First-class Models

On a Noncausal Language for Higher-order and Structurally Dynamic Modelling and Simulation

George Giorgidze, BSc, MSc

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

June 14, 2011

Abstract

The field of physical modelling and simulation plays a vital role in advancing numerous scientific and engineering disciplines. To cope with the increasing size and complexity of physical models, a number of modelling and simulation languages have been developed. These languages can be divided into two broad categories: causal and noncausal. Causal languages express a system model in terms of explicit equations. In contrast, a noncausal model is formulated in terms of implicit equations. The fact that the causality can be left implicit makes noncausal languages more declarative and noncausal models more reusable. These are considered to be crucial advantages in many physical domains.

Current, main-stream noncausal languages do not treat models as first-class values; that is, a model cannot be parametrised on other models or generated at simulation runtime. This results in very limited higher-order and structurally dynamic modelling capabilities, and limits the expressiveness and applicability of noncausal languages.

This thesis is about a novel approach to the design and implementation of noncausal languages with first-class models supporting higher-order and structurally dynamic modelling. In particular, the thesis presents a language that enables: (1) higher-order modelling capabilities by embedding noncausal models as first-class entities into a functional programming language and (2) efficient simulation of noncausal models that are generated at simulation runtime by runtime symbolic processing and just-in-time compilation. These language design and implementation approaches can be applied to other noncausal languages. This thesis provides a self-contained reference for such an undertaking by defining the language semantics formally and providing an in-depth description of the implementation. The language provides noncausal modelling and simulation capabilities that go beyond the state of the art, as backed up by a range of examples presented in the thesis, and represents a significant progress in the field of physical modelling and simulation.

"Unprovided with original learning, unformed in the habits of thinking, unskilled in the arts of composition, I resolved to write a book."

Edward Gibbon

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Physical modelling and simulation plays a vital role in the design, implementation and analysis of systems in numerous areas of science and engineering. Examples include electronics, mechanics, thermodynamics, chemical reaction kinetics, population dynamics and neural networks [Cellier, 1991]. To cope with the increasing size and complexity of physical models, a number of modelling and simulation languages have been developed. The modelling and simulation languages can be divided in two broad categories: *causal* and *noncausal*.

A causal model is formulated in terms of *explicit* equations, for example, *ordinary differential equations* (ODEs) in explicit form. That is, the cause-effect relationship is explicitly specified by the modeller [Cellier and Kofman, 2006]. In other words, the equations are directed: only *unknown* variables can appear on the left hand side of the equal sign, and only *known* variables on the other side. Since the equations are directed, it is relatively straightforward to translate a causal model into a low-level simulation code (e.g., into a sequence of assignment statements) and simulate it. Simulink is a prominent representative of causal modelling languages [Simulink, 2008].

A noncausal model is formulated in terms of *implicit* equations, for example, *differential algebraic equations* (DAEs) in implicit form. In other words, the equations are undirected: both known and unknown variables may appear on both sides of the equal sign [Cellier and Kofman, 2006]. The translation of noncausal models into simulation code involves additional symbolic processing and numerical simulation methods that are not required for causal modelling and simulation. Examples include symbolic transformations that try to causalise noncausal models and, if this is not possible, numerical

solvers for (nonlinear) implicit equations. Modelica is a prominent, state-of-the-art representative of noncausal modelling languages [Modelica, 2010].

Noncausal modelling has a number of advantages over causal modelling. The most important ones are:

- In many physical domains models are more naturally represented using noncausal equations, and in some physical domains models cannot be represented using only causal equations.
- Noncausal languages are more declarative and approach modelling problems from a higher level of abstraction by focusing on *what* to model rather than *how* to model to enable simulation.
- Noncausal models are more reusable as equations can be used in a number of different ways depending on their context of usage (i.e., effectively causalised in a number of different ways).

Although causal modelling remains a dominant paradigm, interest in noncausal modelling has grown recently as evidenced by release of noncausal modelling and simulation tools by prominent vendors such as Maple (MapleSim) and MathWorks (Simscape).

1.1 First-class Models

A language entity is *first-class* if it can be (1) passed as a parameter to functions, (2) returned as a result from functions, (3) constructed at runtime and (4) placed in data structures [Scott, 2009]. Current, main-stream noncausal languages do not treat models as first-class values [Nilsson et al., 2003]. This limits their expressiveness for *higher-order* and *structurally dynamic* modelling.

1.1.1 Higher-order Modelling

Higher-order modelling allows parametrisation of models on other models [Nilsson et al., 2003]. For instance, a car model can be parametrised on the list of tyres it is using, and an electrical transmission line model can be parametrised on the list of electrical components on the line. This style of modelling is not supported by main-stream, noncausal languages, including Modelica. Tool specific and external scripting languages are often used to generate noncausal models for particular instances of higher-order

models [Broman and Fritzson, 2008]. Whilst practical for some applications, this defeats the purpose of a declarative, noncausal modelling language.

This thesis formally defines a language that supports higher-order modelling by treating noncausal models as first-class values in a purely functional programming language and describes its implementation. In this setting, a function from model (or from collections of models placed in a suitable data structure) to model can be seen as a higher-order model and an application of this function can be seen as an instantiation of the higher-order model.

The idea to treat noncausal models as first-class values in a functional programming language was introduced by Nilsson et al. [2003] in the context of a framework called Functional Hybrid Modelling (FHM) for designing and implementing noncausal modelling languages. However, the paper postpones a concrete language definition and implementation for future work. In addition, the FHM framework proposes to exploit the first-class nature of noncausal models for modelling *hybrid* systems (i.e., systems that exhibit both continuous and discrete behaviour); this is relevant in the following section.

Broman [2007] defined and implemented a noncausal language that supports parametrisation of models on other models and allows for a form of higher-order modelling, but construction of noncausal models at simulation runtime and manipulation of collections of models placed in data structures were not considered.

1.1.2 Structurally Dynamic Modelling

Major system behaviour changes are often modelled by changing the equations that describe the system dynamics [Mosterman, 1997]. A model where the equational description changes over time is called structurally dynamic. Each structural configuration of the model is known as a *mode* of operation. Cellier and Kofman [2006] refer to structurally dynamic systems as *variable-structure* systems. Structurally dynamic systems are an example of the more general notion of hybrid systems [Nilsson et al., 2003]. The term structurally dynamic emphasises only one discrete aspect, that is, the change of equations at discrete points in time.

Cyber-physical systems [Lee, 2008], where digital computers interact with continuous physical systems, can also be seen as instances of hybrid systems. In this context, structurally dynamic modelling is relevant; as modelling of a cyber-physical system

where the digital part's influence causes major changes in the physical part may require changing the equations that describe the dynamics of the continuous part. Recently, the US National Science Foundation identified cyber-physical systems as one of its key research areas [NSF, 2008].

Current, noncausal languages offer limited support for modelling structurally dynamic systems [Mosterman, 1997, 1999, Zauner et al., 2007, Zimmer, 2008]. There are a number of reasons for this. However, this thesis concentrates on one particular reason related to the design and implementation of modelling and simulation languages: the prevalent assumption that most or all processing to put a model into a form suitable for simulation will take place *prior* to simulation [Nilsson et al., 2007, Zimmer, 2007]. By enforcing this assumption in the design of a modelling language, its implementation can be simplified as there is no need for simulation-time support for handling structural changes. For instance, a compiler can typically generate static simulation code (often just a sequence of assignment statements) with little or no need for dynamic memory or code management. This results in good performance, but such language design and implementation approaches restrict the number of modes to be modest as, in general, separate code must be generated for each mode. This rules out supporting *highly* structurally dynamic systems where the number of modes is too large or even unbounded.

There are a number of efforts to design and implement modelling and simulation languages with improved support for structural dynamism. Examples include: HYBRSIM [Mosterman et al., 1998], MOSILAB [Nytsch-Geusen et al., 2005], Sol [Zimmer, 2008] and Acumen [Zhu et al., 2010]. However, thus far, implementations have either been interpreted (HYBRSIM and Sol) and thus sacrificing the efficiency, or languages have been restricted so as to limit the number of modes to make it feasible to compile code for all modes prior to simulation (MOSILAB and Acumen).

1.1.3 Contributions

This dissertation presents a novel approach to the design and implementation of noncausal modelling and simulation languages with first-class models supporting higher-order and structurally dynamic modelling. The thesis formally defines a noncausal modelling language called Hydra and describes its implementation in detail. Hydra provides noncausal modelling and simulation capabilities that go beyond the state of

the art and represents significant progress in the field of design and implementation of declarative modelling and simulation languages, in particular:

- The thesis shows how to enable higher-order modelling capabilities by embedding noncausal models as first-class entities into a purely functional programming language. To my knowledge, Hydra is the first language that faithfully treats noncausal models as first-class values (i.e., supports all four points outlined in the beginning of Section 1.1).

- The thesis shows how to use runtime symbolic processing and *just-in-time* (JIT) compilation to enable efficient simulation of noncausal models that are generated at simulation runtime. To my knowledge, Hydra is the first language that enables support *both* for modelling and simulation of highly structurally dynamic systems and for compilation of simulation code for efficiency.

In addition to presenting the language definition and implementation, the aforementioned claims are also backed up by illustrating a range of example physical systems that cannot be modelled and simulated in current, noncausal languages. The examples are carefully chosen to showcase those language features of Hydra that are lacking in other noncausal modelling languages.

The language design choices and implementation approaches presented here can be used to enhance existing noncausal modelling and simulation languages, as well as to design and implement new modelling languages. This thesis provides a self-contained reference for such an undertaking by defining the language semantics formally and providing an in-depth description of the implementation.

