$
    end
```

Figure 7.5: Program for computing the first n-th derivative of a function composition using Bell's polynomials recurrence relation

Figure 7.6: Sparsity of partitions. The graph shows the proportion of coefficients that are 0 in the partitions of n.



The benchmark clearly shows that using either the ancestry relations derived in section 7.2.3.1 or the recurrence relations on Bell's polynomial from section 7.2.3.2 provides a much better way to compute the first n derivatives of composition. Choosing between the two simply by virtue of these benchmarks doesn't seem very easy. Figure 7.8 especially focuses on comparing the two formulations by plotting the ratio of the time taken to compute using Bell's polynomials recurrence relation over the time taken to compute using ancestry relations. While it seems that the recurrence relation gives overall better results, especially for lower number of differentiations (that one except to encounter more often) and high numbers of differentiation, the difference is still small. Other characteristics of each approach, besides performance, can help guide the decision on which one to prefer for an implementation.

Using Bell's recurrence relation has the particular advantage that it never uses any informations on the partitions of the integers. Indeed, it only requires to know about the binomial coefficients, which are needed anyway to compute the n-th derivative of the product. These coefficients are much easier to compute, using Pascal's triangle [117], than the coefficients over partitions and there are a lot fewer of them: for a given n, there are $n+1$ non-zero binomial coefficients of the form

[117] Pascal, *Traité du triangle arithmétique*. 1654

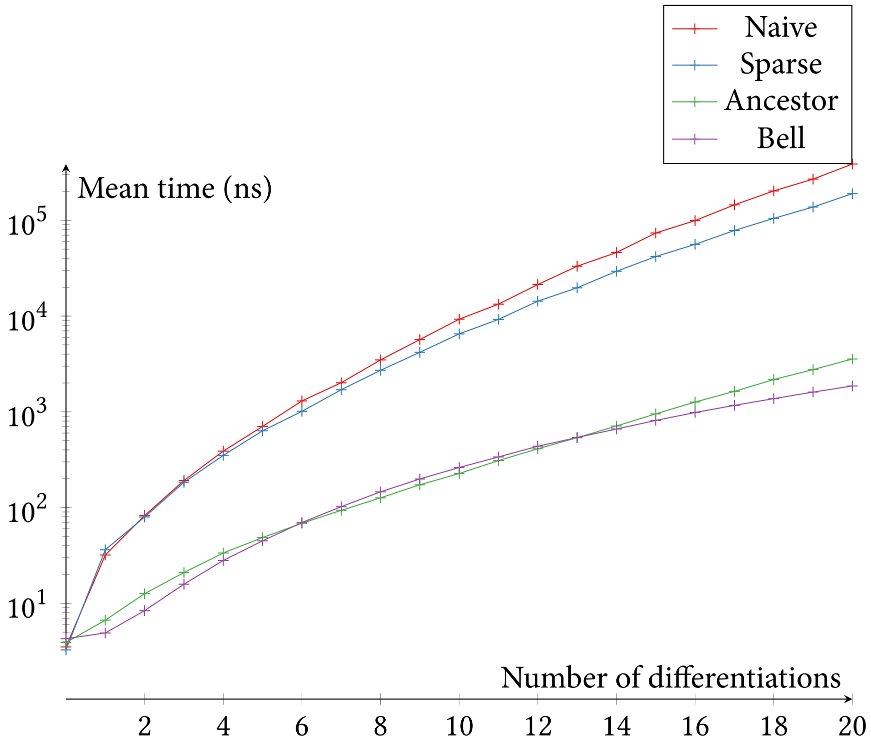


Figure 7.7: Performance of different implementation of Faà di Bruno's formula

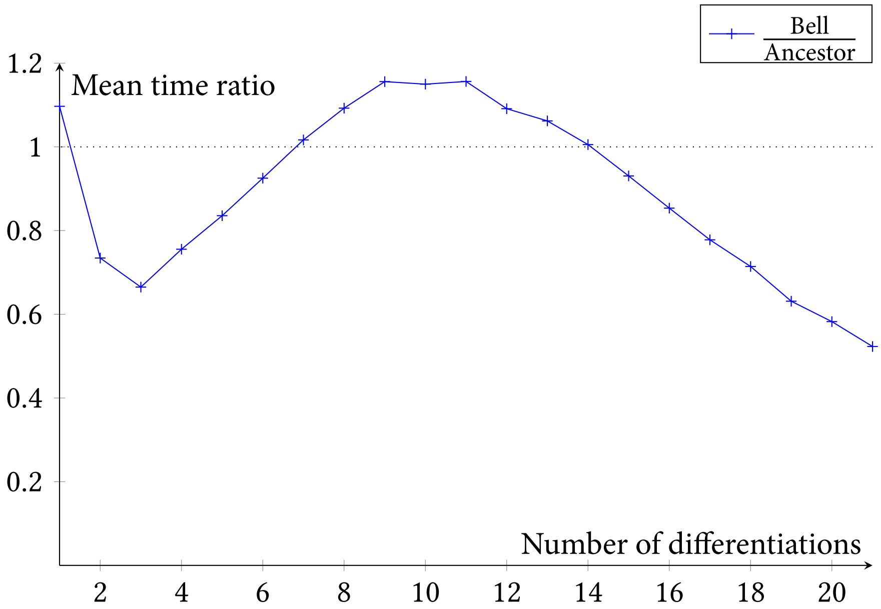


Figure 7.8: Relative performance of using Bell's polynomials recurrence relation vs. the ancestry relation

$\binom{n}{k}$; by contrast, the partition number grows with $2^{\sqrt{n}}$. Overall this means that this asymptotic complexity of computing the first n derivatives of function composition is cubic when using Bell's polynomial and $O\left(2^{\sqrt{n}}\right)$ when using ancestry relations. Finally, the procedure using Bell's polynomial is simpler. For all these reasons, Hydra implements the computation of the n-th derivative of a function composition via the computation of Bell's polynomials.

The drawback of using Bell's recurrence relation is that it is only valid for monovariate functions, as is Faà di Bruno's formula in general. Currently, Hydra has no built-in signal functions that are multivariate, apart from the usual binary operators for which other formulas exist. Furthermore, as signal functions are not first-class in Hydra, the lack of a multivariate composition is not a pressing problem. However, it would be very desirable for them to be in the future, as it would open the way to integrating more aspects of FRP into FHM, and it would be frustrating if the modular compilation technique broke down because of the absence of support for the composition of multivariate functions. Fortunately, some multivariate extensions to Faà di Bruno's formula have been proposed [44, 58, 102] which, given that these are also formulated using informations on partitions, could make use of ancestry relations in an efficient implementation.

44. Constantine, 'A Multivariate Faà Di Bruno Formula With Applications'. 1996 — 58. Encinas, 'A Short Proof of the Generalized Faà Di Bruno's Formula'. 2003 — 102. Mishkov, 'Generalization of the Formula of Faa Di Bruno for a Composite Function with a Vector Argument'. 2000

7.3 — Compiling signal relations & the Hydra runtime

Now that equations compile to order-parametric code, the main obstacle for the modular compilation of Hydra is out of the way. In this section, the compilation of the rest of what makes up Hydra's signal relations is presented. Recall that signal relations are first-class objects in Hydra, therefore it only makes sense that their representation in the target language be first-class objects. This allows for the functional level of the language to be implemented as a regular functional language.

The way a signal relation is used for simulation, and by all the parts of the simulator, dictates how it is should be represented. §6.4 (figure 6.8 in particular) showed exactly what steps a simulator took to simulate a DAE.

The simulator Hydra uses is implemented by the Hydra runtime. It is in charge of structural analysis, such as index-reduction and state selection, memory management, handling of mode changes and the communication between the solver and the user. The runtime consists of a small C program which is linked to a compiled Hydra signal relation. Some of its limitations have been described in §5.5.6. The solvers are provided by the Sundials solver suite [74]. Sundials is an industrial-strength suite of solvers for numerous types of problems. Hydra makes use of KINSOL, which is an algebraic solver and

74. Hindmarsh et al., 'SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers'. 2005

ida which is a dae solver. The former is used for the computation of initial conditions and the latter for the simulation of the dae.

The organisation in steps of the simulation lends itself well for representing Hydra signal relations as a record of functions: each function encodes the behaviour needed for one step of the simulation process. Thus, there is one function that compute the initial state of the signal relation, one which computes the signature matrix (based on the current state), one which computes the residual of the equations appearing in the signal function (based on the state and the informations from index-reduction), etc. In addition, signal relations capture any of the external expressions that appear in their body, similar to how closures are represented in functional languages.

The state of a signal relation is represented in the way outlined in §6.4. That is, the state of a switch is a triple indicating the current mode (including the value of the mode arguments), whether the switch has just become active and the state of the signal relations in the active branch. Because signal relation applications are not inlined, a signal relation application also has a state associated with it in the state of the user of that relation. Note that the state is external to the signal relation, making signal relations immutable objects (even after the simulation starts). An alternative could have been to treat signal relations as objects and have their state be a mutable attribute. Treating the state as separate makes compilation easier and makes the compilation of the functional host a little bit simpler, since signal relations do not require any special treatment, such as instantiation.

The following few sections discuss other problems that arise due to the modular compilation used for Hydra. §7.3.1 will look into the way Hydra variables are represented in compiled code, in such a way that all functions of a signal relation and the runtime agree on where to find informations about a given variable. Next, §7.3.2 deals with the way memory required for order-parametric differentiation is managed during the simulation. Finally, §7.3.3 shows how code can be generated to compute the Jacobian matrix associated with the equation system from order-parametric code directly.

7.3.1 — *Mapping Hydra variables to solver variables*

This section discusses the way in which variables are represented in the compiled code (and handled by the runtime system). The number of variables manipulated by a signal relation is not fixed due to switching. Furthermore, while in a more traditional programming language, local variables are truly local, in the sense that they exist only within the scope in which they're declared; in Hydra the numerical solver must be aware of all the signals present in a signal relation, especially since it is the very thing that computes their value. In non-modular implementations, this is not a problem: all the variables can be gathered while the dae is being assembled and given a specific index. But in this implementation, there must be a way to know, from the compiled code within a signal relation, where to find the value of a given signal (local or not).

The solver passes, to the residual function, an array with the values it guessed for each signal at a given point in time. Let's ignore for now the case of differentiated variables and index-reduction and assume that signals are only represented by their zero-order derivative. The problem is therefore to map a HYDRA name to an index in the input array. The solution used in HYDRA is reminiscent of the stack, used in traditional languages to store local variables. When a new variable is introduced, its location is at the top of the stack and it stays there (with eventually more variables being pushed over) until the scope is exited, at which point it is removed from the stack. This works perfectly thanks to scopes being nested: variables being pushed over a given variable are guaranteed to have been popped before this variable needs to be popped. HYDRA signal relations share this structure and therefore a similar idea can be used except that the index used for a variable can never be reused: once the index has been chosen, all variables introduced after it must have a larger index than it, including after the scope of the variable has been left. Let's illustrate this on the following piece of HYDRA core:

```
let a, b {
  let c {
    a + b + c = 0
  };
  let d {
    d = 1;
    b + d = 0
  };
  a, b <> r;
}
```

Suppose the next free index when entering this signal relation is index k. Then index k can be attributed to `a` and index $k+1$ to `b`. Then when the next **`let`**-block is encountered, `c` gets index $k+2$. Unlike with a stack-based approach, the next free index doesn't come back to $k+1$ when `c` goes out of scope, therefore `d` gets index $k+3$. That scheme works completely straightforwardly with signal relations applications: the indices for the interface variables `a` and `b`, as well as the current value of the next index $k+4$ are passed in to the relevant method of `r`, which returns after it has executed the new value of the next available index.

This scheme is easy and cheap to maintain at runtime: it allows for a common scheme to be shared between all the methods a signal relation compiles to, one simply has to maintain this index. Because the index is maintained at runtime, the index of a variable may be dependant on the mode the system is in. However once a mode has been chosen, the index stays the same. The presence of differentiated variables makes the situation a little more complicated. The variable number is still computed in the same way, however to retrieve the value of its position in the array of guesses provided by the solver is a little more complicated. Assuming the array contains the variables in the same order as the one that was decided on and with their derivatives next to each other. If x is the n-th variable, then the value of $x^{(0)}$ will be after all the derivatives of the first variable (included the 0-order derivative), then all the derivatives of the second, etc. and after all the derivatives of variable $n - 1$. Its index in the array will therefore be at index $\mathrm{D}[0] + 1 + \mathrm{D}[1] + 1 + \cdots + \mathrm{D}[n - 1] + 1$ and its k-th derivative will be k slots after that. Naturally, having to compute this sum every time one wants to reference to a value of a derivative of x is inefficient, so that index can be tracked in exactly the same way as the variable number for x, only taking into account the number of differentiations required. Note that not all methods require to compute this index, most notably the method which computes structural information on the signal relation, used for index-reduction and for which D is undefined. An alternative technique, which doesn't require keeping track of these informations, is to have the runtime transform the input from an array of reals, where all the derivatives are next to each other, to an array of pointers, the pointer at index i pointing to a block of memory of size $\mathrm{D}[i] + 1$ containing the derivatives of variable i. This doesn't require copying any values: the pointers can be made to point into the original input array. Still, it requires a fair amount of extra work and simply keeping track of the index dynamically is likely to be a lot cheaper.

7.3.2 — *Memory management during simulation*

Order-parametric differentiation relies on memory allocation in order to memoise intermediate result. Unlike with first-order automatic differentiation, the size of the allocation is not constant and depends on the number of differentiations required. Since this is not known until runtime, this memory must be dynamically allocated. However dynamically allocating memory on the heap during simulation is not ideal for performance. Of course, when the index is small, the memory could be allocated on the stack which is fast enough although very large systems with high-index could cause issues. There is however a better alternative.

The required space only depends on the number of differentiations needed to compute the first n derivatives of an expression. This number is known as soon as index-reduction has been performed and thus the memory needed can be computed (and allocated) by the runtime just after index-reduction. To compute the required memory, it is easy to

Figure 7.9: Definition of the first-order differential of an iir expression (part 1)

$$\begin{array}{rl}
\mathrm{D}_{x,k}(\Gamma, \mathrm{Diff}(x, p)) & = \left| \begin{array}{l} \textbf{if } k = p \textbf{ then} \\ \quad 1 \\ \textbf{else} \\ \quad 0 \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, \mathrm{Diff}(y, p)) & = \left| \; 0 \right. \\
\mathrm{D}_{x,k}(\Gamma, i : \tau) & = \left| \; \Gamma[i] \right. \\
\mathrm{D}_{x,k}(\Gamma, c) & = \left| \; 0 \right. \\
\mathrm{D}_{x,k}(\Gamma, p = q) & = \left| \; 0 \right. \\
\mathrm{D}_{x,k}(\Gamma, p \;\textbf{mod}\; q) & = \left| \; 0 \right. \\
\mathrm{D}_{x,k}(\Gamma, p + q) & = \left| \begin{array}{l} \textbf{let } p' := \mathrm{D}_{x,k}(\Gamma, p) \textbf{ in} \\ \textbf{let } q' := \mathrm{D}_{x,k}(\Gamma, q) \textbf{ in} \\ p' + q' \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, \sin(p)) & = \left| \begin{array}{l} \textbf{let } p' := \mathrm{D}_{x,k}(\Gamma, p) \textbf{ in} \\ -p' * \cos(p) \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, z[p]) & = \left| \begin{array}{l} \textbf{let } z' := \mathrm{D}_{x,k}(\Gamma, z) \textbf{ in} \\ z'[p] \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, !y) & = \left| \begin{array}{l} \textbf{let } y' := \mathrm{D}_{x,k}(\Gamma, y) \textbf{ in} \\ !y' \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, p \,;\, q) & = \left| \begin{array}{l} \mathrm{D}_{x,k}(\Gamma)(p) \\ \mathrm{D}_{x,k}(\Gamma)(q) \end{array}\right. \\
\mathrm{D}_{x,k}(\Gamma, \textbf{let } i := p \textbf{ in } q) & = \left| \begin{array}{l} \textbf{let } i := p \textbf{ in} \\ \textbf{let } i' := \mathrm{D}_{x,k}(\Gamma, p) \textbf{ in} \\ \mathrm{D}_{x,k}(\Gamma[i \mapsto i'])(q) \end{array}\right.
\end{array}$$

extract, from an iir program, another iir program that computes how much memory that program needs. This is what the Hydra compiler does after generating the iir code for an equation. At runtime, the runtime system then allocates a block of memory large enough and passes it to the generated code. Allocations in iir are then transformed into offsets in that memory buffer. This should be as fast as stack allocation, only amounting to a pointer bump when the code for a given equation runs, but without any risk of overflow since the memory is allocated on the heap.

Note that there is no need to allocate memory for every equation: since equations are not executed in parallel, enough memory to satisfy the largest memory request is enough and the buffer is 'reset' as soon as the code for one equation has finished executing.

7.3.3 — *Computing Jacobians: automatic differentiation for* iir

§5.2.2 showed how the Jacobian matrix was used by the dae solver to compute a solution, but providing a Jacobian to the numerical solver is not strictly an obligation. For instance, Sundials [74] can approximate the Jacobian around a point using numerical differentiation on the residual. However, doing so may lead to poor performance, since the solver simply uses repeated evaluations of the residual and doesn't

[74] Hindmarsh et al., 'SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers'. 2005

Figure 7.10: Definition of the first-order differential of an IIR expression (part 2)

```
$\mathrm{D}_{x,k}(\Gamma, \textbf{allocate}\ \tau[n])$ =
    allocate $\tau\ [n]$

$\mathrm{D}_{x,k}(\Gamma, !y)$ =
    let $y' := \mathrm{D}_{x,k}(\Gamma, y)$ in
    $!y'$

$\mathrm{D}_{x,k}(\Gamma, i \leftarrow p)$ =
    let $p' := \mathrm{D}_{x,k}(\Gamma, p)$ in
    $i \leftarrow p$
    $i' \leftarrow p'$

$\mathrm{D}_{x,k}(\Gamma, \textbf{if}\ c\ \textbf{then}\ p\ \textbf{else}\ q)$ =
    if $\mathrm{c}$ then
        $\mathrm{D}_{x,k}(\Gamma, p)$
    else
        $\mathrm{D}_{x,k}(\Gamma, q)$

$\mathrm{D}_{x,k}(\Gamma, \textbf{sum}\ i\ \textbf{from}\ m\ \textbf{to}\ n\ p)$ =
    sum $i$ from $m$ to $n$
        $\mathrm{D}_{x,k}(\Gamma[i \mapsto 0])(p)$
    end

$\mathrm{D}_{x,k}(\Gamma, \textbf{prod}\ i\ \textbf{from}\ m\ \textbf{to}\ n\ \{p\})$ =
    let $ps$ := allocate $\mathbb{R}\ [m-n]$ in
    let $ps'$ := allocate $\mathbb{R}\ [m-n]$ in
    iter $i$ from $m$ to $n$
        $ps[i] \leftarrow p$
        $ps'[i] \leftarrow \mathrm{D}_{x,k}(\Gamma[i \mapsto 0], p)$
    end
    sum $i$ from $m$ to $n$
        prod $j$ from $m$ to $n$
            if $i = j$ then
                $!ps'[i]$
            else
                $!ps[i]$
                $p$
        end
    end

$\mathrm{D}_{x,k}(\Gamma, \textbf{iter}\ i\ \textbf{from}\ m\ \textbf{to}\ n\ \{p\})$ =
    iter $i$ from $m$ to $n$
        $\mathrm{D}_{x,k}(\Gamma[i \mapsto 0])(p)$
    end
```

exploit the sparse structure of the Jacobian. Indeed, since few variables appear in any given equation, the Jacobian mostly consists of zeroes but, without structural information at its disposal, the solver will waste lots of time and space rediscovering that fact.

Note that not providing a Jacobian is mainly a problem for performance and not so much correctness of the simulation. Indeed, the Jacobian is only used as a means to compute the solution; if it is imprecise, then it might make finding a solution more difficult but if a solution is found, it will be correct.

To generate a Jacobian, one could try to find symbolic expressions for the partial derivatives of the n-th temporal derivative of an expression and translate it to imperative code in a way similar to what was done with residual. A much simpler alternative however is to define a first-order automatic differentiation scheme on the generated IIR code.

Figure 7.9 and 7.10 define a function $D_{x,k}$ which, given a mapping from identifiers to IIR expressions and an IIR expression of type τ, outputs an IIR expression of type τ which corresponds to the partial derivative of the expression with respect to $x^{(k)}$. Note that k in this case may be an arbitrary IIR atom, not necessarily a constant. This is necessary since the IIR code refers to arbitrary derivatives of signals. The environment Γ passed to $D_{x,k}$ maps an identifier bound in the original IIR expression to an IIR atom (either a constant or another identifier) representing its derivative with respect to $x^{(k)}$.

Note that it is never possible for a conditional or the bound of a loop to depend on the value of a signal. If it were possible, then defining the derivative of the following expression with respect to $x^{(0)}$ would be difficult:

```
if Diff$(x, 0) = 0$ then
    $0$
else
    $x$
```

This expression is equivalent to the expression that returns x unchanged. However, applying the definition of $D_{x,0}$ would give 0 when $x = 0$ and 1 everywhere else. This problem is discussed at great length in the automatic differentiation and differentiable programming literature [1] but fortunately doesn't apply to IIR programs, since it only exists when the condition depends in some way on the variable being differentiated against.

The scheme used here is somewhat simplistic and will perform some redundant work in cases. For instance, differentiating the following program:

```
let $x$ :=
    sum $i$ from $0$ to $n$
        $y$
    end
in
$z$
```

will yield:

```
let $x$ :=
    sum $i$ from $0$ to $n$
        $y$
    end
in
let $x'$ :=
    sum $i$ from $0$ to $n$
        $y'$
    end
```

[1] Abadi et al., 'A Simple Differentiable Programming Language'. 2019

in
z'

which performs two loops while it should be possible for it to perform one. Furthermore, if y performs assignments, these assignments will be performed again in y' without, however, changing the meaning of the program. Finally, it differentiates the program many time, once for each signal appearing in the equation. A better implementation could directly generate a matrix for all the variables appearing in the equation. This could be done by following the work presented in [137] for instance, which defines a differentiable array based programming language. Another solution would be to rely on an external system that could perform automatic differentiation at a lower level, such as Clad [148], which performs automatic differentiation of llvm programs or Tapenade [72], which works on Fortran or C programs.

7.4 — Performance evaluation

In this section, benchmarks are presented that compare the runtime of computing the residual of various equations and their derivatives, either using an explicit representation (obtained by applying first-order automatic differentiation repeatedly) or an implicit representation, as presented in section 7.2. Overall, the benchmarks show that using an implicit representation results in performance ranging from on-par with the explicit form to considerably worse. However, some leads on improving the situation in the most problematic cases will be presented in §8.3.1. Also, recall that this scheme may trade some performance during the simulation, but enables modular compilation, making it attractive as an alternative to jit compilation. Further, nothing rules out using a combination of approaches, leveraging their respective advantages: this approach should thus be seen as complementary to existing approaches, with its own distinct characteristics profile, not necessarily as an alternative.

The benchmarks were obtained by generating llvm [89] code from the Hydra compiler[a]. The resulting code was then compiled with clang on the O3 optimisation level and benchmarked using Google's benchmark library [68]. The benchmarks were run on a PC with a 4-core Intel i3-7100T @ 3.4 GHz and 8 GB of RAM. Results are presented in figures 7.11, 7.12, 7.13, 7.14 and 7.15. On the graph, labelled 'Implicit' is the curve giving the runtime for the code generated in implicit form, meaning it can compute any n derivatives. The curve labelled 'Explicit' corresponds to the runtime for code that has been specifically generated by using repeated application of first-order automatic differentiation, as presented in section 7.2.1. The 'Slowdown' curve corresponds to the ratio of the time taken by the implicit form over the time taken by the explicit form, it should be read on the second y-axis.

137. Shaikhha et al., 'Efficient Differentiable Programming in a Functional Array-Processing Language'. 2019

148. Vassilev et al., 'Clad – Automatic Differentiation Using Clang and LLVM'. 2016

72. Hascoet et al., 'The Tapenade Automatic Differentiation Tool'. 2013

89. Lattner et al., 'LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation'. 2004

[a] The resulting LLVM code as well as the benchmarking code can be found at `https://gitlab.com/chupin/hydra-v2/-/tree/separate_compilation/examples/benchmark`

68. Google, *Benchmark*. 2021

The derivative of a product (figure 7.12), when expressed using Leibniz's rule offers performance characteristics that are very close to the explicit code. For high-order differentiation, it even runs faster. The reason for that behaviour is unclear to us: one possibility is that the code becomes so large that it is detrimental to performance, due to cache effects; another possibility is that the compiler fails to perform some optimisation (possibly due to the strange arithmetic of floating-point numbers) which results in more operations overall.

The implementation of Faà di Bruno's formula doesn't seem to benefit from such an effect. This can viewed on the benchmarks in figure 7.11, for $\exp(x)$ and figure 7.14 for x^2. The case of exponential is a particularly good example of one problem with our approach. Since exponential is its own derivative, when the explicit code is generated, it creates many opportunities to use already computed results. These opportunities can easily be identified and exploited by the backend. Exploiting these opportunities in implicit form is much harder for the backend and that is the reason of this large gap in performance on that particular benchmark. By contrast, the benchmark for x^2 shows that, although the implicit form is slightly slower, the gap remains tolerable, especially as the number of differentiations rise. For x^2, whose derivatives do not repeat in the same way, there are indeed a lot less opportunities for the backend to generate better quality code.

The benchmarks in figure 7.13 and 7.15 show that the computation of the successive derivatives of division is much slower in implicit form compared to the explicit form. This is due to the fact that in the implicit, form, the n-th derivative of $\frac{x}{y}$ is computed as the n-th derivative of xy^{-1}. In the explicit form, the usual formula $\frac{x'y-xy'}{x^2}$ is used instead. Although the two formulation are equivalent mathematically, from a computer's perspective they are not, as one generates calls to an exponentiation function while the other simply uses divisions. Analysis of the benchmarks using Linux's perf tool show that, indeed, calls to exponentiation is a significant part of the time spent computing the successive derivatives of the quotient. Note however, that the absolute runtime is not much worse than the runtime for computing the derivatives of x^2 (figure 7.14), which is computed in a very similar way. This seems to confirm our interpretation that the difference in runtimes can be explained by the fact that the explicit case simply uses a better formula. This is of course problematic, not only for expressions in which divisions appear, but also expressions involving functions whose first-order derivative involves a division, like the inverse trigonometric and hyperbolic functions (such as asin, see figure 7.15). However, alternative ways of computing the n-th derivative of the division that do not rely on exponentiation could provide a way to improve this situation [138].

On some benchmarks, for instance in the benchmark for x^2 (fig-

[138] Shieh et al., 'A General Formula for the Nth Derivative of 1/f(x)'. 1967

ure 7.14), the performance gap between the implicit and explicit forms is larger for a smaller number of differentiations. This could be mitigated by using a hybrid approach, where code is generated in explicit form for a small number of differentiations and then falls back to using order-parametric differentiation for larger numbers, where the gap is smaller.

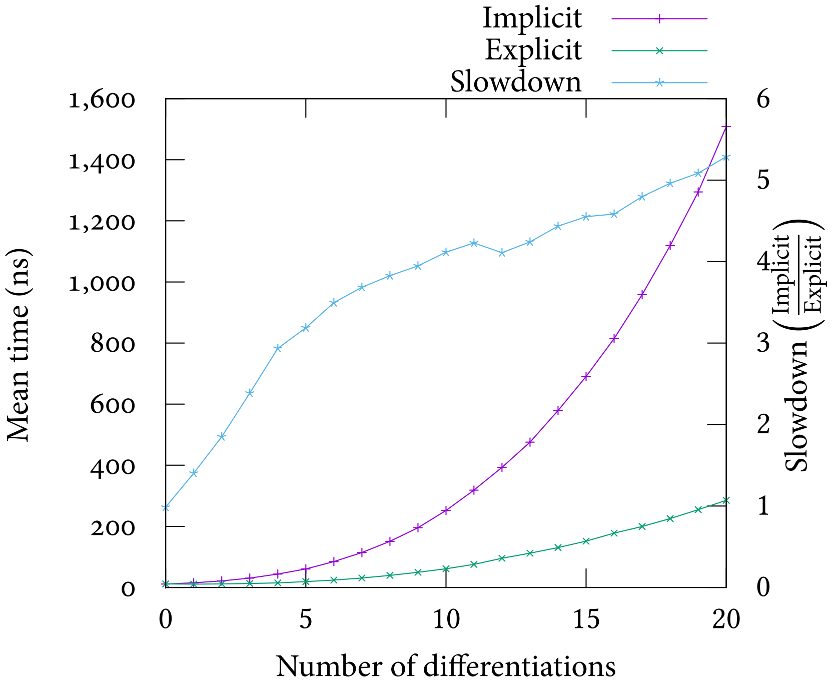


Figure 7.11: Benchmarks results for the computation of the first n-th temporal derivatives of $\exp(x)$

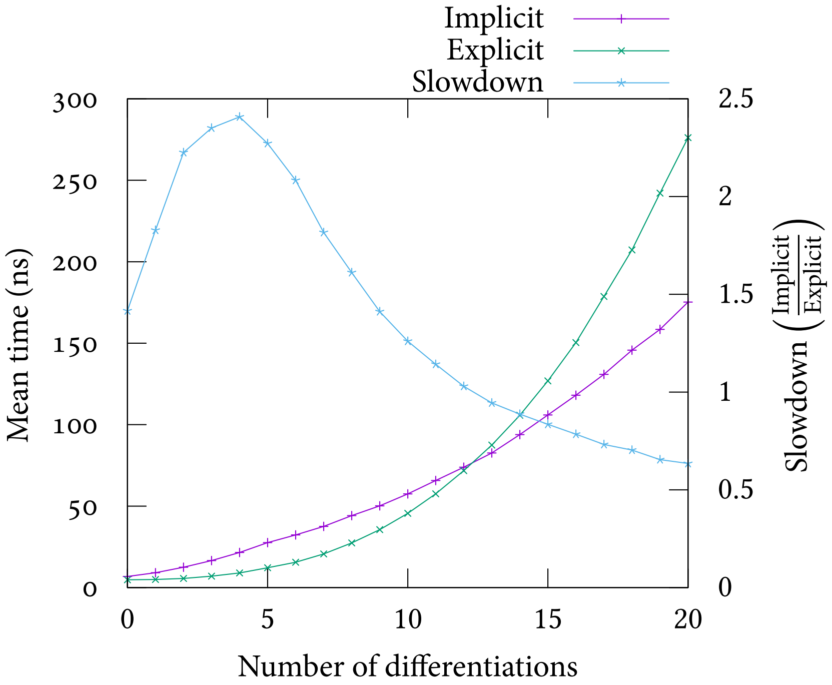


Figure 7.12: Benchmarks results for the computation of the first n-th temporal derivatives of $x * y$

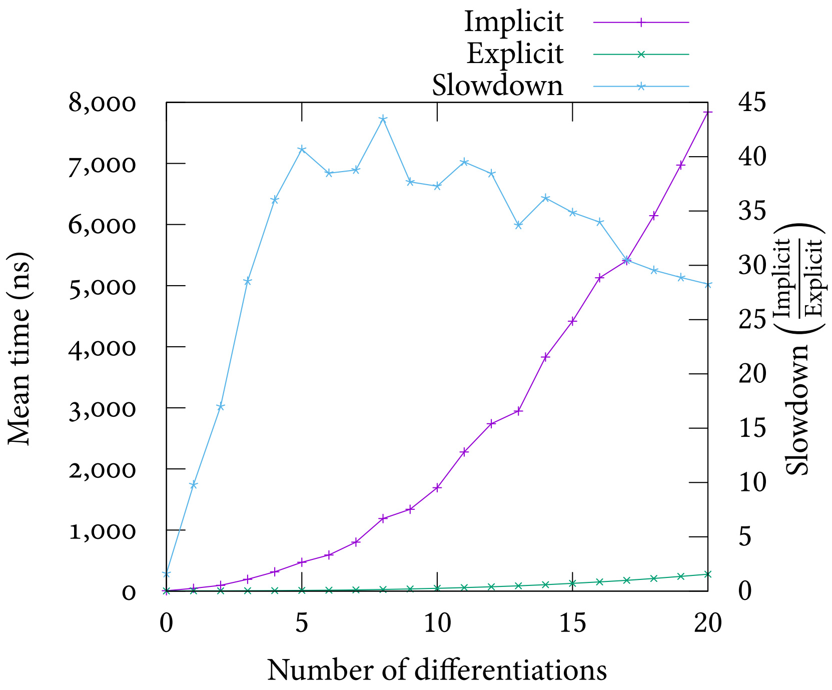


Figure 7.13: Benchmarks results for the computation of the first n-th temporal derivatives of $\frac{x}{y}$

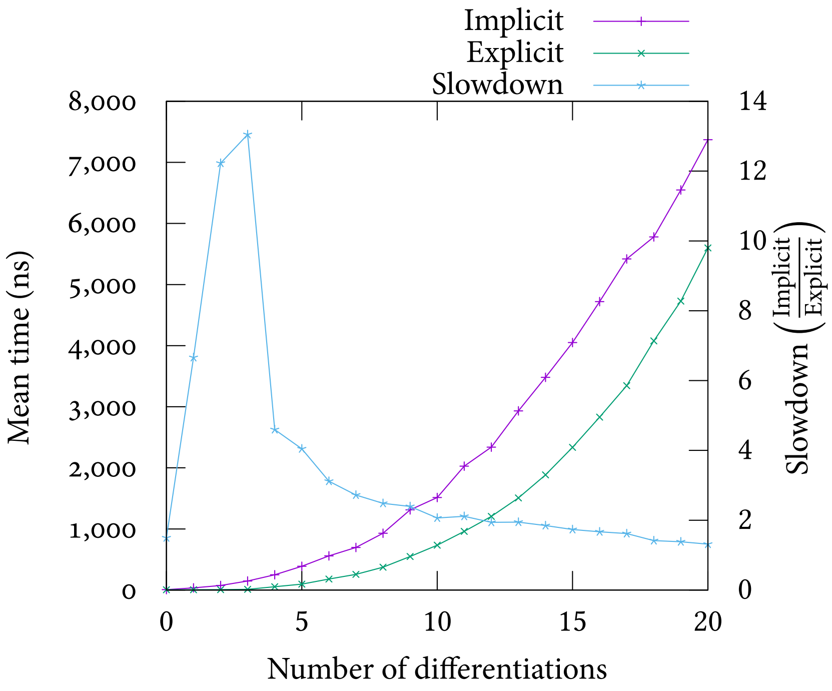


Figure 7.14: Benchmarks results for the computation of the first n-th temporal derivatives of x^2

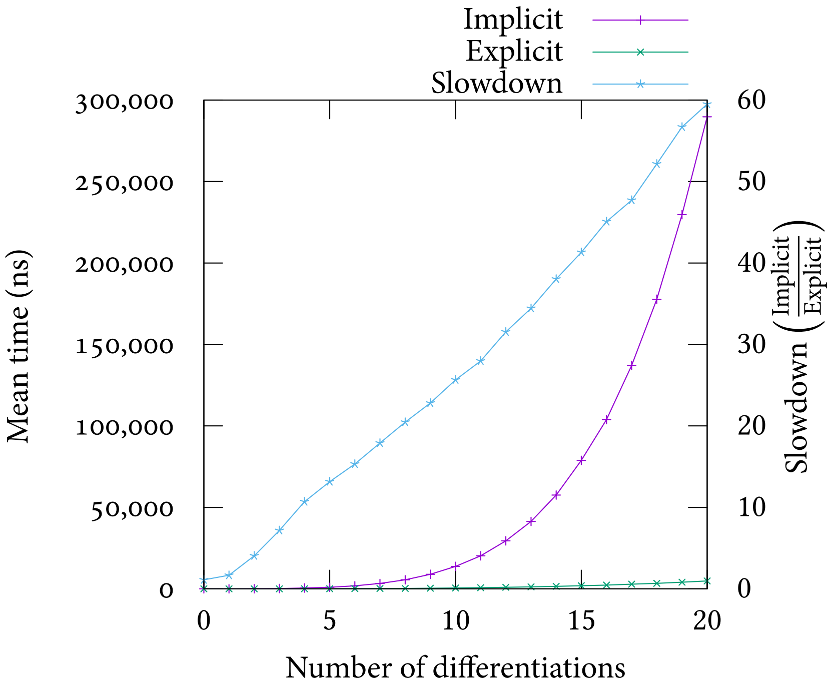


Figure 7.15: Benchmarks results for the computation of the first n-th temporal derivatives of asin(x)

Related works & conclusions 8

8.1 — The original Hydra

The work presented in this thesis builds on the research done for the previous implementation of Hydra [63], to which the term 'original Hydra' refers to. Even though Hydra as a language was first proposed in [112], it did not have an implementation at the time, making a comparison less interesting.

The original Hydra was implemented as an EDSL in Haskell in a way very similar to the embedding of SFRP described in part I. Signal relations were represented using a deep-embedding as a GADT and a quasi-quoter was provided for expressing signal relations in a convenient syntax. To perform simulation, efficient code was extracted using JIT compilation to LLVM and then to machine code [66].

The main difference between the new Hydra and the original one is the choice of embedding, which had important consequences on the implementation. Embedding in an existing language like the original Hydra (or SFRP) saves a considerable amount of effort to the implementor: implementing a modelling language is difficult enough without having to also implement an efficient general purpose language. It is also more appealing for users who can reuse existing libraries of the host language and include the program written in the DSL as part of a larger program more easily. This additional implementation difficulty is the reason for some missing features of this new Hydra, such as the lack of support for unbounded structural dynamism which become more complicated to implement [a]. This also makes the evaluation of the performance of the implementation more difficult, since it becomes important to avoid benchmarking the performance of the host language.

However, controlling the functional host does open much more freedom in the implementation of novel compilation techniques and static analysis. While the original Hydra used JIT compilation, this new version features a fully modular compilation scheme which forgoes the need to incorporate a JIT compiler into the runtime and moves that runtime cost to compile time. Although it would be possible to implement modular compilation in a language like the original Hydra [b], it would have been a lot harder to conceptualize and develop it without having full control over the host language. While this work did not discuss possible static analysis on models, it would be a very interesting avenue for future work. Here again, it would be easier to develop such analysis, in particular modular analysis, while having con-

63. Giorgidze, 'First Class Models'. 2012

112. Nilsson et al., 'Functional Hybrid Modeling'. 2003

66. Giorgidze et al., 'Mixed-Level Embedding and JIT Compilation for an Iteratively Staged DSL'. 2011

[a] See §8.3.4 for a discussion on this particular subject.

[b] See §8.3.5 for a discussion on that specific subject.

trol over the host language. While Haskell's type system can be used to enforce some interesting invariants (as was demonstrated in part I), it could become limiting or unergonomic to implement more complex analysis that would be better implemented as part of a separate dedicated compiler. This choice was made in [11, 13] for example.

11. Benveniste et al., 'Multi-Mode DAE Models - Challenges, Theory and Implementation'. 2019 — 13. Benveniste et al., 'Structural Analysis of Multi-Mode DAE Systems'. 2017

To a user, if one omits the differences in host languages and the missing features mentioned earlier, the differences between the original Hydra and the new Hydra lie only 'at the surface'. That is in the syntax and in differences in implementations but the semantics and the general philosophy of the languages remain the same.

8.2 — Equation-based object-oriented languages

Many non-causal simulation languages are designated by the term *Equation-based object-oriented (eoo)* languages. These languages take another approach to combining models than that of fhm. In these languages, models are objects, in many ways similar to objects in object-oriented languages. While an object is usually a collection of attributes and methods, objects in eoo languages are collections of attributes and equations, which relate the values of the attributes together. Mechanisms inherited from object-oriented languages, such as inheritance, are used to extend models. eoo languages also feature connection equations, which are used to specify equality relations between different components of a model. By contrast, fhm makes use of higher-order relations and signal relation applications both for model extensibility and specifying connections.

The eoo paradigm is largely dominant in the field of non-causal modelling languages. This section gives a few examples of languages designed following these principles, as well as work done on their modular compilation. This has been mainly the case in the context of the Modelica language [62].

62. Fritzson et al., 'Modelica — A Unified Object-Oriented Language for System Modeling and Simulation'. 1998

8.2.1 — *Modelica*

Modelica [62] is a standardised non-causal modelling language and the main representative of eoo languages. It has been used industrially in a large range of applications [40, 86, 95, 134, 156].

62. Fritzson et al., 'Modelica — A Unified Object-Oriented Language for System Modeling and Simulation'. 1998

40. Chrisofakis, 'Simulation-Based Development of Automotive Control Software with Modelica'. 2011 — 86. Konstantinopoulos, 'Dynamic System Modeling and Stability Assessment of an Aircraft Distribution Power System Using Modelica and FMI'. 2020 — 95. Matejak, 'Free Modelica Library for Chemical and Electrochemical Processes'. 2015 — 134. Schmidt, 'Magnetic Force from Experiment, Equation- and Geometry-Based Calculation Using the Example of a Switching Magnet'. 2019 — 156. Zimmer, 'Robust Object-Oriented Formulation of Directed Thermofluid Stream Networks'. 2020

In Modelica, a model is specified as a class with attributes and equations (instead of methods). Below is a simple model, that relates two time-varying quantities `a` and `b` and makes use of two constants, `c` and `d`. `c` is a parameter, it is constant during the simulation but can be set to different values when the model is instantiated; while `d`'s value is set at the time the model is written.