Many language features of Hydra follow closely those proposed by Nilsson et al. [2003] in the context of the FHM framework and can be seen as the first concrete language definition and implementation that is based on the FHM framework. However, as already mentioned, at this stage Hydra supports only one aspect of hybrid modelling, namely, structural dynamism. Other discrete aspects that do not lead to structural reconfigurations (e.g., *impulses* [Nilsson et al., 2003, Nilsson, 2003]) are not considered in this thesis, but, in principle, can be incorporated in the Hydra language.

This work can be seen as an application of successful ideas developed in functional programming languages research to declarative modelling and simulation languages. I hope that this work will aid to further cross-fertilisation and the exchange of ideas between these research communities.

1.2 Embedding

Hydra is a Haskell-embedded *domain-specific language* (DSL). Here, the domain is noncausal modelling and simulation using implicitly formulated DAEs. Haskell is a purely functional, higher-order, statically typed programming language [Peyton Jones, 2003], which is widely used for embedded DSL development [Stewart, 2009].

Embedding is a powerful and popular way to implement DSLs [Hudak, 1998]. Compared with implementing a language from scratch, extending a suitable general-purpose programming language, the *host language*, with notions addressing a particular application or problem domain tends to save a lot of design and implementation effort. The motivation behind using an embedding approach for Hydra is to concentrate the language design and implementation effort on noncausal modelling notions that are domain specific and absent in the host language, and to reuse the rest from the host language.

Having said that, the concept of first-class models, and the runtime symbolic processing and JIT compilation approaches implemented in Hydra, are not predicated on embedded implementation. These language design and implementation approaches can be used in other noncausal modelling languages, embedded or otherwise.

There are two basic approaches to language embeddings: *shallow* and *deep*. In a shallow embedding, domain-specific notions are expressed directly in host-language terms. A shallow embedding is commonly realised as a higher-order combinator library. This is a light-weight approach for leveraging the facilities of the host language [Hudak, 1998]. In contrast, a deep embedding is about building embedded language terms as data in a suitable representation. These terms are given meaning by interpretation or compilation [Hudak, 1998]. This is a more heavy-weight approach, but also more flexible one. Indeed, it is often necessary to inspect the embedded language terms for optimisation or compilation. To benefit from the advantages of both shallow and deep embeddings, a combined approach called *mixed-level* embedding can be used [Giorgidze and Nilsson, 2010].

As mentioned in Section 1.1, Hydra supports runtime generation and JIT compilation of noncausal models. Specifically, in response to *events* occurring at discrete points in time, the simulation is stopped and, depending on the simulation results thus far, new equations are generated for further simulation [Giorgidze and Nilsson, 2009]. This kind of DSL is referred to as *iteratively staged*, emphasising that the domain is characterised

by repeated program generation, compilation and execution [Giorgidze and Nilsson, 2010].

Because performance is a primary concern in the domain, the simulation code for each mode of the model has to be compiled. As this code is determined dynamically this necessitates JIT compilation. For this part of the language Hydra employs deep embedding techniques, along with the Low Level Virtual Machine (LLVM) compiler infrastructure [Lattner, 2002], a language-independent, portable, optimising, compiler backend with JIT support. In contrast, shallow embedding techniques are used for the parts of Hydra concerned with high-level, symbolic computations [Giorgidze and Nilsson, 2010].

An alternative might have been to use a *multi-staged* host language like MetaOCaml [Taha, 2004]. The built-in runtime code generation capabilities of the host language then would have been used instead of relying on an external code generation framework such as LLVM. This approach has not been pursued, as the tight control over the dynamically generated code is essential in this application domain.

1.2.1 Contributions

Compilation of embedded DSLs is today a standard tool in the DSL-implementer's tool box. The seminal example is the work by Elliott et al. on compiling embedded languages, specifically the image synthesis and manipulation language Pan [Elliott et al., 2000]. Pan, like Hydra, provides for program generation by leveraging the host language combined with compilation to speed up the resulting performance-critical computations. However, the program to be compiled is generated once and for all, meaning the host language acts as a powerful but fundamentally conventional macro language: program generation, compilation, and execution is a process with a fixed number of stages.

Hydra is iteratively staged and, also, rather than acting merely as a powerful meta language that is out of the picture once the generated program is ready for execution, the host language is in this case part of the dynamic semantics of the embedded language through the shallow parts of the embedding. We thus add further tools to the DSL tool box for embedding a class of languages that thus far has not been studied much from an embedding and staged programming perspective.

While embedded DSL development methodology is not the main focus of this work, I nevertheless think that the thesis should be of interest to embedded DSL implementers, as it presents an application of a new embedding technique. In particular:

- The thesis presents a case study of mixed-level embedding of an iteratively staged DSL in a host language that does not provide built-in multi-stage programming capabilities.
- The thesis describes how to use JIT compilation to implement an iteratively staged embedded DSL efficiently.

1.3 Published Peer-reviewed Contributions

The content of this thesis is partly based on the peer-reviewed publications that are listed in this section. I wrote the papers in collaboration with my coauthors. This thesis was written by myself. I have implemented the software described in this dissertation and in the following papers. The software is available on my webpage[1] under the open source BSD license.

The following four papers describe various aspects of the design and implementation of Hydra, as well as a number of its applications.

- George Giorgidze and Henrik Nilsson. Embedding a Functional Hybrid Modelling language in Haskell. In *Revised selected papers of the 20th international symposium on Implementation and Application of Functional Languages, Hatfield, England*, volume 5836 of *Lecture Notes in Computer Science*. Springer, 2008.
- George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference, Como, Italy*. Linköping University Electronic Press, 2009.
- George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Revised selected papers of the 19th international workshop on Functional and (Constraint) Logic Programming, Madrid, Spain*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2010.
- George Giorgidze and Henrik Nilsson. Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In *Proceedings of the 7th*

[1] `http://www.cs.nott.ac.uk/~ggg/`

EUROSIM Congress on Modelling and Simulation, Prague, Czech Republic. Czech Technical University Publishing House, 2010.

The following two papers are about highly structurally dynamic, causal modelling and simulation using Yampa, a Haskell-embedded Functional Reactive Programming (FRP) language [Hudak et al., 2003]. The combinator that allows switching of equations during simulation runtime in Hydra draws its inspiration from switching combinators featured in Yampa [Nilsson et al., 2002, Courtney et al., 2003].

- George Giorgidze and Henrik Nilsson. Demo outline: Switched-on Yampa. In *Proceedings of the ACM SIGPLAN Haskell workshop, Freiburg, Germany*. ACM, 2007.

- George Giorgidze and Henrik Nilsson. Switched-on Yampa: declarative programming of modular synthesizers. In *Proceedings of the 10th international symposium on Practical Aspects of Declarative Languages, San Francisco, CA, USA*, volume 4902 of *Lecture Notes in Computer Science*. Springer, 2008.

Some of the embedding techniques described in this thesis are also used in the following paper.

- George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell boards the Ferry: Database-supported program execution for Haskell. In *Revised selected papers of the 22nd international symposium on Implementation and Application of Functional Languages, Alphen aan den Rijn, Netherlands*. Springer, 2010. Peter Landin Prize for the best paper at IFL 2010, to appear.

1.4 Prerequisites

Some parts of the thesis assume that the reader is familiar with Haskell, predicate logic, and BNF notation. Haskell is used for defining Hydra, as well as for implementing it. BNF notation is used for specifying the concrete syntax of Hydra. Predicate logic is used for explaining the language concepts and to give the ideal denotational semantics of Hydra.

Readers unfamiliar with Haskell may refer to the language report by Peyton Jones [2003] or one of the following books: Hutton [2007], Thompson [1999], Hudak [1999] or O'Sullivan et al. [2008]. Having said that, readers familiar with other higher-order,

typed functional programming languages, such as Standard ML [Milner et al., 1997], should also be able to follow the thesis in its entirety.

1.5 Outline

The rest of the dissertation is organised as follows:

- Chapter 2 overviews the field of physical modelling, and the state-of-the-art causal and noncausal modelling languages.
- Chapter 3 introduces the central concepts of the Hydra language.
- Chapter 4 explains how to model physical systems in Hydra by means of instructive examples. The examples were carefully chosen to showcase those language features that are absent in other noncausal modelling languages.
- Chapter 5 formally defines Hydra's concrete syntax, abstract syntax, type system and ideal denotational semantics.
- Chapter 6 describes how Hydra is implemented.
- Chapter 7 overviews the related work and, in light of it, positions the contributions of this thesis in further detail.
- Chapter 8 concludes the thesis.

Chapter 2

Background

2.1 Modelling and Simulation of Physical Systems

This chapter overviews the field of physical modelling and simulation by using simple and instructive examples. By modelling and simulating the example physical systems, basic concepts of modelling and simulation are introduced. Where necessary, the presentation abstracts from the concrete examples and defines the basic concepts more generally.

2.1.1 Mathematical Modelling

Figure 2.1 depicts a simple electrical circuit. The circuit is grounded and has the following four two-pin electrical components: voltage source, resistor, inductor and capacitor. The following system of equations is a mathematical model of the circuit.

$$u_S = sin(2\pi t) \tag{2.1a}$$

$$u_R = R \cdot i_1 \tag{2.1b}$$

$$i_1 = C \cdot \frac{du_C}{dt} \tag{2.1c}$$

$$u_L = L \cdot \frac{di_2}{dt} \tag{2.1d}$$

$$i_1 + i_2 = i \tag{2.1e}$$

$$u_R + u_C = u_s \tag{2.1f}$$

$$u_S = u_L \tag{2.1g}$$

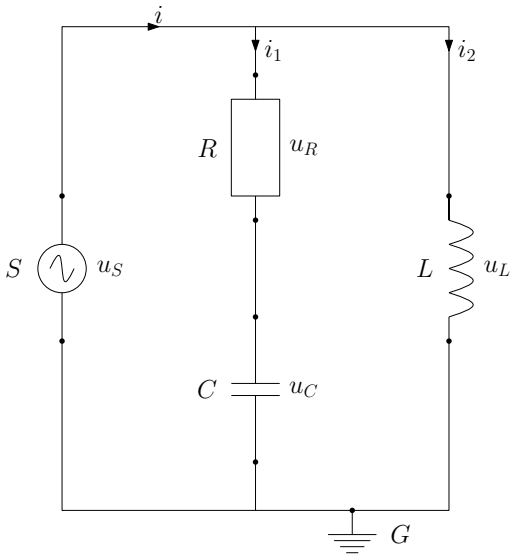

Figure 2.1: Simple electrical circuit.

The first four equations describe the component behaviours. The last three equations describe the circuit topology. The system of equations consists of implicitly defined algebraic and differential equations. This mathematical representation is a system of implicit *differential algebraic equations* (DAEs) [Cellier and Kofman, 2006]. More generally, a system of implicit DAEs can be written in the following form:

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0 \tag{2.2}$$

Here, $\vec{x}$ is a vector of *differential variables*, also known as *state variables*, $\vec{y}$ is a vector of algebraic variables and t is an independent scalar variable. In physical modelling t denotes *time*.

Numerical integration is a widely used approach for deriving approximate solutions of systems of DAEs. This is partly because, in general, exact symbolic methods do not suffice for solving systems of DAEs [Brenan et al., 1996]. There are a number of methods for numerical integration of an implicit DAE. For example, there are numerical solvers that directly operate on the implicit representation (e.g., the IDA solver from the SUNDIALS numerical suite [Hindmarsh et al., 2005]), but in some cases it is possible to translate a DAE into a system of explicit *ordinary differential equations* (ODEs), which makes it possible to simulate the system using an ODE solver (e.g., the CVODE solver from the SUNDIALS numerical suite [Hindmarsh et al., 2005]). In the following we illustrate the latter approach.

2.1.2 Symbolic Manipulation

In order to transform the implicit DAE describing the simple electrical circuit into an explicit one, we perform the following steps. Firstly, we identify *known* and *unknown* variables. Secondly, we decide which unknown variable should be solved in which equation. Thirdly, we sort the equations in such a way that no unknown variable is used before it is solved.

Time t and the differential variables u_c and i_2 are assumed to be known, the rest of the variables are unknowns, including the derivatives ($\frac{du_c}{dt}$ and $\frac{di_2}{dt}$). The equations that contain only one unknown are solved for it. After that, the solved variables are assumed to be known and rest of the variables are solved. In this case this technique suffices and we get the following explicit DAE:

$$u_S = sin(2\pi t) \tag{2.3a}$$

$$u_L = u_S \tag{2.3b}$$

$$u_R = u_S - u_C \tag{2.3c}$$

$$i_1 = \frac{u_R}{R} \tag{2.3d}$$

$$i = i_1 + i_2 \tag{2.3e}$$

$$\frac{du_C}{dt} = \frac{i_1}{C} \tag{2.3f}$$

$$\frac{di_2}{dt} = \frac{u_L}{L} \tag{2.3g}$$

This symbolic manipulation process is called *causalisation*. Now the cause-effect relationship is explicitly specified which was not the case for the implicit DAE.

Let us substitute the variables defined in the first five equations into the last two equations. This effectively eliminates the algebraic equations from the system.

$$\frac{du_C}{dt} = \frac{sin(2\pi t) - u_C}{R \cdot C} \tag{2.4a}$$

$$\frac{di_2}{dt} = \frac{sin(2\pi t)}{L} \tag{2.4b}$$

This representation is a system of explicit ODEs and can be passed to a numerical ODE solver. This representation is also called *state-space model*. More generally, a

system of explicit ODEs can be written in the following form:

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t) \tag{2.5}$$

Here, $\vec{x}$ is a vector of differential variables and t is time.

2.1.3 Numerical Integration

In the following the *forward Euler* method, which is the simplest numerical integration method for ODEs, is explained. The key idea is to replace the derivatives with the following approximation:

$$\frac{d\vec{x}}{dt} \approx \frac{\vec{x}(t+h) - \vec{x}(t)}{h} \tag{2.6}$$

Here, h is a *sufficiently small* positive scalar which is referred to as the *step size* of the numerical integration.

Let us make use of Equation 2.5 and substitute the derivative.

$$\vec{x}(t+h) \approx \vec{x}(t) + h \cdot f(\vec{x}, t) \tag{2.7}$$

Let us also fix the step size h and construct the following discrete sequences:

$$t_0 = 0, t_1 = t_0 + h,\ t_2 = t_1 + 2h, ..., t_n = t_{n-1} + nh, ... \tag{2.8}$$

$$\vec{x}_0 = \vec{x}(t_0), ..., \vec{x}_{n+1} = \vec{x}_n + h \cdot f(\vec{x}_n, t_n),\ ... \tag{2.9}$$

Here, $\vec{x}_n$ is a numerical approximation of $\vec{x}(t_n)$.

More accurate and efficient numerical integration methods are available based on different approximations and integration algorithms. A comprehensive presentation of this and other more sophisticated methods can be found in the book by Cellier and Kofman [2006].

2.1.4 Simulation

Once an initial condition (i.e., a value of the differential vector at time zero) is given it is possible to numerically integrate the ODE. The Haskell code that is given in Figure 2.2 numerically integrates the ODE given in Equation 2.4 using the forward Euler method.

```
integrateSimpleCircuit :: Double → Double → Double → Double
                        → [(Double, Double, Double)]
integrateSimpleCircuit dt r c l = go 0 0 0
  where
  go t uc i2 = let di2 = (sin (2 * π * t) / l) * dt
                   duc = ((sin (2 * π * t) − uc) / (r * c)) * dt
               in  (t, i2, uc) : go (t + dt) (uc + duc) (i2 + di2)
```

Figure 2.2: Function that numerically integrates the ODE given in Equation 2.4 using the forward Euler method.

```
integrateSimpleCircuit :: Double → Double → Double → Double
                        → [(Double, Double, Double, Double)]
integrateSimpleCircuit dt r c l = go 0 0 0
  where
  go t uc i2 = let di2 = (sin (2 * π * t) / l) * dt
                   duc = ((sin (2 * π * t) − uc) / (r * c)) * dt
                   i1  = (sin (2 * π * t) − uc) / r
               in  (t, i2, uc, i1) : go (t + dt) (uc + duc) (i2 + di2)
```

Figure 2.3: Function that adds two output equations to the function given in Figure 2.2.

Given the numerical integration time step and the circuit parameters, this function computes the approximate solution and delivers the values of the differential vector at the discrete points of time given in Equation 2.9 as a list.

In the case of the simple electrical circuit model, the algebraic variables can also be solved by adding so called *output equations* in the function that numerically integrates the system of equations. Output equations are explicit algebraic equations where the algebraic variables are defined in terms of the differential variables. The Haskell code given in Figure 2.3 refines the integration function by adding the output equation that solves the algebraic variable i_1. Figure 2.4 shows a partial simulation result obtained by evaluating the function with output equations.

The simple electrical-circuit example highlights the three essential steps involved in the process of modelling and simulation of physical systems:

- Mathematical modelling of the system behaviour
- Translation of the mathematical representation into a computer program
- Simulation of the system by compiling and executing the computer program

As we have already seen, for some systems, it is feasible to conduct this process manually. Indeed translation of systems of equations into code in general purpose pro-

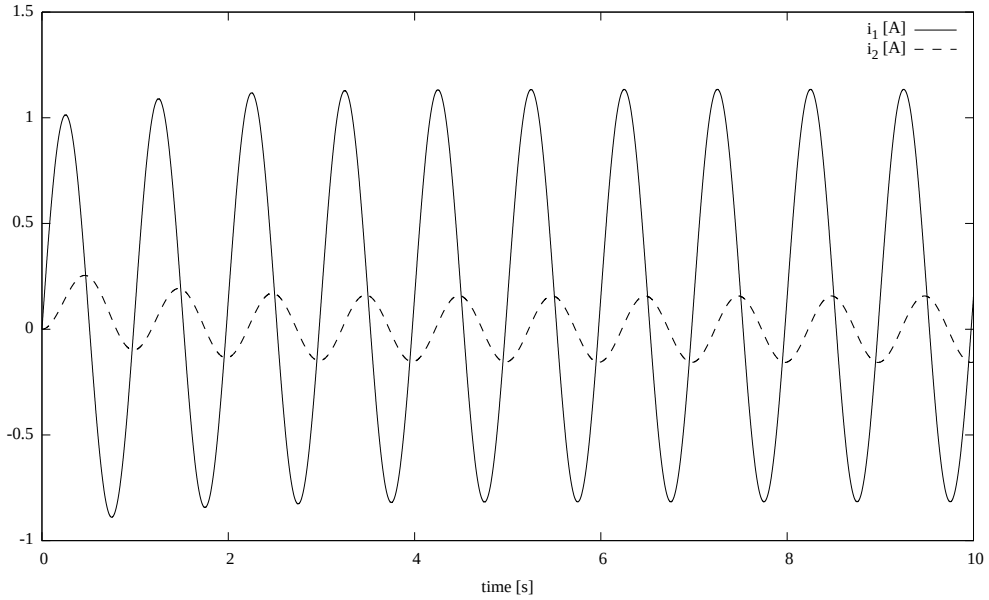

Figure 2.4: Plot showing how variables i_1 and i_2 change over time.

gramming languages like Fortran, C, Java or Haskell is a common practise. However, manual translation becomes tedious and error prone with growing complexity. Imagine conducting the process presented in this section for a physical system described with hundreds of thousands of equations.

Modelling languages and simulation tools can help with all three phases mentioned above. The following section overviews state-of-the-art representatives of causal and noncausal modelling languages. In this thesis, we focus on modelling languages capable of simulating mathematical models without assuming a particular domain of physics.