```
model Foo
  Real a;
  Real b;
  parameter Real c;
  constant Real d = 10.0;
equation
  a * c = der(b);
  c * d = b * der(a);
end Foo;
```

To use this model as part of a larger model, it is possible to add instances of it as attributes to other models, like so:

```
model Bar
  Foo f1(c = 1.0);
  Foo f2(c = 2.0);
equation
  f1.a = f2.b;
  f1.b + f2.a = 0;
end Bar;
```

It is also possible for a model to extend another model through inheritance, much like a class can inherit from another in a traditional object-oriented language. For instance, here:

```
model Baz
  extends Foo;
  Real e;
equation
  e = a;
end Baz;
```

`Baz` extends **`Foo`**: all the attributes of the parent model **`Foo`** are available to **`Baz`** and all its equations also hold. Hydra has none of these features but instead makes extensive use of signal relation application to produce similar models.

With Hydra, connecting several components together was done by using higher-order signal relations and equality constraints, such as `serial` or `parallel` from §5.5.3. Instead, Modelica provides connect equations and connectors. A connector is a record containing variables for which the notion of connection can be made precise. For instance, the following example[a] defines an electrical connector as a pair of an electric potential `v` and an electrical current `i`:

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

Note that `i` is annotated with **`flow`**. Indeed, according to Kirchhoff's laws, connecting two pins physically means that their potentials are

[a] *a.* Taken from the documentation for OpenModelica, see: `https://build.openmodelica.org/Documentation/ModelicaReference.'flow'.html`.

equal but that the current that flow through them sum to zero. This can be formulated in Modelica like so:

```
model OnePin
  Pin p;
end OnePin;

model Connect
  OnePin p1;
  OnePin p2;
equation
  connect (p1.p, p2.p);
end Connect;
```

The connect equation will translate to the following pair of equations:

```
p1.p.v = p2.p.v;
p2.p.i + p1.p.i = 0.0;
```

Similar laws exists in other fields of physics for other physical quantities. While Hydra doesn't strictly need connect equations, it would be interesting to add some support for them, as having to manually state these connections laws can be error-prone. The original design for FHM [112] had plans for supporting connect equations but they were never supported in concrete implementations, and the one developed for this thesis is no exception. Integrating them in is not entirely obvious however due to the need to distinguish between potential and flow variables, which Hydra does not make at all.

Modelica currently has limited support for hybrid and structurally dynamic models. Current implementations rely on the assumption that all symbolic processing (such as index-reduction or causalisation) happens prior to the simulation. Earlier sections showed how this was generally impossible in the presence of structurally dynamic models, in practice this results in models which are accepted by the compiler but fail to simulate.

Several Modelica-like languages have been proposed that offer more expressivity, for instance Sol (discussed in §8.2.5) or Model Composition Language (MCL) (discussed in §8.2.3), with the intent to later integrate the resulting techniques into Modelica implementations. A recent proposal to integrate JIT compilation in an implementation of Modelica [145] to allow for more flexibility at runtime but also to help with compilation times by allowing to delay some code

In [155], Zimmer presents theoretical work towards module-preserving compilation of Modelica models. The goal of the author is, when compiling a large complete model, to find ways to reuse previously generated code segments. The author focuses particularly on the issue of causalisation, which has not been considered in this thesis.

112. Nilsson et al., 'Functional Hybrid Modeling'. 2003

145. Tinnerholm et al., 'Towards Introducing Just-in-Time Compilation in a Modelica Compiler'. 2019

155. Zimmer, 'Module-Preserving Compilation of Modelica Models'. 2009

Central to his idea for reusing code is the notion of a causal entity, which corresponds to a particular use of a submodel with a given causality. If the same causal entity appears several time throughout a larger model, code only needs to be generated once and can then be reused by all entities. This work also contains reflections on how to decide whether to reuse the code for an entity or to regenerate it. The difficulties caused by index-reduction are mentioned but the precise way in which the author's approach is reconciled with these difficulties is left as future work. Although the goal of the author is not ahead of time compilation of partial models, the possibility that this technique could be used to distribute pre-compiled code is mentioned but the details are left for future work. It would be interesting to explore whether one could use this work in conjunction with our approach for handling latent equations to obtain a modular compiler for code in causal form.

8.2.2 — *MKL*

The Modelling Kernel Language (mkl) [24] was proposed as a kernel language for non-causal modelling language. It consists of a simple functional language, similar to the functional host used for Hydra in this part. The specification of models is handled by an effectful extension to the language, similar to how side-effects are stated in ml-style languages. Models are first-class entities, enabling a similar style of modelling to the one promoted by Hydra.

mkl however also has connect statements in the style of Modelica, with a distinction between potential and flow variables. Early research of mkl precisely defined a connection semantics for mkl [23] in terms of the flow λ-calculus, which is used to produce correct equations from connection statements. If Hydra were to be extended with connect statements, it would be natural to base this extension on this work, given the similarities between mkl and Hydra.

Modelyze [25] is an extension of mkl with the aim to allow for the implementation of extensible dsl. The language uses a gradual typing scheme to allow for expressive manipulation of symbolic expressions at runtime, while retaining static typing guarantees when desired. This approach allows Modelyze to define hybrid dae as an extension to the language, and not a built-in feature.

Both mkl and Modelyze are implemented through interpreters written in OCaml.

8.2.3 — *MCL*

The Model Composition Language (mcl) [76] is similar in spirit to mkl, as it is intended as a core non-causal language. It is as expressive as the original Hydra [111], supporting unbounded structural dynamism and higher-order models.

[24] Broman, 'Meta-Languages and Semantics for Equation-Based Modeling and Simulation'. 2010

[23] Broman, *Flow Lambda Calculus for Declarative Physical Connection Semantics.* 2007

[25] Broman et al., 'Gradually Typed Symbolic Expressions'. 2017

[76] Höger, 'Compiling Modelica'. 2018

[111] Nilsson et al., 'Exploiting Structural Dynamism in Functional Hybrid Modelling for Simulation of Ideal Diodes'. 2010

MCL was designed as a tool to investigate the separate translation of Modelica models. The goal is to define a semantics for any model, including partial models, which is independent of the context in which they are used. Models are compiled separately as OCaml files. The approach to the representation of equations is however fairly different from the one pursued in this work. In compiled MCL, signals are represented with OCaml records, the record contains (in addition to the value of the signal), a few functions that define the result of basic operations (addition, multiplication, differentiation, etc.) over the signal. The approach is inspired by [33] and, with regards to differentiation, is equivalent to the one presented by Karczmarczuk [84] that was already briefly discussed.

Earlier work from the author of MCL was however more relevant to the work presented in this thesis [77, 78]. In particular, in [77], the author explores the modular semantics of a non-causal language using automatic higher-order differentiation. The technique for performing automatic differentiation in this work is derived from [83]. The proposed target language is able to compute an arbitrary order time derivative of equations, like with order-parametric differentiation, although the approach also allows for the computation of the partial derivatives of expressions, which are useful in computing the Jacobian matrix used for simulation. A proof of the correction of the translation of arbitrary expression to terms in the target language is provided and the author also provides an implementation of the target language as a Java library. The work differs from ours in our focus to generate machine code from the partial models. The target language in [77] sits at a higher-level, with terms of the language being evaluated in the host language. Regardless, it would be interesting to study whether some of the shortcomings of the current implementation of Hydra can be solved by the approach proposed in this work. Performance is unlikely to be improved however, as the computation of composition and multiplication takes exponential time in the number of differentiations. To mitigate that problem, the author proposes to fallback onto more efficient formulations when they apply, like Faà di Bruno's formula or Leibniz's formula when they apply, which are already at the center of order-parametric differentiation.

33. Carette et al., 'Finally Tagless, Partially Evaluated'. 2009

84. Karczmarczuk, 'Functional Differentiation of Computer Programs'. 2001

77. Höger et al., 'Operational Semantics for a Modular Equation Language'. 2013 — 78. Höger et al., 'Notes on the Separate Compilation of Modelica'. 2010

83. Kalman, 'Doubly Recursive Multivariate Automatic Differentiation'. 2002

8.2.4 — *Modia*

Modia [57] is a non-causal modelling language embedded in Julia [16]. It inherits some features of Modelica, the use of connectors and inheritance through a merging operator.

Modia supports structurally dynamic models in a way similar to Hydra, although they are referred to as *multi-mode DAE* in this context. Models dynamically change by reacting to events. Significant work has been done with Modia with regards to the computation of sensible reset values at mode change. In Hydra, every state variable

57. Elmqvist et al., 'Systems Modeling and Programming in a Unified Environment Based on Julia'. 2016

16. Bezanson et al., 'Julia: A Fresh Approach to Numerical Computing'. 2017

is assumed to be continuous unless it is explicitly reinitialised. When it is, then an explicit equation must be provided to compute the reset value, which can be somewhat inconvenient. In Modia, this is handled automatically by using an approach based on non-standard analysis [11, 13]. The authors argue that, at least when confronted with linear equations, the solution computed by their scheme is the only correct solution. It is not true in the presence of non-linearity, nor is it clear, according to the authors, how correct the computed solution is.

Julia is a jit compiled language: all functions, before being called, are compiled to llvm. This means that any edsl implemented with Julia gets jit compilation for free, including Modia.

8.2.5 — *Sol*

Sol[154] is a language similar to Modelica, intended to explore modelling variable-structure systems in a non-causal setting. Although Sol was interpreted, its implementation featured novel work on some of the algorithms needed for the symbolic processing of the equation systems (e.g. index-reduction) which were optimised for dynamic models, where only some parts of the system changes. It would be interesting to implement these algorithms in the Hydra runtime, instead of the current implementation of the Σ-method, which is not aware of any work realised in previous modes.

8.2.6 — *Scicos*

Scicos is a graphical causal programming environment [30] similar to Simulink. It is only mentioned here for the work in [105, 106], where the authors explored the possibility to integrate pre-compiled partial Modelica models as black-boxes into Scicos causal models. The models the authors considered to be compiled in this way are complete, except for the definition of some signals that are considered inputs to the model and whose value will be provided by the environment in which the model is integrated. Although their work is not concerned with the separate compilation of arbitrary models, the authors note that the technique allows for a form of modular compilation, since two models compiled as black-boxes can be composed in the host environment without the need for recompilation.

8.3 — Limitations and future work

8.3.1 — *Performance improvements*

There are cases where the performance of the code using order-parametric differentiation could be improved. For instance when computing the *n*-th derivative of the quotient. Using a specialised formulation that doesn't rely on exponentiation [138] was already mentioned. Other functions could benefit from similar formulas, like tangent [75], exponential or the power functions [100]. Using these special rules has

11. Benveniste et al., 'Multi-Mode DAE Models - Challenges, Theory and Implementation'. 2019 — 13. Benveniste et al., 'Structural Analysis of Multi-Mode DAE Systems'. 2017

154. Zimmer, 'Equation-Based Modeling of Variable-Structure Systems'. 2010

30. Campbell et al., *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. 2006

105. Nikoukhah et al., 'Extensions to Modelica for Efficient Code Generation and Separate Compilation'. 2007 — 106. Nikoukhah et al., 'Towards a Full Integration of Modelica Models in the Scicos Environment'. 2009

138. Shieh et al., 'A General Formula for the Nth Derivative of 1/f(x)'. 1967

75. Hoffman, 'Derivative Polynomials for Tangent and Secant'. 1995

100. McKiernan, 'On the Nth Derivative of Composite Functions'. 1956

the downside of requiring more work from the compiler implementer, instead of relying on the rule for composition.

In cases where the performance gap between the code using repeated first-order differentiation and the code order-parametric differentiation is larger for a small number of differentiations, one solution could be to generate specialised code for the low-order derivatives using repeated first-order differentiation, and fall back to the implicit formula if higher-order derivatives are required. Deciding when, and up to how many derivatives, to generate code could be done via some heuristic (e.g. by considering the size of the code that would result from generating the code, or by considering whether the implicit form is known to have poor performance) or by user annotations. One should, in general, be careful not to generate too much code, to avoid adverse effects on caching, especially if the code is not necessarily being used.

8.3.2 — *External functions*

User-defined signal functions are not supported by the current implementation. Currently, all functions that operate on signals (`sin`, `exp`, etc.) are built-in operators. This restriction is not due to difficulties that would occur with having to compute the *n*-th derivative of a user-defined function: a simple language mechanism could be provided to allow the user to specify them (e.g. by means of annotations, like in DYMOLA [56]). If the definitions are simple enough, automatic differentiation could also be used.

However, if user-defined were allowed to be multivariate, a suitable way to compose these multivariate functions would have to be found. This would either require the implementation of multivariate variants of Faà di Bruno's formula [44], or using approaches to higher-order automatic differentiation that readily support composition of multivariate functions (e.g. [83], used in [77]).

8.3.3 — *Modular causalisation*

The code generated by the compiler is suitable for simulation with a DAE solver. Non-causal modelling languages typically also perform causalisation, so that the resulting model can be simulated with an ODE solver (eventually with the assistance of a non-linear algebraic solver, in the presence of non-linear algebraic loops). This involves symbolically manipulating the set of undirected equations, so that they appear directed and scheduled.

A partial DAE can be made causal in many ways, depending on the context in which it used. In the presence of structural changes, the causality can even change during the simulation. For these reasons, modular causality, at least in the general case, seems very difficult. A simpler goal could be to generate causal versions of only some models, e.g. models that have only a few ways of being made causal or whose

[56] Elmqvist et al., 'Object-Oriented Modeling of Hybrid Systems'. 1993

[44] Constantine et al., 'A Multivariate Faà Di Bruno Formula With Applications'. 1996

[83] Kalman, 'Doubly Recursive Multivariate Automatic Differentiation'. 2002

[77] Höger, 'Operational Semantics for a Modular Equation Language'. 2013

causality does not change during simulation. These models would then be simulated with an ODE solver and all other models with a DAE solver. Scheduling between these models would then have to be dynamic, in case of structural changes.

8.3.4 — *Unbounded number of modes*

FRP supports unbounded number of modes, as did previous implementations of Hydra. The switch construct present in the language also, technically, supports it, albeit inefficiently, by means of higher-order functions. The reasons for this inefficiency is the same as the one explored in §3.4.3 and a similar solution can likely be used.