2.2 Causal Modelling in Simulink

Simulink is a graphical block diagramming tool for causal modelling and simulation. The block diagram depicted in Figure 2.5 is a model of the simple electrical circuit from Figure 2.1. Note that the diagram uses causal blocks (with input and outputs) for multiplication, summation and integration.

Block diagrams in causal languages correspond to systems of ODEs in explicit form. That is, the causality is explicit: the input and differential variables are used to define the derivatives and the output variables. The construction of a block diagram is closely related to the process of causalisation. Derivation of simulation code from a block

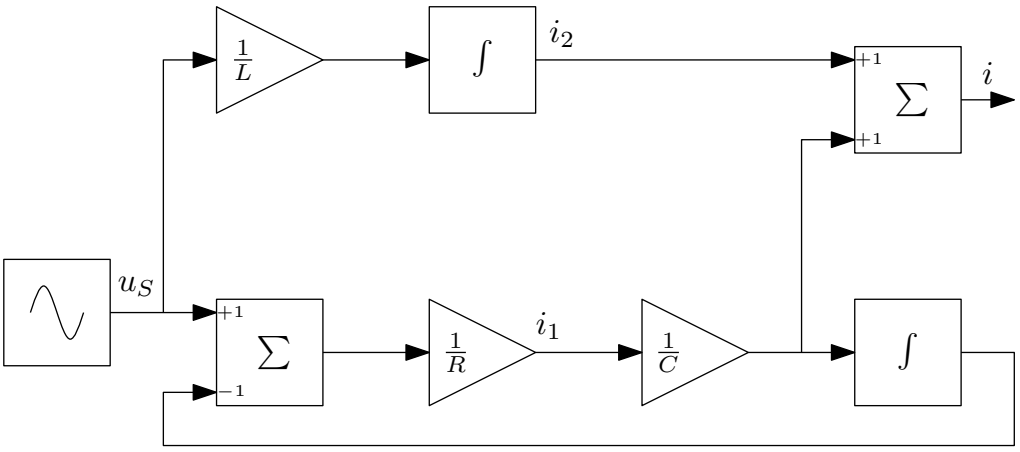

Figure 2.5: Block diagram modelling electrical circuit depicted in Figure 2.1.

diagram is done much in the same way as described in Section 2.1, but using more sophisticated numerical methods.

Structurally, the block diagram in Figure 2.5 is quite far removed from the circuit it models. Because of this, construction of block diagrams is generally regarded as a difficult and somewhat involved task [Nilsson et al., 2007]. Moreover, a slight change in a modelled system might require drastic changes in the corresponding block diagram. This is because causal models limit reuse [Cellier, 1996]. For example, a resistor behaviour is usually modelled using Ohm's law which can be written as $i = \frac{u}{R}$ or $u = R \cdot i$. Unfortunately, no single causal block can capture the resistor behaviour. If we need to compute the current from the voltage, we should use the block that corresponds to the first equation. If we need to compute the voltage from the current, we should use the block that corresponds to the second equation.

To demonstrate the aforementioned reuse problem, we modify the simple electrical circuit by adding one more resistor, as shown in Figure 2.6, and then causally model it as shown in Figure 2.7. Note that we were unable to reuse the resistor model from the original circuit diagram. Furthermore, a simple addition to the physical system caused hardly obvious changes in the causal model.

Simulink can be used to model some structurally dynamic systems: special blocks are used to *switch* between block diagrams as a response to discrete events. This makes Simulink very useful for modelling of structurally dynamic systems. However, the number of modes must be finite and all modes must be predetermined before simulation. Thus Simulink does not enable modelling and simulation of highly structurally dynamic

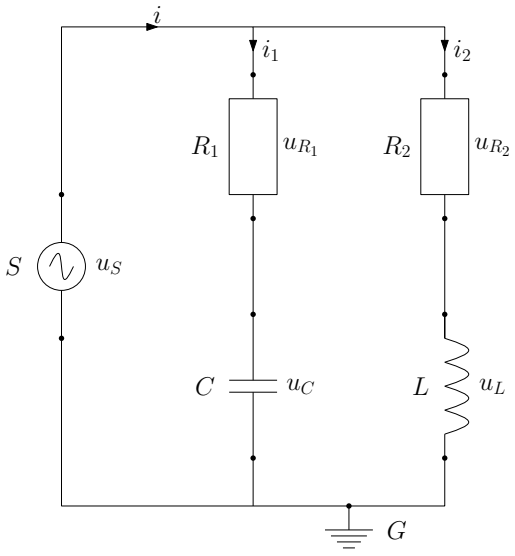

Figure 2.6: Simple electrical circuit with two resistors.

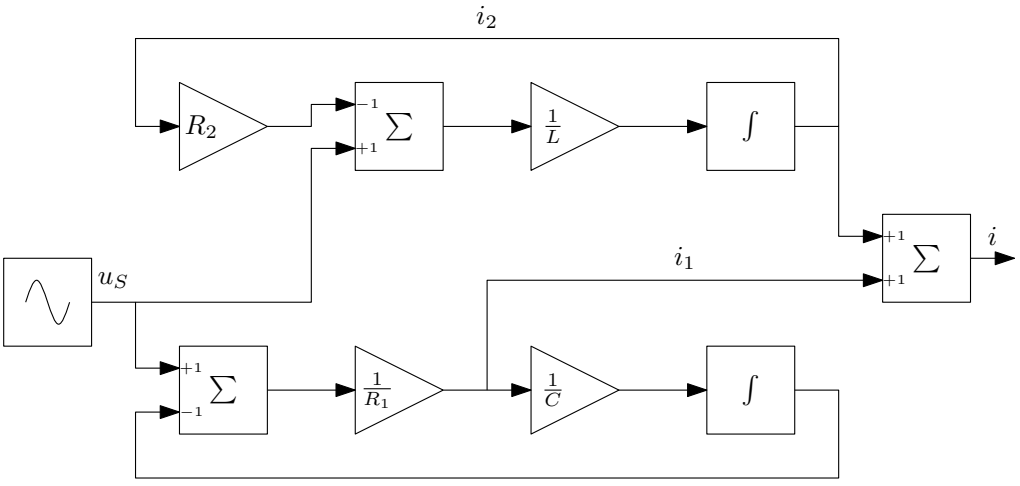

Figure 2.7: Block diagram modelling electrical circuit depicted in Figure 2.6.

```
connetor Pin
  flow Real i;
  Real v;
end Pin;
```

Figure 2.8: Connector record defined in Modelica.

systems. In addition, Simulink block diagrams are first order, thus Simulink does not support higher-order causal modelling.

2.3 Noncausal Modelling in Modelica

Modelica is a declarative language for noncausal modelling and simulation of physical systems. Modelica models are given using implicit DAEs. Modelica features a class system known from object-oriented programming languages for structuring equations and for supporting model reuse.

This section presents a Modelica model of the simple electrical circuit depicted in Figure 2.1 to illustrate basic features of the language.

The Modelica code that is given in Figure 2.8 declares the *connector* record for representing electrical connectors. The connector record introduces the variable i and the variable v representing the current flowing into the connector and the voltage at the connector respectively. In Modelica, connector records do not introduce equations. The meaning of the flow annotation is explained later on when *connect equations* are introduced.

The Modelica code that is given in Figure 2.9 defines the model that captures common properties of electrical components with two connectors. The variables p and n represent the positive and negative pins of an electrical component. The variable u represents the voltage drop across the component. The variable i represents the current flowing into the positive pin. The *TwoPin* model defines the noncausal equations that these variables satisfy.

By *extending* the *TwoPin* model with component-specific equations Figure 2.10 defines the models representing resistor, capacitor, inductor and voltage source. Figure 2.10 also defines the model that represents the ground pin. Note the use of the concept of *inheritance* known from object-oriented programming languages for reusing the equations from the *TwoPin* model.

model *TwoPin*
  Pin p, n;
  Real u, i;
equation
  $u = p.v - n.v$;
  $0 = p.i + n.i$;
  $i = p.i$;
end *TwoPin*;

Figure 2.9: Modelica model for two-pin electrical components.

Variables qualified as **parameter** or as **constant** remain unchanged during simulation. The value of a constant is defined once and for all in the source code, while a parameter can be set when an object of the class is instantiated. In this example all parameters are provided with default values allowing for instantiations with the default parameter values. All other variables represent dynamic, time-varying entities.

The Modelica model that is given in Figure 2.11 uses the circuit component models to define the simple electrical circuit model by "connecting" appropriate pins according to Figure 2.1.

Connect statements are analysed and appropriate *connection equations* are generated by the Modelica compiler as follows. Connected flow variables generate sum-to-zero equations. In this case the sum-to-zero equations correspond to Kirchhoff's current law. For the *SimpleCircuit* model the Modelica compiler generates the following sum-to-zero equations:

$$AC.n.i + C.n.i + L.n.i + G.p.i = 0;$$
$$R.n.i + C.p.i = 0;$$
$$AC.p.i + R.p.i + L.p.i = 0;$$

Connected potential variables generate equality constraints stating that all connected potential variables are equal at any point in time. For the *SimpleCircuit* model the Modelica compiler generates the following equations:

$$AC.n.v = C.n.v;$$
$$C.n.v = L.n.v;$$
$$L.n.v = G.p.v;$$
$$R.n.v = C.p.v;$$
$$AC.p.v = R.p.v;$$
$$R.p.v = L.p.v;$$

```
model Resistor
  extends TwoPin;
  parameter Real R = 1;
equation
  R * i = u;
end Resistor;


model Capacitor
  extends TwoPin;
  parameter Real C = 1;
equation
  C * der (u) = i;
end Capacitor;


model Inductor
  extends TwoPin;
  parameter Real L = 1;
equation
  u = L * der (i);
end Inductor;


model VSourceAC
  extends TwoPin;
  parameter Real VA      = 1;
  parameter Real FreqHz = 1;
  constant Real PI       = 3.14159;
equation
  u = VA * sin (2 * PI * FreqHz * time);
end VSourceAC;


model Ground
  Pin p;
equation
  p.v = 0;
end Ground;
```

Figure 2.10: Modelica models with component-specific equations.

```
model SimpleCircuit
  Resistor     R;
  Capacitor    C;
  Inductor     L;
  VSourceAC    AC;
  Ground       G;
equation
  connect (AC.p, R.p);
  connect (AC.p, L.p);
  connect (R.n,   C.p);
  connect (AC.n, C.n);
  connect (AC.n, L.n);
  connect (AC.n, G.p);
end SimpleCircuit;
```

Figure 2.11: Modelica model for the circuit given in Figure 2.1.

Connect-equations can be used in any physical domain where flow variables (i.e., variables generating sum-to-zero equations at the connection points) and potential variables (i.e, variables generating equality constrains at the connection points) can be identified. The Modelica standard library includes examples of their usage in electrical, hydraulic, and mechanical domains.

Modelica compilers generate executable simulation code from hierarchical systems of equations structured using object-oriented programming constructs by utilising state-of-the-art symbolic and numerical methods.

As we have seen, noncausal languages allow us to model physical systems at a high level of abstraction. The structure of the models resemble the modelled systems. Consequently, it is easy to reuse or modify existing models. For example, it is now trivial to add one more resistor to the Modelica model as shown in Figure 2.12.

2.4 Noncausal Modelling of Structurally Dynamic Systems

A structurally dynamic system is usually modelled using a combination of continuous equations and switching statements that specify discontinuous changes in the system. This section is about structurally dynamic modelling in noncausal languages. Current limitations are illustrated using a Modelica model of a simple structurally dynamic system. In particular, this section highlights the lack of expressiveness of the Modelica language when it comes to dynamic addition and removal of time-varying variables

```
model SimpleCircuit
  Resistor    R1;
  Resistor    R2;
  Capacitor   C;
  Inductor    L;
  VSourceAC   AC;
  Ground      G;
equation
  connect (AC.p, R1.p);
  connect (AC.p, R2.p);
  connect (R1.n, C.p);
  connect (R2.n, L.p);
  connect (AC.n, C.n);
  connect (AC.n, L.n);
  connect (AC.n, G.p);
end SimpleCircuit;
```

Figure 2.12: Modelica model for the circuit given in Figure 2.6.

and continuous equations, and lack of runtime symbolic processing and code generation facilities in Modelica implementations.

Let us model a physical system whose structural configuration changes abruptly during simulation: a simple pendulum that can break at a specified point in time; see Figure 2.13. The pendulum is modelled as a body represented by a point mass m at the end of a rigid, mass-less rod, subject to gravity $m\vec{g}$. If the rod breaks, the body will fall freely.

The code that is given in Figure 2.14 is an attempt to model this system in Modelica that on the surface appears to solve the problem. Unfortunately the code fails to compile. The reason is that the latest version of the Modelica standard [Modelica, 2010] asserts that number of equations in both branches of an if statement must be equal when the conditional expression contains a time-varying variable. If considered separately, the equations in both branches do solve the publicly available variables successfully. In an attempt to fix the model, the modeller might try to add a dummy equation for the variable not needed in the second mode (i.e., the variable ϕ, which represents the angle of deviation of the pendulum before it is broken). This version compiles, but the generated code fails to simulate the system. This example was tried using the OpenModelica [Fritzson et al., 2006] and Dymola [Dymmola, 2008] compilers.

One of the difficulties of this example is that causality changes during the switch between the two modes. In the first mode the position is calculated from the differential variable ϕ, which is not the case after the switch. This makes the job of the simula-

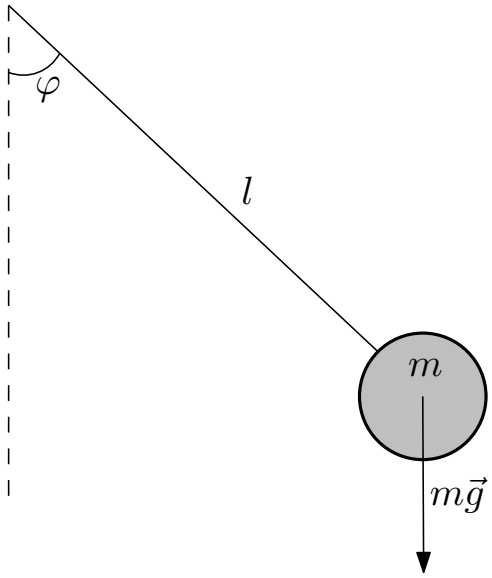

Figure 2.13: Pendulum subject to gravity.

```
model BreakingPendulum
  parameter Real l = 1, φ0 = π / 4, t = 10;
  Real x, y, vx, vy;
  protected Real φ (start = π / 2);
equation
  vx  = der x
  vy  = der y
  if (time < t) then
    x =    l * sin φ
    y = - l * cos φ
    0 = der (der φ) + (g / l) * sin φ
  else
    der vx = 0
    der vy = - g
  end if ;
end BreakingPendulum;
```

Figure 2.14: Attempt to model a breaking pendulum in Modelica.

tion code generator a lot harder and, as it turns out, the Modelica tools are not able to handle it. Specifically, the tools commit to a certain causality before they generate the simulation code. This and related issues are covered in greater detail in [Modelica Tutorial, 2000]. The suggested Modelica solution is more involved and requires reformulation of the model by making it causal. The need for manual reformulation to conform to a certain causality eliminates the advantages of working in a noncausal modelling language.

Currently, the Modelica language lacks the expressiveness to describe structural changes. The breaking pendulum example demonstrates the problems that arise when there is a need to change the number of variables in the system. In addition, the Modelica compilers carry out the symbolic processing and generate the simulation code all at once, prior to simulation, which introduces further limitations.

Chapter 3

Concepts of Hydra

This chapter introduces the three central concepts of the Hydra language: signal, signal function and signal relation. These concepts facilitate development of and reasoning about Hydra models, and are used both in informal (see Chapter 4) and formal (see Chapter 5) presentations of the language. This chapter only covers conceptual definitions; how the concepts of Hydra are implemented is covered in Chapter 6.

3.1 Signal

Conceptually, a *signal* is a time-varying value, that is, a function from time to value:

$$\textbf{type } \mathit{Time} \quad \approx \mathbb{R}$$
$$\textbf{type } \mathit{Signal}\ \alpha \approx \mathit{Time} \to \alpha$$

Time is continuous and is represented as a real number. The type parameter α specifies the type of values carried by the signal; for example, a signal of type $\mathit{Signal}\ \mathbb{R}$ may represent a change to the total amount of current flowing in a certain electrical circuit over time, or a signal of type $\mathit{Signal}\ (\mathbb{R}, \mathbb{R})$ may represent a change in position of a certain object in a two dimensional space over time.

Hydra features signals of reals (i.e., $\mathit{Signal}\ \mathbb{R}$) and signals of arbitrarily nested pairs of reals. Signals of nested pairs are useful for grouping of related signals. As an example of a signal that carries nested pairs of reals, consider a signal of type $\mathit{Signal}\ ((\mathbb{R}, \mathbb{R}), (\mathbb{R}, \mathbb{R}))$. This signal can represent current and voltage pairs at the positive and negative pins of a two-pin electrical component, for example.

3.2 Signal Function

Conceptually, a *signal function* is a function from signal to signal:

$$\textbf{type } SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

A signal function of type $SF\ \alpha\ \beta$ can be applied to an input signal of type $Signal\ \alpha$; it produces an output signal of type $Signal\ \beta$.

Because a pair of signals, say $(Signal\ \alpha, Signal\ \beta)$, is isomorphic to a signal of the pair of the carried types, in this case $Signal\ (\alpha, \beta)$, unary signal functions suffice for handling signal functions of any arity; for example, the binary signal function *add* that takes two signals and computes the sum of their values at each point in time can be given the following type and conceptual definition:

$$add :: SF\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}$$
$$add\ s \approx \lambda t \to fst\ (s\ t) + snd\ (s\ t)$$

Hydra provides a number of primitive signal functions that lift common mathematical operations (e.g., $+$, $*$, *sin* and *cos*) to the signal level. Hydra also provides the *der* signal function of type $SF\ \mathbb{R}\ \mathbb{R}$. This signal function differentiates the given signal. Later we will see that the use of the *der* signal function in noncausal equations allows for the definition of differential equations.

3.3 Signal Relation

Conceptually, a *signal relation* is a relation on signals. Stating that some signals are in a particular relation to each other imposes *constraints* on those signals. Assuming these constraints can be satisfied, this allows some of the signals to be determined in terms of the others depending on which signals are known and unknown in a given context; that is, signal relations are noncausal, unlike signal functions where the knowns and unknowns (inputs and outputs) are given a priori.

An ordinary relation can be seen as a predicate that specifies whether some given values are related or not. The same is true for signal relations:

$$\textbf{type } SR\ \alpha \approx Time \to Signal\ \alpha \to Prop$$

Given a point in time and a signal, a signal relation defines a proposition constraining the signal starting from the given point in time. Here, *Prop* is a type for propositions

defined in second-order logic. *Solving* a relation for a given starting time thus means finding a signal that satisfies the predicate.

Just like for signal functions, unary signal relations suffice for handling signal relations of any arity; for example, the following code defines a binary signal relation:

$$equal :: SR\ (\mathbb{R}, \mathbb{R})$$
$$equal\ t_0\ s \approx \forall\, t \in \mathbb{R}.\ t \geqslant t_0 \Rightarrow fst\ (s\ t) \equiv snd\ (s\ t)$$

This signal relation asserts that the first and second components of the signal s are equal starting from t_0.