8.3.5 — *Re-embedding Hydra*

While the fully standalone version of Hydra has its advantages, particularly for exploring alternative compilation methods like the one presented in this work, it also has major disadvantages. The amount of work required to have an advanced functional language compiler is substantial. As a result, the functional language in which Hydra is embedded is by no means state-of-the-art and lacks in expressivity in some key areas. Furthermore, its compilation is somewhat naïve. This makes doing performance evaluations difficult, since bad performance may be attributed to the functional host. Finally, because the ecosystem of the language is inexistent, it mechanically limits adoption of the language by other modellers.

The scheme presented in this work could be implemented by an embedded version of Hydra, in a fashion similar to that of the original implementation. GHC's meta-programming capabilities are powerful enough to support compile time code generation for another language, as demonstrated by libraries such as inline-C or inline-Java [98, 147].

Re-embedding Hydra into Haskell would be a positive step: it would considerably ease the development process of Hydra, would allow for some level of adoption. It is my opinion that this would compensate the added conceptual difficulty for exploring new compilation techniques and compile-time analysis for hybrid non-causal languages. This last point becoming less and less relevant as GHC gets new type-level features, hopefully even some form of dependent types in the near future [50–52].

8.3.6 — *Optimisations in the context of modularly compiled signal relations*

No study has been done on how compiling models modularly affects the simulation of the resulted code. In general purpose language, not inlining a function affects optimisation, as the function is essentially a black-box. The same applies in a non-causal language. For instance, not inlining a model can prevent propagating equalities between variables. This can cause more variables and equations to be present

[98] Mazzoli, *Inline-c.* 2015 — [147] Tweag I/O, *Inline-Java.* 2021

[50] Eisenberg et al., *Dependent Types in Haskell.* 2016 — [51] Eisenberg et al., *Design for Dependent Types.* 2021 — [52] Eisenberg et al., *Dependent Haskell.* 2021

during the simulation than necessary. The number of equality constraints between interface variables and local variables is already a heuristic used by the Hydra compiler to inline some signal relations.

A simple solution to this problem would be to have a signal relation produce additional metadata about the equation it is made from. This could for instance be done when emitting the signature matrix. This could allow the runtime to dynamically propagate equalities between variables and constants and deactivate these simple equations when possible. Work is currently being done in the runtime and compiler to allow for the detection of such equalities. In addition to optimisations, this would allow for the simulation of systems that *rely* on such dynamic simplification, like the full-wave rectifier model presented in [111].

[111] Nilsson et al., 'Exploiting Structural Dynamism in Functional Hybrid Modelling for Simulation of Ideal Diodes'. 2010

8.4 — Conclusions

This part presented the implementation of a compiler for a hybrid non-causal modelling language based on the principles of Functional Hybrid Modelling. The implementation generates machine code using llvm in a modular fashion, even for partial models, alleviating the problem caused by latent equations by using order-parametric differentiation. This allows generating code capable of computing the value of an arbitrary derivative of an equation. The performance of the proposed scheme has been evaluated and compared with that of a more common scheme that generates code for derivatives on demand. Although the evaluation shows mixed results, the core idea is sound. The cost in additional simulation time is balanced by shorter compilation times, which could yield an overall simulation performance gain in the setting of hybrid languages where jit compilation otherwise would have to be carried out during simulation. Further, order-parametric differentiation could be integrated with other methods, such as conventional automatic differentiation or just-in-time compilation, allowing different techniques to be used in different parts of a model depending on their specific performance characteristics. In other words, order-parametric differentiation is at least an interesting complement to existing implementation techniques for cases where modular compilation is a key concern.

Part III

Conclusions

9 Conclusions

This thesis explored new design and implementation ideas for expressive modelling languages. In the first part, it showed how using ideas from the compilation of synchronous dataflow languages could greatly speed-up the implementation of a Functional Reactive Programming (FRP) library, while retaining the characteristic expressivity of the approach. The resulting library SFRP also allowed for additional constraints, which limit the kinds of models that can be written in meaningful ways: rejecting inconsistent models or for which it is not possible to give a reasonable interpretation. The library lost little in ways of convenience, thanks to the implementation of custom syntax inside GHC by use of quasi-quotation. While SFRP was presented as being focused on modelling applications, its design principles are applicable to other reactive libraries intended for traditional reactive applications. Fundamentally, the efficiency of SFRP resides in the distinction it makes between single signals and collections of signals. Other aspects, such as the distinction between different single signal kinds could be modified (or even removed) without much consequences, as far as efficiency is concerned.

In the second part, the thesis explored possibilities for the modular compilation of a language based on FHM. This enables ahead-of-time compilation, previously only available to less expressive non-causal languages. To achieve this, a new technique was developed called order-parametric differentiation, which allows to generate code able to compute an arbitrary derivative of an expression, thus alleviating problems caused by latent equations for the simulation of systems of undirected equations. The performance characteristics of this technique were studied and compared to that of code generated using traditional first-order automatic differentiation techniques. While the evaluation shows mixed results, there are possible ways to improve the situation. If performance is an issue, this technique can also be used alongside other techniques that produce better code at the cost of some modularity, such as JIT compilation.

Ideas for future research have already been stated at the end of each part separately for FRP implementations and FHM implementations. Let us therefore conclude this work by discussing avenues for possible development at the intersection of FRP and FHM. Indeed, while very close, each language is distinctly in a class of its own. Still, the work presented in this thesis already showed that features common to both approaches, most notably switching and type-level restrictions on models, could be implemented and treated in a similar manner between FRP and FHM. It would be interesting to see if some extensions to FRP could also be implemented in the non-causal setting of FHM. For instance, support for richer classes of switching constructs, such as collection-based switches [110]; or for impulses [108], to allow modellers to state discontinuities without structural changes.

However, the commonalities between the implementations of FRP and FHM presented in this thesis suggest a more ambitious way forward, in the form of a language unifying these two paradigms. A unified language would have some significant advantages over both individual approaches. Indeed, although for modelling applications FHM is generally a more pleasant setting to work in (simply by virtue of it supporting non-causal models), it can be awkward to express some models, for example, those making use of stateful signal functions which, in contrast, are very easily expressed with FRP. A language based on FHM but with FRP-like aspects would result in an even more expressive and flexible modelling language, offering the best of both causal and non-causal worlds to the modeller.

[110] Nilsson et al., 'Functional Reactive Programming, Continued'. 2002

[108] Nilsson, 'Functional Automatic Differentiation with Dirac Impulses'. 2003

A

Proofs of results on the structure of integer partitions

This appendix contains the proofs of the various results presented in section 7.2.3.1. It is unlikely that these results are new or significant outside of the work of this thesis and they are, for the most part, very easy to prove. However, not being able to find these results in the literature, I present their proofs here.

Proof of lemma 7.2.1. Let $p = (m_1, \dots, m_n)$ be a partition of n. For $\left(m_1, \dots, m_j + 1, \dots, m_n\right)$ to be a partition of $n + j$, the following quantity must equal $n + j$:

$$1 \cdot m_1 + \cdots + j \cdot (m_j + 1) + \cdots + n \cdot m_n + (n+1) \cdot 0 + \cdots + (n+j) \cdot 0$$

The above can be rearranged as:

$$\left(1 + m_1 + \cdots + j \cdot m_j + \cdots + n \cdot m_n\right) + j$$

which is indeed equal to $n + j$, given that p is a partition of n. □

Proof of lemma 7.2.2 and lemma 7.2.3. Let $p = (m_1, \dots, m_n)$ be a partition of n which is not the partition $(0, \dots, 0, 1)$.

Since $m_n = 0$, there exist two indices $u, v \in [1, n[$ and such that m_u is the first non-zero coefficient of p and m_v is the last non-zero coefficient of p That is:

$$\begin{aligned} m_u &\neq 0 \\ m_v &\neq 0 \\ \forall k \notin [u, v], m_k &= 0 \end{aligned}$$

p can thus be rewritten as $(0, \dots, m_u, \dots, m_v, \dots, 0)$ and thus the following holds:

$$u \cdot m_u + \cdots + v \cdot m_v = n \tag{A.1}$$

Using this equation one can prove that $v \leq n - u$, which will be useful below. By simple rearranging, one gets:

$$v \cdot m_v = n - u \cdot m_u - (u+1) \cdot m_{u+1} - \cdots - (v-1) \cdot m_{v-1}$$

Since $m_u > 0$, $u \leq u \cdot m_u$ and since all coefficients m_{u+1}, …, m_{v-1} are positive, the term on the right of the equation is bounded by $n - u$, giving:

$$v \cdot m_v \leq n - u$$

Since $m_v > 0$, it follows that $v \leq v \cdot m_v$ and thus $v \leq n - u$.

By subtracting u on both sides of equation A.1, one gets:

$$u \cdot (m_u - 1) + \cdots + v \cdot m_v = n - u$$

Since $v \leq n - u$, this shows that $(0, \dots, m_u - 1, \dots, m_v, \dots, 0)$ is a valid partition of $n - u$ and therefore the u-ancestor of p. It is the closest ancestor of p, since all coefficients of p before m_u are 0, which proves lemma 7.2.3. □

Proof of theorem 7.2.2. Let p be a partition of n of the form whose first j coefficients are 0, with $j \geq 0$, thus $p = \left(0, \dots, 0, m_{j+1}, \dots, m_n\right)$.

Let's consider p_k, the k-child of p, with $1 \leq k \leq j + 1$. By definition of the k-child, the first non-zero coefficient of p_k is its k-th coefficient: either 1 if $k \leq j$ or $m_{j+1} + 1$ if $k = j + 1$. From lemma 7.2.3, it follows that p is its closest ancestor. □

Proof of theorem 7.2.3. From lemma 7.2.2, we know that every partition of n (save for $(0, \dots, 0, 1)$) is the closest child of partition of a predecessor of n. Therefore the construction indeed produces every partition of n. Furthermore, no partition can be the closest child of two partitions, since it would mean it has two closest ancestors which is impossible. Therefore, the construction produce every partition exactly once without duplicates. □

Bibliography

[1] Martín Abadi and Gordon D. Plotkin. 'A Simple Differentiable Programming Language'. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019), 38:1–38:28. DOI: 10.1145/3371106 (cit. on p. 138).

[2] Alfred V. Aho, John E. Hopcroft and Jeffrey Ullman. *Data Structures and Algorithms*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 978-0-201-00023-8 (cit. on p. 56).

[3] Pascalin Amagbégnon, Loïc Besnard and Paul Le Guernic. 'Implementation of the Data-Flow Synchronous Language SIGNAL'. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. PLDI '95. New York, NY, USA: Association for Computing Machinery, June 1995, pp. 163–173. ISBN: 978-0-89791-697-4. DOI: 10.1145/207110.207134 (cit. on p. 5).

[4] Heinrich Apfelmus. *Reactive-Banana: Library for Functional Reactive Programming (FRP)*. http://hackage.haskell.org/package/reactive-banana. 2011 (cit. on pp. 4, 22, 72).

[5] Louis François Antoine Arbogast. *Du calcul des dérivations*. fr. Strasbourg, France: Levrault frères, 1800 (cit. on p. 122).

[6] T. Archibald, Craig Fraser and Ivor Grattan-Guinness. 'The History of Differential Equations, 1670–1950'. In: *Oberwolfach Reports* (Jan. 2004), pp. 2729–2794. DOI: 10.4171/OWR/2004/51 (cit. on p. 1).

[7] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. 1st. USA: Society for Industrial and Applied Mathematics, 1998. ISBN: 978-0-89871-412-8 (cit. on p. 18).

[8] Manuel Bärenz. *Rhine: Functional Reactive Programming with Type-Level Clocks*. http://hackage.haskell.org/package/rhine. 2017 (cit. on p. 22).

[9] E. T. Bell. 'Partition Polynomials'. In: *Annals of Mathematics* 29.1/4 (1927), pp. 38–46. ISSN: 0003-486X. DOI: 10.2307/1967979 (cit. on p. 123).

[10] Albert Benveniste et al. 'A Type-Based Analysis of Causality Loops in Hybrid Systems Modelers'. In: *Journal of Nonlinear Analysis Hybrid Systems* (2017) (cit. on pp. 5, 21, 40, 43, 69).

[11] Albert Benveniste et al. 'Multi-Mode DAE Models - Challenges, Theory and Implementation'. en. In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 283–310. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_16 (cit. on pp. 146, 151).

[12] Albert Benveniste et al. 'Non-Standard Semantics of Hybrid Systems Modelers'. In: *Journal of Computer and System Sciences*. In Commemoration of Amir Pnueli 78.3 (May 2012), pp. 877–910. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2011.08.009 (cit. on p. 69).

[13] Albert Benveniste et al. 'Structural Analysis of Multi-Mode DAE Systems'. In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. HSCC '17. New York, NY, USA: ACM, 2017, pp. 253–263. ISBN: 978-1-4503-4590-3. DOI: `10.1145/3049797.3049806` (cit. on pp. 146, 151).

[14] Gérard Berry. 'Systèmes réactifs logiciels, le design du langage synchrone Esterel v5' (Amphithéâtre Maurice Halbwachs, Collège de France, Paris, France). 16th Apr. 2013. URL: `https://www.college-de-france.fr/site/gerard-berry/course-2013-04-16-10h00.htm` (visited on 18/04/2022) (cit. on p. 4).

[15] Dimitri P. Bertsekas. *Linear Network Optimization: Algorithms and Codes*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 978-0-262-02334-4 (cit. on p. 86).

[16] Jeff Bezanson et al. 'Julia: A Fresh Approach to Numerical Computing'. In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: `10.1137/141000671` (cit. on pp. 3, 150).

[17] Dariusz Biernacki et al. 'Clock-Directed Modular Code Generation for Synchronous Dataflow Languages'. en. In: *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. Tucson, Arizona, June 2008, p. 10 (cit. on pp. 5, 35, 50).

[18] Fischer Black and Myron Scholes. 'The Pricing of Options and Corporate Liabilities'. In: *Journal of Political Economy* 81.3 (1973), pp. 637–654. ISSN: 0022-3808 (cit. on p. 1).