Chapter 4

Hydra by Examples

This chapter presents the Hydra language informally, by means of instructive examples. The formal definition of the language and its implementation are given in Chapter 5 and Chapter 6, respectively.

4.1 Syntax of Hydra

Hydra is a two-level language. It features the *functional level* and the *signal level*. The functional level allows for the definition of ordinary functions operating on time-invariant values. The signal level allows for the definition of signal relations and signal functions on time-varying values.

Signal relations and signal functions are first-class entities at the functional level. In contrast, signals are not first-class entities at the functional level. However, crucially, instantaneous values of signals can be passed to the functional level, allowing for the generation of new signal relations that depend on signal values at discrete points in time.

A definition at the signal level may freely refer to entities defined at the functional level as the latter are time-invariant, known parameters as far as solving the signal-level equations are concerned. However, the opposite is not allowed; that is, a time-varying entity is confined to the signal level. The signal-level notions that exist at the functional level are signal relation and signal function. These notions are time-invariant.

The Hydra language is implemented as a Haskell-embedded DSL using quasiquoting [Mainland, 2007, Mainland et al., 2008]. As a result, Haskell provides the functional level for free through shallow embedding. In contrast, the signal level is realised through deep embedding; that is, signal relations expressed in terms of Hydra-specific syntax are,

through the quasiquoting machinery, turned into an internal representation, an abstract syntax tree (AST), that then is used for compilation into simulation code (see Chapter 6 for the details).

The Haskell-embedded implementation of Hydra adopts the following syntax for defining signal relations:

$$[rel \mid pattern \rightarrow equations \mid]$$

The symbol $[rel \mid$ is the opening quasiquote and the symbol $\mid]$ is the closing quasiquote. The pattern binds *signal variables* that scope over the equations that follow. The equations are DAEs stated using *signal relation application* (the operator $\diamond$). Signal relation application is how the constraints embodied by a signal relation are imposed on particular signals. The equations are required to be well typed. For example, consider the signal relation application $sr \diamond s$. Here, if sr has the type $SR\ \alpha$ then s must have the type $Signal\ \alpha$.

Hydra provides a more conventional-looking syntax for application of the built-in equality signal relation. For example, the equation $a * x + b = 0$ is equivalent to $(=) \diamond (a * x + b, 0)$.

In addition to user-defined signal relations, Hydra provides for user-defined signal functions. Hydra uses the following syntax for defining signal functions.

$$[fun \mid pattern \rightarrow expression \mid]$$

Just like for signal relations, quasiquoting is used for defining signal functions. The pattern binds signal variables that scope over the expression that follows. Signal functions can be applied to signals by juxtaposing them together:

$$sf\ s$$

Signal function applications are required to be well typed. In this example, if sf has the type $SF\ \alpha\ \beta$ then s must have the type $Signal\ \alpha$. The type of the resulting signal is $Signal\ \beta$.

The quasiquotes, in addition to serving as an embedded DSL implementation tool, can be seen as clear syntactic markers separating the signal level from the functional level. These markers are useful when reading Hydra code listings. The separation is also enforced at the type level of the host language by the SR and SF type constructors.

Because signals are not first-class entities at the functional level, it is not possible to construct a value of type *Signal* α directly at the functional level. Signals only exist indirectly through the signal level definitions of signal relations and signal functions.

4.2 The *switch* Combinator

The built-in equality signal relation (i.e., $=$) is capable of describing flat systems of equations and the signal relation application operator (i.e., $\diamond$) provides for hierarchically structured systems of equations. In this section we introduce one more built-in (higher-order) signal relation that allows for description of structurally dynamic signal relations.

$$switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$$

The *switch* combinator forms a signal relation by temporal composition. The combinator takes three arguments and returns the composite signal relation (of type $SR\ a$). The first argument (of type $SR\ a$) is a signal relation that is initially active. The second argument is a signal function (of type $SF\ a\ \mathbb{R}$). Starting from the first point in time when the signal (of type *Signal* $\mathbb{R}$) that is computed by applying the signal function to the signal constrained by the composite signal relation crosses zero (i.e., changes its sign from negative to positive or from positive to negative), the composite behaviour is defined by the signal relation that is computed by applying the third argument (a function of type $a \to SR\ a$) to the instantaneous value of the constrained signal. A formally defined meaning of the *switch* combinator is given in Chapter 5.

The *switch* combinator allows for definition of a signal relation whose equational description changes over time. In addition, the *switch* combinator allows for state transfer from the old mode and initialisation of the new mode using the function that computes the new mode from an instantaneous value of the constrained signal.

In the signal relation notation described earlier, the list of equations that follows the pattern is not necessarily a static one as the equations may contain a signal relation application of a structurally dynamic signal relation.

4.3 Models with Static Structure

Let us introduce the Hydra language by modelling the circuit that is depicted in Figure 2.1. Let us first define the *twoPin* signal relation that captures the common behaviour of electrical components with two connectors (see Figure 4.1):

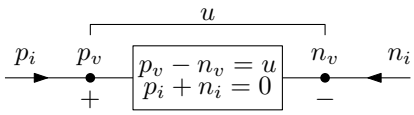

Figure 4.1: Electrical component with two connectors.

```
type Pin = (ℝ, ℝ)

twoPin :: SR ((Pin, Pin), ℝ)
twoPin = [rel | (((p_i, p_v), (n_i, n_v)), u) →
  p_v − n_v = u
  p_i + n_i = 0
|]
```

The signal variables p_i and p_v, which are bound in the pattern, represent the current into the positive pin and the voltage at the positive pin, respectively. The signal variables n_i and n_v, which are also bound in the pattern, represent the current into the negative and the voltage at the negative pin, respectively. The signal variable u represents the voltage drop across the electrical component.

We can now use the *twoPin* signal relation to define a signal relation that models a resistor:

```
resistor :: ℝ → SR (Pin, Pin)
resistor r = [rel | ((p_i, p_v), (n_i, n_v)) →
  local u
  $ twoPin $ ◇(((p_i, p_v), (n_i, n_v)), u)
  $ r $ ∗p_i = u
|]
```

Note that a parametrised signal relation is an ordinary function returning a signal relation. In the *resistor* signal relation, the signal variable u is declared as a local signal variable; that is, it is not exposed in the pattern of the signal relation. As a consequence, u can only be constrained in this signal relation, unlike the rest of the variables in the pattern, which can be constrained further.

Hydra uses two kinds of variables: the functional-level ones representing time-invariant parameters, and the signal-level ones, representing time-varying entities, the signals. Functional-level fragments, such as variable references, are spliced into the signal level by enclosing them between antiquotes, \$. On the other hand time-varying entities are

not allowed to escape to the functional level; that is, signal-variables are not in scope between antiquotes and outside the quasiquotes.

The *resistor* signal relation uses antiquoting to splice in a copy of the *twoPin* signal relation; that is, its equations are reused in the context of the resistor model. Readers familiar with object-oriented, noncausal languages like Modelica, can view this as a definition of the resistor model by extending the *twoPin* model with an equation that characterises the specific concrete electrical component, in this case Ohm's law.

To clearly see how *twoPin* contributes to the definition of the *resistor* signal relation, let us consider what happens when the resistor model is *flattened* as part of flattening of a complete model, a transformation that is described in detail in Chapter 6. Intuitively, flattening can be understood as inlining of applied signal relations to reduce the signal relation into a list of signal equality constraints (i.e., a flat DAE). In the process of flattening, the arguments of a signal relation application are substituted into the body of the applied signal relation, and the entire application is then replaced by the instantiated signal relation body. In our case, the result of flattening the signal relation *resistor* 10 is:

$$
\begin{aligned}
&[\,rel \mid ((p_i, p_v), (n_i, n_v)) \rightarrow \\
&\quad \textbf{local}\ u \\
&\quad p_v - n_v = u \\
&\quad p_i + n_i \ = 0 \\
&\quad 10 * p_i \ = u \\
&\,|]
\end{aligned}
$$

Models for an inductor, a capacitor, a voltage source and a ground reference are defined in Figure 4.2. Note that the inductor and the capacitor signal relations contain **init** equations. An **init** equation is enforced only at the point in time when the signal relation becomes active. In this example, the **init** equations are used to initialise the differential variables involved in the inductor and the capacitor signal relations.

Modelica implicitly initialises differential variables to zero. That is why initialisation equations were not considered in the corresponding Modelica models given in Chapter 2. Hydra does not allow for implicit initialisation; that is, all initialisation equations must be specified explicitly.

```
iInductor :: ℝ → ℝ → SR (Pin, Pin)
inductor p_{i_0} l = [rel | ((p_i, p_v), (n_i, n_v)) →
  init p_i = p_{i_0}
   $ twoPin $ ◇(((p_i, p_v), (n_i, n_v)), u)
   $ l $ *der p_i = u
|]

iCapacitor :: ℝ → ℝ → SR (Pin, Pin)
iCapacitor u_0 c = [rel | ((p_i, p_v), (n_i, n_v)) →
  init u = u_0
   $ twoPin $ ◇(((p_i, p_v), (n_i, n_v)), u)
   $ c $ *der u = p_i
|]

vSourceAC :: ℝ → ℝ → SR (Pin, Pin)
vSourceAC v f = [rel | ((p_i, p_v), (n_i, n_v)) →
   $ twoPin $ ◇(((p_i, p_v), (n_i, n_v)), u)
  u = $v $ *sin (2 * $π $ * $ f $ *time)
|]

ground :: SR (Pin)
ground = [rel | (p_i, p_v) where
  p_v = 0
|]
```

Figure 4.2: Hydra models for an inductor, a capacitor, a voltage source and a ground reference.