[19] Olivier Bouissou and Alexandre Chapoutot. 'An Operational Semantics for Simulink's Simulation Engine'. In: *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*. LCTES '12. New York, NY, USA: Association for Computing Machinery, June 2012, pp. 129–138. ISBN: 978-1-4503-1212-7. DOI: `10.1145/2248418.2248437` (cit. on p. 69).

[20] Timothy Bourke and Marc Pouzet. 'Zélus, a Synchronous Language with ODEs'. In: *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*. Philadelphia, USA: ACM, 2013 (cit. on pp. 1, 2, 5, 69).

[21] Timothy Bourke et al. 'A Synchronous Look at the Simulink Standard Library'. In: *ACM Transactions on Embedded Computing Systems* 16.5s (Sept. 2017), 176:1–176:24. ISSN: 1539-9087. DOI: `10.1145/3126516` (cit. on p. 69).

[22] K. E. Brenan, S. L. Campbell and L. R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1995. ISBN: 978-0-89871-353-4. DOI: `10.1137/1.9781611971224` (cit. on p. 17).

[23] David Broman. *Flow Lambda Calculus for Declarative Physical Connection Semantics*. en. Tech. rep. 1. Linköping, Sweden: Deparment of Computer and Information Science, Linköping University, Dec. 2007, p. 19 (cit. on p. 149).

[24] David Broman. 'Meta-Languages and Semantics for Equation-Based Modeling and Simulation'. en. PhD Thesis. Linköping, Sweden: Linköpings universitet, 2010 (cit. on pp. 2, 8, 149).

[25] David Broman and Jeremy G. Siek. 'Gradually Typed Symbolic Expressions'. In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '18. New York, NY, USA: Association for Computing Machinery, Dec. 2017, pp. 15–29. ISBN: 978-1-4503-5587-2. DOI: `10.1145/3162068` (cit. on pp. 2, 149).

[26] Peter N. Brown, Alan C Hindmarsh and Linda Ruth Petzold. *DASKR Package: DAE Solver with Krylov Methods and Rootfinding*. June 2011 (cit. on p. 17).

[27] Richard L. Buden and J. Douglas Faires. *Numerical Analysis*. Ninth. Boston, MA: Brooks/Cole, 2011. ISBN: 978-0-538-73351-9 (cit. on p. 118).

[28] Daniel Bünzli. *React, Functional Reactive Programming for OCaml*. https://erratique.ch/talks/react-ocamlum-2010.pdf. Accessed: 2019-05-09. 2010 (cit. on pp. 4, 22).

[29] Benoît Caillaud, Mathias Malandain and Joan Thibault. 'Implicit Structural Analysis of Multimode DAE Systems'. In: *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*. HSCC '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–11. ISBN: 978-1-4503-7018-9. DOI: 10.1145/3365365.3382201 (cit. on pp. 85, 90, 113).

[30] Stephen L. Campbell, Jean-Philippe Chancelier and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. en. New York: Springer-Verlag, 2006. ISBN: 978-0-387-30486-1. DOI: 10.1007/0-387-30486-X (cit. on p. 151).

[31] Stephen L. Campbell and C. William Gear. 'The Index of General Nonlinear DAEs'. en. In: *Numerische Mathematik* 72.2 (Dec. 1995), pp. 173–196. ISSN: 0945-3245. DOI: 10.1007/s002110050165 (cit. on p. 7).

[32] John Capper. 'Semantics Methods for Functional Hybrid Modelling'. en. PhD Thesis. Nottingham, United Kingdom: University of Nottingham, Nov. 2014 (cit. on p. 111).

[33] Jacques Carette, Oleg Kiselyov and Chung-chieh Shan. 'Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages'. In: *Journal of Functional Programming* 19.5 (Sept. 2009), pp. 509–543. ISSN: 0956-7968. DOI: 10.1017/S0956796809007205 (cit. on p. 150).

[34] Francesco Casella. *Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives*. Sept. 2015, p. 468. DOI: 10.3384/ecp15118459 (cit. on p. 113).

[35] Francesco Casella and Bernhard Bachmann. 'On the Choice of Initial Guesses for the Newton-Raphson Algorithm'. en. In: *Applied Mathematics and Computation* 398 (June 2021), p. 125991. ISSN: 0096-3003. DOI: 10.1016/j.amc.2021.125991 (cit. on p. 83).

[36] Paul Caspi et al. 'LUSTRE: A Declarative Language for Programming Synchronous Systems'. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*. Vol. 178. 1987, p. 188 (cit. on pp. 5, 40).

[37] Francois E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0-387-26102-8 (cit. on pp. 1, 14, 17, 35, 82).

[38] Manuel M. T. Chakravarty, Gabriele Keller and Simon Peyton Jones. 'Associated Type Synonyms'. In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. New York, NY, USA: ACM, 2005, pp. 241–253. ISBN: 978-1-59593-064-4. DOI: 10.1145/1086365.1086397 (cit. on pp. 10, 57).

[39] Manuel M. T. Chakravarty et al. 'Associated Types with Class'. In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. New York, NY, USA: ACM, 2005, pp. 1–13. ISBN: 978-1-58113-830-6. DOI: 10.1145/1040305.1040306 (cit. on pp. 10, 57).

[40] Emmanuel Chrisofakis et al. ‘Simulation-Based Development of Automotive Control Software with Modelica’. In: *Proceedings of the 8th Modelica Conference*. Dresden, Germany: Linköping University Electronic Press, 2011. DOI: 10.3384/ECP110631 (cit. on p. 146).

[41] Guerric Chupin and Henrik Nilsson. ‘Functional Reactive Programming, Restated’. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*. PPDP ’19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–14. ISBN: 978-1-4503-7249-7. DOI: 10.1145/3354166.3354172 (cit. on p. 7).

[42] Guerric Chupin and Henrik Nilsson. ‘Modular Compilation for a Hybrid Non-Causal Modelling Language’. en. In: *Electronics* 10.7 (Jan. 2021), p. 814. DOI: 10.3390/electronics10070814 (cit. on p. 10).

[43] Louis Comtet. *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. en. Springer Science & Business Media, Dec. 2012. ISBN: 978-94-010-2196-8 (cit. on p. 128).

[44] G. M. Constantine and T. H. Savits. ‘A Multivariate Faà Di Bruno Formula With Applications’. In: *Transactions of the American Mathematical Society* 348.2 (1996), pp. 503–520. ISSN: 0002-9947 (cit. on pp. 132, 152).

[45] Antony Courtney, Henrik Nilsson and John Peterson. ‘The Yampa Arcade’. In: *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell ’03. Uppsala, Sweden: ACM, 2003, pp. 7–18. ISBN: 1-58113-758-3. DOI: 10.1145/871895.871897 (cit. on pp. 4, 22).

[46] Pascal Cuoq and Marc Pouzet. ‘Modular Causality in a Synchronous Stream Language’. In: *10th European Symposium on Programming (ESOP’01)*. Vol. 2028. Lecture Notes in Computer Science. Genova, Italy: Springer, Apr. 2001, pp. 237–251. DOI: 10.1007/3-540-45309-1_16 (cit. on p. 50).

[47] C. F. Curtiss and J. O. Hirschfelder. ‘Integration of Stiff Equations’. In: *Proceedings of the National Academy of Sciences of the United States of America* 38.3 (1952), pp. 235–243. ISSN: 0027-8424 (cit. on p. 18).

[48] Evan Czaplicki. ‘Elm: Concurrent FRP for Functional GUIs’. Undergraduate Thesis. Harward University, 2012 (cit. on pp. 4, 22).

[49] Dominic Duggan and Frederick Bent. ‘Explaining Type Inference’. en. In: *Science of Computer Programming* 27.1 (July 1996), pp. 37–83. ISSN: 0167-6423. DOI: 10.1016/0167-6423(95)00007-0 (cit. on p. 56).

[50] Richard A. Eisenberg. ‘Dependent Types in Haskell: Theory and Practice’. PhD Thesis. University of Pennsylvania, 2016 (cit. on p. 153).

[51] Richard A. Eisenberg. *Design for Dependent Types*. Jan. 2021. URL: %5Curl%7Bhttps://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0378-dependent-type-design.rst%7D (cit. on p. 153).

[52] Richard A. Eisenberg and GHC Contributors. *Dependent Haskell*. en. https://gitlab.haskell.org/ghc/ghc/-/wikis/dependent-haskell. 2021 (cit. on p. 153).

[53] J. Eker et al. ‘Taming Heterogeneity - the Ptolemy Approach’. In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 127–144. ISSN: 1558-2256. DOI: 10.1109/JPROC.2002.805829 (cit. on pp. 1, 69).

[54] Conal Elliott and Paul Hudak. ‘Functional Reactive Animation’. In: *International Conference on Functional Programming*. Amsterdam, The Netherlands, June 1997 (cit. on pp. 3, 21–23).

[55] Conal M. Elliott. 'Push-Pull Functional Reactive Programming'. en. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell - Haskell '09*. Edinburgh, Scotland: ACM Press, 2009, p. 25. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596643 (cit. on p. 70).

[56] Hilding Elmqvist, François E. Cellier and Martin Otter. 'Object-Oriented Modeling of Hybrid Systems'. en. In: *Proceedings of the European Simulation Symposium*. Delft, The Netherlands, 1993, pp. 31–43 (cit. on pp. 1, 101, 152).

[57] Hilding Elmqvist, Toivo Henningsson and Martin Otter. 'Systems Modeling and Programming in a Unified Environment Based on Julia'. en. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 198–217. ISBN: 978-3-319-47169-3. DOI: 10.1007/978-3-319-47169-3_15 (cit. on pp. 2, 8, 113, 150).

[58] L. Hernández Encinas and J. Muñoz Masqué. 'A Short Proof of the Generalized Faà Di Bruno's Formula'. en. In: *Applied Mathematics Letters* 16.6 (Aug. 2003), pp. 975–979. ISSN: 0893-9659. DOI: 10.1016/S0893-9659(03)90026-7 (cit. on p. 132).

[59] Leonhard Euler. *Institutionum Calculi Integralis*. Latin. Saint Petersburg, Russia: Impensis Academiae Imperialis Scientiarum, 1768. The curious reader can find an English translation of this work at http://www.17centurymaths.com/contents/integralcalculus.html. (Cit. on p. 15).

[60] Francesco Faà Di Bruno. 'Note sur une nouvelle formule de calcul différentiel'. fr. In: *The Quaterly Journal of Pure and Applied Mathematics* 1 (1857), pp. 359–360. An English translation of this work can be found at https://www.mn.uio.no/math/english/people/aca/michaelf/translations/faadibruno_english.pdf. (Cit. on pp. 122, 165).

[61] Francesco Faà di Bruno. 'Sullo sviluppo delle funzioni'. it. In: *Annali di scienze matematiche e fisiche* 6 (1855). This work is the Italian translation of [60]. Refer to this entry for an English translation., pp. 479–480 (cit. on p. 122).

[62] Peter Fritzson and Vadim Engelson. 'Modelica — A Unified Object-Oriented Language for System Modeling and Simulation'. In: *ECOOP'98 — Object-Oriented Programming*. Ed. by Eric Jul. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 67–90. ISBN: 978-3-540-69064-1 (cit. on pp. 1, 146).

[63] George Giorgidze. 'First Class Models: On a Non-Causal Language for Higher-Order and Structurally Dynamic Modelling and Simulation'. en. PhD Thesis. Nottingham, United Kingdom: University of Nottingham, 2012 (cit. on pp. 91, 94, 112, 145).

[64] George Giorgidze and Henrik Nilsson. 'Embedding a Functional Hybrid Modelling Language in Haskell'. In: *Implementation and Application of Functional Languages: 20th International Symposium, IFL 2008, Revised Selected Papers*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Vol. 5836. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 138–155. ISBN: 978-3-642-24451-3. DOI: 978-3-642-24451-3 (cit. on p. 8).

[65] George Giorgidze and Henrik Nilsson. 'Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems'. en. In: *The 7 International Modelica Conference, Como, Italy*. Oct. 2009, pp. 208–218. DOI: 10.3384/ecp09430137 (cit. on pp. 91, 113).

[66] George Giorgidze and Henrik Nilsson. 'Mixed-Level Embedding and JIT Compilation for an Iteratively Staged DSL'. In: *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*. Ed. by Julio Mariño. Vol. 6559. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 48–65 (cit. on pp. 8, 145).

[67] George Giorgidze and Henrik Nilsson. 'Switched-On Yampa'. en. In: *Practical Aspects of Declarative Languages*. Ed. by Paul Hudak and David S. Warren. Vol. 4902. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 282–298. ISBN: 978-3-540-77441-9. DOI: 10.1007/978-3-540-77442-6_19 (cit. on p. 4).

[68] Google. *Benchmark*. Google. Aug. 2021 (cit. on pp. 129, 139).

[69] Alexander Gorban and Gregory Yablonsky. 'Three Waves of Chemical Dynamics'. In: *Mathematical Modelling of Natural Phenomena* 10 (Aug. 2015), pp. 1–5. DOI: 10.1051/mmnp/201510501 (cit. on p. 1).

[70] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Vol. 105. Siam, 2008 (cit. on p. 120).

[71] Godrey H. Hardy and Edward M. Wright. *An Introduction to the Theory of Numbers*. Ed. by Roger Heath-Brown, Joseph Silverman and Andrew Wiles. Sixth Edition. Oxford, New York: Oxford University Press, July 2008. ISBN: 978-0-19-921986-5 (cit. on p. 122).

[72] Laurent Hascoet and Valérie Pascual. 'The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification'. In: *ACM Transactions on Mathematical Software* 39.3 (May 2013), 20:1–20:43. ISSN: 0098-3500. DOI: 10.1145/2450153.2450158 (cit. on p. 139).

[73] Alan C. Hindmarsh, Radu Serban and Aaron M. Collier. *Example Programs for IDA v5.7.0*. en. Tech. rep. UCRL-SM-208112. Lawrence Livermore National Laboratory, Feb. 2021, p. 32 (cit. on p. 1).

[74] Alan C. Hindmarsh et al. 'SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers'. In: *ACM Trans. Math. Softw.* 31.3 (Sept. 2005), pp. 363–396. ISSN: 0098-3500. DOI: 10.1145/1089014.1089020 (cit. on pp. 17, 18, 132, 136).

[75] Michael E. Hoffman. 'Derivative Polynomials for Tangent and Secant'. In: *The American Mathematical Monthly* 102.1 (1995), pp. 23–30. ISSN: 0002-9890. DOI: 10.2307/2974853 (cit. on p. 151).

[76] Christoph Höger. 'Compiling Modelica : About the Separate Translation of Models from Modelica to OCaml and Its Impact on Variable-Structure Modeling'. en. PhD thesis. Berlin, Germany: Technische Universität Berlin, May 2018 (cit. on pp. 2, 149).

[77] Christoph Höger. 'Operational Semantics for a Modular Equation Language'. en. In: *Proceedings of the 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop*. Vancouver, Canada, Dec. 2013, p. 5. DOI: 10.3384/ecp13090002 (cit. on pp. 113, 150, 152).

[78] Christoph Höger, Florian Lorenzen and Peter Pepper. 'Notes on the Separate Compilation of Modelica'. en. In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Oslo, Norway: Linköping University Electronic Press, Sept. 2010. ISBN: 978-91-7519-824-8 (cit. on p. 150).

[79] Paul Hudak et al. 'Arrows, Robots, and Functional Reactive Programming'. In: *Summer School on Advanced Functional Programming 2002, Oxford University*. Vol. 2638. Lecture Notes in Computer Science. Springer-Verlag, 2003, pp. 159–187 (cit. on pp. 4, 22).

[80] John Hughes. 'Generalising Monads to Arrows'. In: *Science of computer programming* 37.1-3 (2000), pp. 67–111 (cit. on pp. 4, 23).

[81] ISO. *ISO C Standard 1999*. Tech. rep. 1999 (cit. on p. 120).

[82] Carl Gustav Jacob Jacobi. 'De formatione et proprietatibus Determinatium'. Latin. In: *Journal für die reine und angewandte Mathematik* 22 (1841), pp. 285–318. DOI: 10.1515/crll.1841.22.285 (cit. on p. 83).

[83] Dan Kalman. 'Doubly Recursive Multivariate Automatic Differentiation'. In: *Mathematics Magazine* 75.3 (2002), pp. 187–202. ISSN: 0025-570X. DOI: 10.2307/3219241 (cit. on pp. 150, 152).

[84] Jerzy Karczmarczuk. 'Functional Differentiation of Computer Programs'. en. In: *Higher-Order and Symbolic Computation* 14.1 (Mar. 2001), pp. 35–57. ISSN: 1573-0557. DOI: 10.1023/A:1011501232197 (cit. on pp. 115, 150).

[85] K. S. Kölbig. 'Programs for Computing the Logarithm of the Gamma Function, and the Digamma Function, for Complex Argument'. en. In: *Computer Physics Communications* 4.2 (Nov. 1972), pp. 221–226. ISSN: 0010-4655. DOI: 10.1016/0010-4655(72)90012-4 (cit. on p. 121).

[86] Stavros Konstantinopoulos, Hamed Nademi and Luigi Vanfretti. 'Dynamic System Modeling and Stability Assessment of an Aircraft Distribution Power System Using Modelica and FMI'. In: *2020 AIAA/IEEE Electric Aircraft Technologies Symposium (EATS)*. Aug. 2020, pp. 1–12 (cit. on p. 146).

[87] Neelakantan R. Krishnaswami and Nick Benton. 'Ultrametric Semantics of Reactive Programs'. en. In: *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. Toronto, ON, Canada: IEEE, June 2011, pp. 257–266. ISBN: 978-1-4577-0451-2. DOI: 10.1109/LICS.2011.38 (cit. on p. 71).