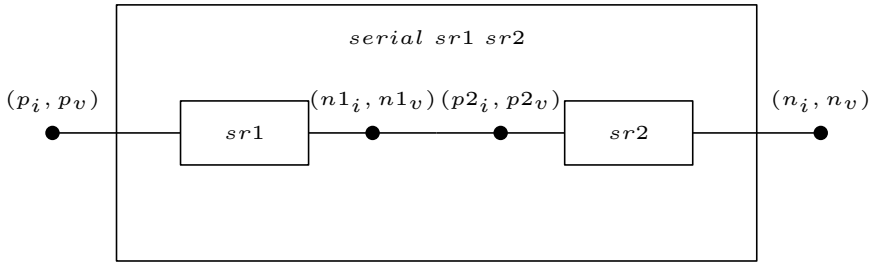

Figure 4.3: Serial connection of two electrical components.

4.4 Noncausal Connections

Unlike Modelica, Hydra does not provide a special language construct for specifying noncausal connections; that is, Hydra does not provide a construct like **connect** from the Modelica language. However, because signal relations are first-class entities, it is possible to implement higher-order combinators that facilitate connection of noncausal models.

To model the simple electrical circuit as an interconnection of the already modelled components let us define three higher-order signal relations facilitating noncausal connection of two-pin electrical components.

Firstly, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the serial connection of the two electrical components. The graphical representation of the signal relation is given in Figure 4.3.

$$
\begin{aligned}
&\mathit{serial} :: \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \rightarrow \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \rightarrow \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\\
&\mathit{serial}\ \mathit{sr1}\ \mathit{sr2} = [\mathit{rel} \mid ((p_i, p_v), (n_i, n_v)) \rightarrow\\
&\quad \mathbf{local}\ \mathit{p1}_i; \mathbf{local}\ \mathit{p1}_v; \mathbf{local}\ \mathit{n1}_i; \mathbf{local}\ \mathit{n1}_v;\\
&\quad \$\ \mathit{sr1}\ \$\ \diamond ((\mathit{p1}_i, \mathit{p1}_v), (\mathit{n1}_i, \mathit{n1}_v))\\
&\quad \mathbf{local}\ \mathit{p2}_i; \mathbf{local}\ \mathit{p2}_v; \mathbf{local}\ \mathit{n2}_i; \mathbf{local}\ \mathit{n2}_v;\\
&\quad \$\ \mathit{sr2}\ \$\ \diamond ((\mathit{p2}_i, \mathit{p2}_v), (\mathit{n2}_i, \mathit{n2}_v))\\
&\quad (-p_i) + \mathit{p1}_i = 0\\
&\quad p_v = \mathit{p1}_v\\
&\quad \mathit{n1}_i + \mathit{p2}_i = 0\\
&\quad \mathit{n1}_v = \mathit{p2}_v\\
&\quad \mathit{n2}_i + (-n_i) = 0\\
&\quad \mathit{n2}_v = n_v\\
&|]
\end{aligned}
$$

Secondly, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the

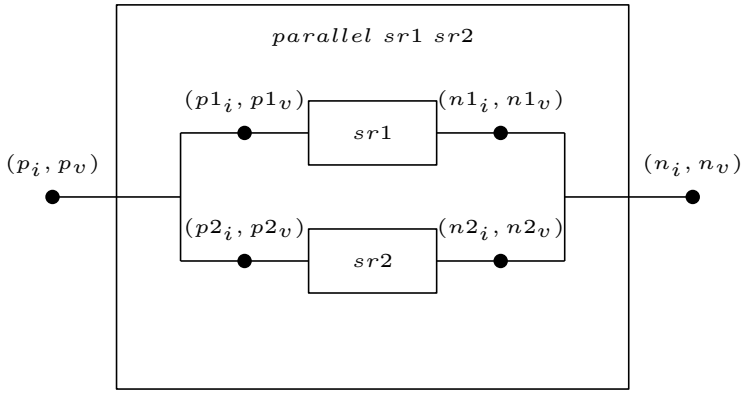

Figure 4.4: Parallel connection of two electrical components.

parallel connection of the two electrical components. The graphical representation of the signal relation is given in Figure 4.4.

$$
\begin{aligned}
&\mathit{parallel} :: \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \to \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \to \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\\
&\mathit{parallel}\ \mathit{sr1}\ \mathit{sr2} = [\mathit{rel} \mid ((p_i, p_v), (n_i, n_v)) \to\\
&\quad \textbf{local}\ \mathit{p1}_i; \textbf{local}\ \mathit{p1}_v; \textbf{local}\ \mathit{n1}_i; \textbf{local}\ \mathit{n1}_v;\\
&\quad \$\ \mathit{sr1}\ \$\ \diamond ((\mathit{p1}_i, \mathit{p1}_v), (\mathit{n1}_i, \mathit{n1}_v))\\
&\quad \textbf{local}\ \mathit{p2}_i; \textbf{local}\ \mathit{p2}_v; \textbf{local}\ \mathit{n2}_i; \textbf{local}\ \mathit{n2}_v;\\
&\quad \$\ \mathit{sr2}\ \$\ \diamond ((\mathit{p2}_i, \mathit{p2}_v), (\mathit{n2}_i, \mathit{n2}_v))\\
&\quad (-p_i) + \mathit{p1}_i + \mathit{p2}_i = 0\\
&\quad p_v = \mathit{p1}_v\\
&\quad \mathit{p1}_v = \mathit{p2}_v\\
&\quad (-n_i) + \mathit{n1}_i + \mathit{n2}_i = 0\\
&\quad n_v = \mathit{n1}_v\\
&\quad \mathit{n1}_v = \mathit{n2}_v\\
&|]
\end{aligned}
$$

Finally, we define a higher-order signal relation that takes two signal relations modelling two-pin electrical components and returns the signal relation that models the grounded circuit involving the two electrical components. The graphical representation of the signal relation is given in Figure 4.5.

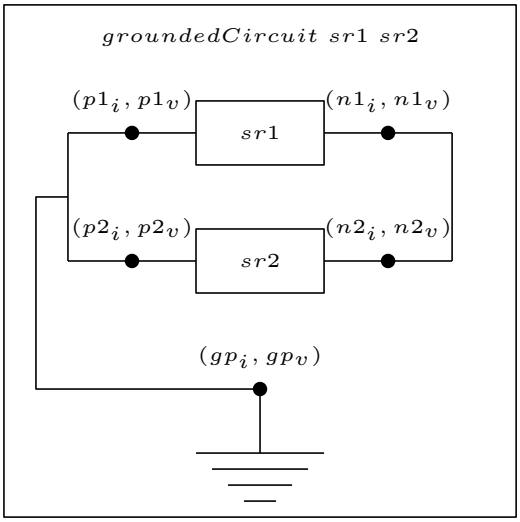

Figure 4.5: Grounded circuit involving two electrical components.

```
groundedCircuit :: SR (Pin, Pin) → SR (Pin, Pin) → SR ()
groundedCircuit sr1 sr2 = [rel | () →
  local p1_i; local p1_v; local n1_i; local n1_v;
  $ sr1 $ ◇((p1_i, p1_v), (n1_i, n1_v))
  local p2_i; local p2_v; local n2_i; local n2_v;
  $ sr2 $ ◇((p2_i, p2_v), (n2_i, n2_v))
  local gp_i; local gp_v;
  $ ground $ ◇(gp_i, gp_v)
  p1_i + p2_i = 0
  p1_v + p2_v = 0
  n1_i + n2_i + gp_i = 0
  n1_v = n2_v
  n2_v = gp_v
|]
```

Now we can assemble the models of the electrical components into the simple electrical circuit as follows:

```
simpleCircuit :: SR ()
simpleCircuit =
  groundedCircuit (vSourceAC 1 1)
                  (parallel (serial (resistor 1) (iCapacitor 0 1))
                            (iInductor 0 1))
```

Note that the above code is a direct textual representation of how the components are connected in the circuit. Having said that, unlike the Modelica model that specifies the noncausal connections in terms of connections of time-varying variables, Hydra al-

lows for definition of higher-order combinators that are capable of specifying noncausal connections by connecting noncausal models directly.

It is trivial in Hydra to reuse the circuit components and model the modified circuit that is depicted on Figure 2.6:

```
simpleCircuit2 :: SR ()
simpleCircuit2 =
  groundedCircuit (vSourceAC 1 1)
                  (parallel (serial (resistor 1) (iCapacitor 0 1))
                            (serial (resistor 1) (iInductor  0 1)))
```

4.5 Simulation

The Haskell-embedded implementation of Hydra features the following function:

```
simulate :: SR () → Experiment → IO ()
```

The *simulate* function takes two arguments. The first argument is the signal relation that needs to be simulated. The second argument describes the *experiment*, essentially a description of what needs to happen during the simulation. Using the second argument the modeller can set the simulation starting and ending times, desired time step, symbolic processor, numerical solver, or how to visualise the trajectories of the constrained signals. The definition of the *Experiment* data type and the default experiment description are given in Chapter 6.

The simple circuit model can be simulated using the default experiment description as follows:

```
simulate simpleCircuit defaultExperiment
```

With the *defaultExperiment* parameter the *simpleCircuit* signal relation is simulated for 10 seconds of simulation time starting from the time point of zero. The time step is set to 0.001 and the trajectories of the constrained signals are printed to the standard output in the gnuplot[1] compatible format. The default numerical solver is SUNDIALS [Hindmarsh et al., 2005], but users are allowed to provide their own symbolic processors and numerical solvers. This and other implementation aspects are described in detail in Chapter 6.

[1] `http://www.gnuplot.info/`

The earlier sections of this chapter introduced the Hydra language using the simple electrical-circuit example. This example allows readers familiar with object-oriented, noncausal languages like Modelica to compare the Hydra model given in this Chapter to the Modelica model given in Chapter 2. The rest of the Chapter focuses on the features of Hydra that are absent from main-stream, noncausal modelling languages. Specifically, we discuss the higher-order and structurally dynamic modelling capabilities of the Hydra language.