[88] Wilhelm Kutta. 'Beitrag Zur Naherungsweisen Integration Totaler Differentialgleichungen'. In: *Zeitschrift für Mathematik und Physik* 46 (1901), pp. 435–453 (cit. on p. 15).

[89] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation'. In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88 (cit. on pp. 116, 139).

[90] Justin Le. *Auto: Denotative, Locally Stateful Programming DSL & Platform*. http://hackage.haskell.org/package/auto. 2015 (cit. on p. 22).

[91] Edward A. Lee and Haiyang Zheng. 'Operational Semantics of Hybrid Systems'. en. In: *Hybrid Systems: Computation and Control*. Ed. by Manfred Morari and Lothar Thiele. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 25–53. ISBN: 978-3-540-31954-2. DOI: 10.1007/978-3-540-31954-2_2 (cit. on p. 70).

[92] Hai Liu and Paul Hudak. 'Plugging a Space Leak with an Arrow'. en. In: *Electronic Notes in Theoretical Computer Science*. Festschrift Honoring Gary Lindstrom on His Retirement from the University of Utah after 30 Years of Service 193 (Nov. 2007), pp. 29–45. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.10.006 (cit. on p. 22).

[93] Jérôme Mahuet. *Flappy Haskell*. https://github.com/Rydgel/flappy-haskell. 2015 (cit. on pp. 4, 5, 60).

[94] Geoffrey Mainland. 'Why It's Nice to Be Quoted: Quasiquoting for Haskell'. en. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop - Haskell '07*. Freiburg, Germany: ACM Press, 2007, p. 73. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211 (cit. on pp. 6, 45).

[95] M. Matejak et al. 'Free Modelica Library for Chemical and Electrochemical Processes'. en. In: *Proceedings of the 11th Modelica Conference*. Versailles, France: Linköping University Electronic Press, 2015. DOI: 10.3384/ECP15118359 (cit. on p. 146).

[96] Mathworks. *Simulation and Model-Based Design*. 2020 (cit. on pp. 1, 2, 69).

[97] Sven Erik Mattsson and Gustaf Söderlind. 'Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives'. In: *SIAM Journal on Scientific Computing* 14.3 (May 1993), pp. 677–692. ISSN: 1064-8275. DOI: 10.1137/0914043 (cit. on pp. 85, 87, 89).

[98] Francesco Mazzoli and Mathieu Boespflug. *Inline-c: Write Haskell Source Files Including C Code Inline. No FFI Required*. en. https://hackage.haskell.org/package/inline-c. 2015 (cit. on p. 153).

[99] Ross McKenzie and John Pryce. 'Structural Analysis Based Dummy Derivative Selection for Differential Algebraic Equations'. en. In: *BIT Numerical Mathematics* 57.2 (June 2017), pp. 433–462. ISSN: 1572-9125. DOI: 10.1007/s10543-016-0642-9 (cit. on p. 89).

[100] Michel McKiernan. 'On the Nth Derivative of Composite Functions'. In: *The American Mathematical Monthly* 63.5 (1956), pp. 331–333. ISSN: 0002-9890. DOI: 10.2307/2310518 (cit. on p. 151).

[101] Robin Milner, Mads Tofte and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 978-0-262-63181-5 (cit. on p. 106).

[102] Rumen L. Mishkov. 'Generalization of the Formula of Faa Di Bruno for a Composite Function with a Vector Argument'. en. In: *International Journal of Mathematics and Mathematical Sciences* 24.7 (2000), pp. 481–491. ISSN: 0161-1712. DOI: 10.1155/S0161171200002970 (cit. on p. 132).

[103] Modelica Association. *Modelica ® – A Unified Object-Oriented Language for Systems Modeling*. en. Feb. 2021 (cit. on pp. 2, 101, 111).

[104] T. N. Narasimhan. 'Fourier's Heat Conduction Equation: History, Influence, and Connections'. en. In: *Proceedings of the Indian Academy of Sciences - Earth and Planetary Sciences* 108.3 (Sept. 1999), pp. 117–148. ISSN: 0973-774X. DOI: 10.1007/BF02842327 (cit. on p. 1).

[105] Ramine Nikoukhah. 'Extensions to Modelica for Efficient Code Generation and Separate Compilation'. In: *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Berlin, Germany: Linköping University Electronic Press, Jan. 2007 (cit. on p. 151).

[106] Ramine Nikoukhah and Sébastien Furic. 'Towards a Full Integration of Modelica Models in the Scicos Environment'. In: *Proceedings of the 7th International Modelica Conference*. Como, Italy, Oct. 2009. DOI: 10.3384/ecp09430024 (cit. on p. 151).

[107] Henrik Nilsson. 'Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types'. In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. New York, NY, USA: ACM, 2005, pp. 54–65. ISBN: 978-1-59593-064-4. DOI: 10.1145/1086365.1086374 (cit. on pp. 27, 32).

[108] Henrik Nilsson. 'Functional Automatic Differentiation with Dirac Impulses'. In: *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ICFP '03. New York, NY, USA: ACM, 2003, pp. 153–164. ISBN: 978-1-58113-756-9. DOI: 10.1145/944705.944720 (cit. on p. 158).

[109] Henrik Nilsson and Guerric Chupin. 'Funky Grooves: Declarative Programming of Full-Fledged Musical Applications'. In: *19th International Symposium on Practical Aspects of Declarative Languages (PADL 2017)*. Ed. by Yuliya Lierler and Walid Taha. Vol. 10137. Lecture Notes in Computer Science. Paris: Springer, Jan. 2017, pp. 163–172. ISBN: ISBN 978-3-319-51675-2. DOI: 10.1007/978-3-319-51676-9_11 (cit. on p. 4).

[110] Henrik Nilsson, Antony Courtney and John Peterson. 'Functional Reactive Programming, Continued'. In: *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*. Pittsburgh, Pennsylvania, USA: ACM, Oct. 2002, pp. 51–64 (cit. on pp. 4, 6, 37, 73, 158).

[111] Henrik Nilsson and George Giorgidze. 'Exploiting Structural Dynamism in Functional Hybrid Modelling for Simulation of Ideal Diodes'. In: *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*. Prague, Czech Republic: Czech Technical University Publishing House, Sept. 2010. ISBN: 978-80-01-04589-3 (cit. on pp. 149, 154).

[112] Henrik Nilsson, John Peterson and Paul Hudak. 'Functional Hybrid Modeling'. en. In: *Practical Aspects of Declarative Languages*. Ed. by Veronica Dahl and Philip Wadler. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 376–390. ISBN: 978-3-540-36388-0. DOI: 10.1007/3-540-36388-2_25 (cit. on pp. 3, 91, 145, 148).

[113] Brian O'Sullivan. *Criterion, a New Benchmarking Library for Haskell*. en-US. http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/. Sept. 2009 (cit. on p. 62).

[114] Lennart Ochel, Bernhard Bachmann and Francesco Casella. 'Symbolic Initialization of Over-Determined Higher-Index Models'. en. In: *Proceedings of the 10th International Modelica Conference*. Lund, Sweden, Mar. 2014, pp. 1179–1187. DOI: 10.3384/ecp140961179 (cit. on p. 90).

[115] Bjarno Oeyen, Sam Van den Vonder and Wolfgang De Meuter. 'Reactive Sorting Networks'. In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 38–50. ISBN: 9781450381888. DOI: 10.1145/3427763.3428316. URL: https://doi.org/10.1145/3427763.3428316 (cit. on p. 4).

[116] Constantinos C. Pantelides. 'The Consistent Initialization of Differential-Algebraic Systems'. In: *SIAM Journal on Scientific and Statistical Computing* 9.2 (1988), pp. 213–231 (cit. on pp. 8, 85).

[117] Blaise Pascal. *Traité du triangle artithmétique: avec quelques autres petits traités sur la même matière*. fr. An English translation of this (short) treaty by Richard Pulskamp can be found at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.422.8504. Paris: Guillaume Desprez, 1654 (cit. on p. 130).

[118] Gergely Patai. 'Efficient and Compositional Higher-Order Streams'. en. In: *Functional and Constraint Logic Programming*. Ed. by Julio Mariño. Vol. 6559. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 137–154. ISBN: 978-3-642-20774-7. DOI: 10.1007/978-3-642-20775-4_8 (cit. on p. 71).

[119] Gergely Patai. 'Eventless Reactivity from Scratch'. en. In: *Implementation and Application of Functional Languages*. 2009, pp. 126–140 (cit. on p. 71).

[120] Ross Paterson. 'A New Notation for Arrows'. In: *International Conference on Functional Programming*. Firenze, Italy: ACM Press, Sept. 2001, pp. 229–240. URL: http://www.soi.city.ac.uk/%20ross/papers/notation.html (cit. on pp. 6, 25, 45).

[121] Linda R. Petzold. 'A Description of DASSL: A Differential/Algebraic System Solver'. en. In: *IMACS World Congress*. Montreal, Canada, Aug. 1982 (cit. on p. 82).

[122] Simon Peyton Jones et al. 'Simple Unification-Based Type Inference for GADTs'. In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. New York, NY, USA: ACM, 2006, pp. 50–61. ISBN: 978-1-59593-309-6. DOI: 10.1145/1159803.1159811 (cit. on pp. 10, 33).

[123] Atze van der Ploeg and Koen Claessen. 'Practical Principled FRP: Forget the Past, Change the Future, FRPNow!' In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 302–314. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784752 (cit. on p. 71).

[124] Marc Pouzet. *Lucid Synchrone – Tutorial and Reference Manual*. en. Apr. 2006. URL: https://www.di.ens.fr/~pouzet/lucid-synchrone/manual_html/index.html (cit. on pp. 2, 5, 69).

[125] Marc Pouzet and Pascal Raymond. 'Modular Static Scheduling of Synchronous Data-Flow Networks'. en. In: *Design Automation for Embedded Systems* 14.3 (Sept. 2010), pp. 165–192. ISSN: 1572-8080. DOI: 10.1007/s10617-010-9053-3 (cit. on p. 29).

[126] J. D. Pryce. 'A Simple Structural Analysis Method for DAEs'. en. In: *BIT Numerical Mathematics* 41.2 (Mar. 2001), pp. 364–394. ISSN: 1572-9125. DOI: 10.1023/A:1021998624799 (cit. on pp. 8, 85, 86, 90).

[127] Joseph Raphson. *Analysis Æequationum Universalis*. Latin. Second. London: Thomas Brandy, 1697 (cit. on p. 83).

[128] Microsoft. *ReactiveX*. http://reactivex.io/. Accessed: 2019-05-09. 2011 (cit. on pp. 4, 22).

[129] Gunther Reissig, Wade S. Martinson and Paul I. Barton. 'Differential–Algebraic Equations of Index 1 May Have an Arbitrarily High Structural Index'. In: *SIAM Journal on Scientific Computing* 21.6 (Jan. 2000), pp. 1987–1990. ISSN: 1064-8275. DOI: 10.1137/S1064827599353853 (cit. on p. 8).

[130] Abraham Robinson. *Non-Standard Analysis*. Second. Princeton University Press, 1974 (cit. on p. 69).

[131] B. K. Rosen, M. N. Wegman and F. K. Zadeck. 'Global Value Numbers and Redundant Computations'. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. New York, NY, USA: Association for Computing Machinery, Jan. 1988, pp. 12–27. ISBN: 978-0-89791-252-5. DOI: 10.1145/73560.73562 (cit. on p. 117).

[132] Carl Runge. 'Über Die Numerische Auflösung von Differentialgleichungen'. In: *Mathematische Annalen* 46.2 (1895), pp. 167–178 (cit. on p. 15).

[133] N. Scaife et al. 'Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre'. In: *Proceedings of the 4th ACM International Conference on Embedded Software*. EMSOFT '04. New York, NY, USA: Association for Computing Machinery, Sept. 2004, pp. 259–268. ISBN: 978-1-58113-860-3. DOI: 10.1145/1017753.1017795 (cit. on p. 69).

[134] Herbert Schmidt and Silvia Hacia. 'Magnetic Force from Experiment, Equation- and Geometry-Based Calculation Using the Example of a Switching Magnet'. In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 67–76. ISBN: 978-1-4503-7713-3. DOI: 10.1145/3365984.3365992 (cit. on p. 146).

[135] Schell Scivally. *Varying: FRP through Value Streams and Monadic Splines.* `http//hackage.haskell.org/package/varying`. 2015 (cit. on p. 22).

[136] Neil Sculthorpe and Henrik Nilsson. 'Keeping Calm in the Face of Change: Towards Optimisation of FRP by Reasoning about Change'. In: *Journal of Higher-Order and Symbolic Computation* 23.2 (2011), pp. 227–271. issn: Print: 1388-3690; Electronic:1573-0557. doi: `10.1007/s10990-011-9068-x` (cit. on p. 37).

[137] Amir Shaikhha et al. 'Efficient Differentiable Programming in a Functional Array-Processing Language'. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (July 2019), 97:1–97:30. doi: `10.1145/3341701` (cit. on p. 139).

[138] P. Shieh and K. Verghese. 'A General Formula for the Nth Derivative of 1/f(x)'. In: *The American Mathematical Monthly* 74.10 (1967), pp. 1239–1240. issn: 0002-9890. doi: `10.2307/2315688` (cit. on pp. 140, 151).

[139] D D Sleator, R E Tarjan and W P Thurston. 'Rotation Distance, Triangulations, and Hyperbolic Geometry'. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing.* STOC '86. New York, NY, USA: ACM, 1986, pp. 122–135. isbn: 978-0-89791-193-1. doi: `10.1145/12130.12143` (cit. on p. 48).

[140] Ertugrul Söylemez. *Netwire: Functional Reactive Programming Library.* `http://hackage.haskell.org/package/netwire`. 2011 (cit. on p. 22).

[141] Ertugrul Söylemez. *Wires: Functional Reactive Programming Library.* `http://hackage.haskell.org/package/wires`. 2017 (cit. on pp. 4, 22).

[142] Robert Tarjan. 'Depth-First Search and Linear Graph Algorithms'. In: *SIAM Journal on Computing* 1.2 (June 1972), pp. 146–160. issn: 0097-5397. doi: `10.1137/0201010` (cit. on p. 82).

[143] Jonathan Thaler, Thorsten Altenkirch and Peer-Olaf Siebers. 'Pure Functional Epidemics: An Agent-Based Approach'. In: *IFL 2018: The 30th symposium on Implementation and Application of Functional Languages.* ACM, 2018, pp. 1–12 (cit. on pp. 4, 22).

[144] The Ptolemy Project. *System Design, Modeling, and Simulation Using Ptolemy II.* Claudius Ptolemaeus, 2014 (cit. on p. 69).

[145] John Tinnerholm, Martin Sjölund and Adrian Pop. 'Towards Introducing Just-in-Time Compilation in a Modelica Compiler'. In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools.* EOOLT '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 11–19. isbn: 978-1-4503-7713-3. doi: `10.1145/3365984.3365990` (cit. on p. 148).

[146] Ljiljana Trajković. *The Electrical Engineering Handbook.* en. Ed. by WAI-KAI Chen. Burlington: Academic Press, 2005. isbn: 978-0-12-170960-0. doi: `10.1016/B978-012170960-0/50008-6` (cit. on p. 80).

[147] Tweag I/O. *Inline-Java: Call Any JVM Function from Haskell.* Tweag. Aug. 2021 (cit. on p. 153).

[148] V. Vassilev et al. 'Clad – Automatic Differentiation Using Clang and LLVM'. en. In: *Journal of Physics: Conference Series.* Vol. 608. 1. IOP Publishing, May 2016, p. 012055. doi: `10.1088/1742-6596/608/1/012055` (cit. on p. 139).

[149] P. Wadler and S. Blott. 'How to Make Ad-Hoc Polymorphism Less Ad Hoc'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. New York, NY, USA: Association for Computing Machinery, Jan. 1989, pp. 60–76. ISBN: 978-0-89791-294-5. DOI: 10.1145/75277.75283 (cit. on p. 106).

[150] John Wallis. 'A Treatise of Algebra, Both Historical and Practical'. In: *Philosophical Transactions of the Royal Society of London* 15.173 (1685), pp. 1095–1106. DOI: 10.1098/rstl.1685.0053 (cit. on p. 83).

[151] Zhanyong Wan and Paul Hudak. 'Functional Reactive Programming from First Principles'. In: *SIGPLAN Not.* 35.5 (May 2000), pp. 242–252. ISSN: 0362-1340. DOI: 10.1145/358438.349331 (cit. on pp. 3, 21).

[152] Jeremy Yallop and Hai Liu. 'Causal Commutative Arrows Revisited'. In: *Proceedings of the 9th International Symposium on Haskell*. Haskell 2016. New York, NY, USA: ACM, 2016, pp. 21–32. ISBN: 978-1-4503-4434-0. DOI: 10.1145/2976002.2976019 (cit. on p. 72).

[153] Brent A. Yorgey et al. 'Giving Haskell a Promotion'. en. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation - TLDI '12*. Philadelphia, Pennsylvania, USA: ACM Press, 2012, p. 53. ISBN: 978-1-4503-1120-5. DOI: 10.1145/2103786.2103795 (cit. on p. 38).

[154] Dirk Zimmer. 'Equation-Based Modeling of Variable-Structure Systems'. PhD Thesis. Zurich, Switzerland: ETH Zurich, 2010 (cit. on pp. 2, 8, 113, 151).

[155] Dirk Zimmer. 'Module-Preserving Compilation of Modelica Models'. In: *Proceedings of the 7th International Modelica Conference*. Como, Italy: Linköping University Electronic Press, Oct. 2009, pp. 880–889. ISBN: 978-91-7393-513-5. DOI: 10.3384/ecp09430028 (cit. on pp. 82, 148).

[156] Dirk Zimmer. 'Robust Object-Oriented Formulation of Directed Thermofluid Stream Networks'. In: *Mathematical and Computer Modelling of Dynamical Systems* 26.3 (May 2020), pp. 204–233. ISSN: 1387-3954. DOI: 10.1080/13873954.2020.1757726 (cit. on p. 146).