4.6 More Higher-order Modelling

We have already seen several higher-order models; for example, the *serial*, *parallel* and *groundedCircuit* signal relations. This section considers more higher-order modelling examples, but this time concentrating on signal relations that are parametrised on collections of signal relations. The examples are again from the physical domain of electronics.

Let us define a higher-order signal relation that takes as an argument a list of signal relations modelling two-pin electrical components and returns the signal relation that models serial connection of the given electrical components. The following higher-order signal relation can be used to model electronic transmission lines, for example.

$$serialise :: [SR\ (Pin, Pin)] \rightarrow SR\ (Pin, Pin)$$
$$serialise = foldr\ serial\ wire$$

This definition begs for detailed explanation. The *foldr* function is defined in the standard Haskell prelude and has the following type signature and definition.

$$foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$$
$$foldr\ \_\ z\ [\,] = \quad z$$
$$foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$$

The function takes as arguments a binary operator, a starting value that is typically the right-identity of the binary operator, and a list. The *foldr* function folds the list using the binary operator, from right to left:

$$foldr\ serial\ wire\ [sr1, sr2, ..., srn] =$$
$$sr1\ `serial`\ (sr2\ `serial`\ ...(srn\ `serial`\ wire)...)$$

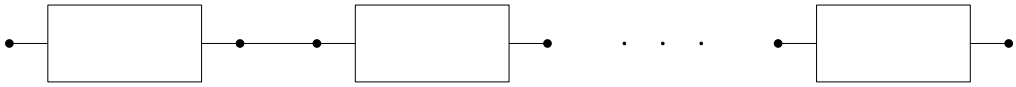

Figure 4.6: Serial connection of electrical components.

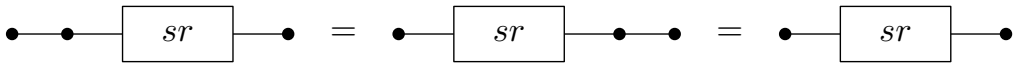

Figure 4.7: The *wire* signal relation as a left and right identity of the *serial* higher-order signal relation.

Here the higher-order signal relation *serial* is in the role of a binary operator and the *wire* signal relation is in the role of a starting value which is a right identity of the binary operator. Figure 4.6 graphically demonstrates the result of this application of the *foldr* function.

The *wire* signal relation models an electrical wire and is defined as follows.

$$
\begin{aligned}
&wire :: SR\ (Pin, Pin)\\
&wire = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \rightarrow\\
&\quad \$\ twoPin\ \$ \diamond ((p_i, p_v), (n_i, n_v), u)\\
&\quad u = 0\\
&\,|\,]
\end{aligned}
$$

Just like other two-pin electrical components, the *wire* signal relation is modelled by extending the *twoPin* signal relation with a suitable equation.

The *wire* signal relation is both left and right identity of the *serial* higher-order signal as stated by the following equation and illustrated in Figure 4.7.

$$wire\ `serial`\ sr = sr\ `serial`\ wire = sr$$

Here by the equality of the signal relations we mean that the signal relations introduce equivalent constrains, and not necessarily the same equations. Because the *wire* signal relation is both left and right identity of the *serial* binary function, in the definition of the *serialise* signal relation we could also use the left fold instead of the right fold.

Somewhat similarly to the *serialise* signal relation the higher-order signal relation *parallelise* that takes as an argument a list of signal relations modelling two-pin electrical components and returns the signal relation that models parallel connection of the given electrical components can be defined as follows:

$$
\begin{aligned}
&parallelise :: [SR\ (Pin, Pin)] \rightarrow SR\ (Pin, Pin)\\
&parallelise = foldr\ parallel\ noWire
\end{aligned}
$$

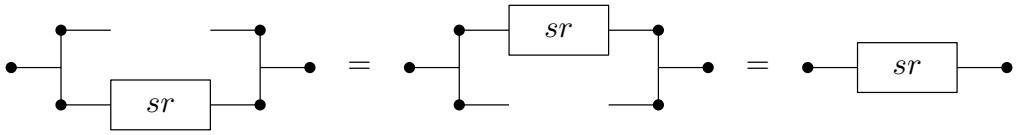

Figure 4.8: The *noWire* signal relation as a left and right identity of the *parallel* higher-order signal relation.

The *noWire* signal relation is defined as follows:

$$
\begin{aligned}
&noWire :: SR\ (Pin, Pin)\\
&noWire = [\,rel \mid ((p_i, p_v), (n_i, n_v)) \rightarrow\\
&\quad \$\ twoPin\ \$ \diamond((p_i, p_v), (n_i, n_v), u)\\
&\quad p_i = 0\\
&\,|]
\end{aligned}
$$

The *noWire* signal relation is both left and right identity of the *parallel* higher-order signal relation as stated by the following equation and illustrated in Figure 4.8.

$$noWire\ `parallel`\ sr = sr\ `parallel`\ noWire = sr$$

In addition to demonstrating higher-order modelling capabilities of Hydra, this section puts an emphasis on how the host, higher-order functional language can provide expressive facilities for higher-order, noncausal modelling.

4.7 Structurally Dynamic Modelling

To introduce structurally dynamic modelling in Hydra, let us model the breaking-pendulum system described in Section 2.4. The system system has two modes of operation. The differences between the two modes are sufficiently large that, for example, Modelica does not support noncausal modelling of this system, as discussed in Section 2.4.

The code that is given in Figure 4.9 shows how to model the two modes of the pendulum in Hydra. The type *Body* denotes the state of the pendulum body; that is, its position and velocity, where position and velocity both are 2-dimensional vectors represented by pairs of reals. Each model is represented by a function that maps the parameters of the model to a relation on signals. In the unbroken mode, the parameters are the length of the rod l and the initial angle of deviation *phi0*. In the broken mode, the signal relation is parametrised on the initial state of the body. Once again, note that

```
type Pos  = (ℝ, ℝ)
type Vel  = (ℝ, ℝ)
type Body = (Pos, Vel)
g :: ℝ
g = 9.81
freeFall :: Body → SR Body
freeFall ((x0, y0), (vx0, vy0)) = [rel | ((x, y), (vx, vy)) →
  init (x, y)       = ($x0$, $y0$)
  init (vx, vy)     = ($vx0$, $vy0$)
  (der x, der y)    = (vx, vy)
  (der vx, der vy)  = (0, −$g$)
|]
pendulum :: ℝ → ℝ → SR Body
pendulum l φ0 = [rel | ((x, y), (vx, vy)) →
  local φ
  init φ     =    $ φ0 $
  init der φ =    0
  init vx    =    0
  init vy    =    0
  x          =        $ l $ ∗ sin φ
  y          =    −   $ l $ ∗ cos φ
  (vx, vy)   =    (der x, der y)
  der (der φ) + ($g / l$) ∗ sin φ = 0
|]
```

Figure 4.9: Signal relations modelling the two modes of the pendulum.

the equations that are marked by the keyword **init** are initialisation equations used to specify initial conditions.

To model a pendulum that breaks at some point, we need to create a composite signal relation where the signal relation that models the dynamic behaviour of the unbroken pendulum is replaced, at the point when it breaks, by the signal relation modelling a free falling body. These two submodels must be suitably joined to ensure the continuity of both the position and velocity of the body of the pendulum.

To this end, the *switch* combinator is used:

```
breakingPendulum :: ℝ → ℝ → ℝ → SR Body
breakingPendulum t l phi0 =
  switch (pendulum l phi0) [fun | λ_ → time − $t$ |] (λb → freeFall b)
```

In this signal relation, the switch happens at an *a priori* specified point in time, but the switching condition could be an arbitrary time-varying entity. Note how the succeeding signal relation (i.e., *freeFall*) is initialised so as to ensure the continuity of

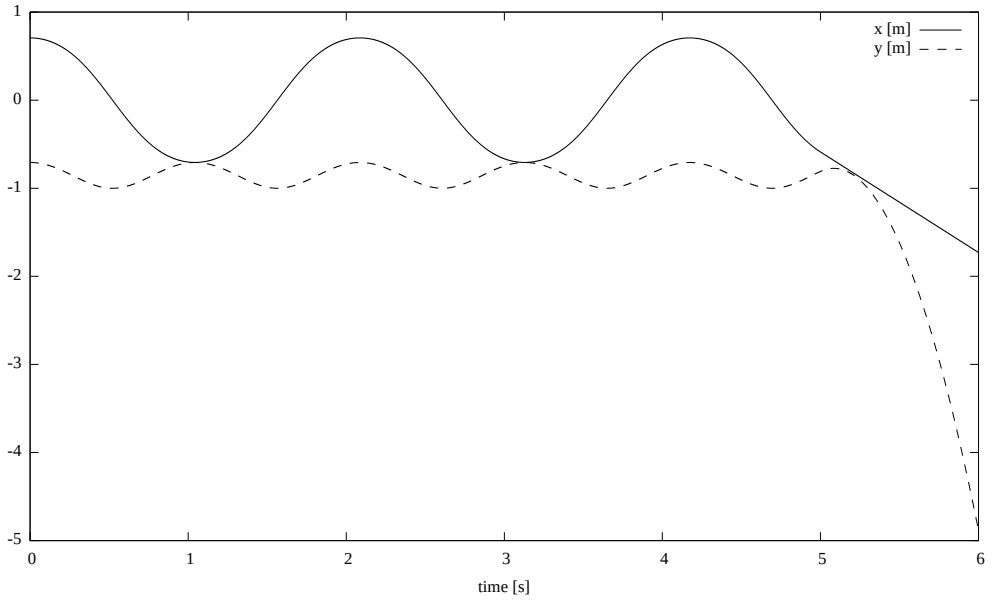

Figure 4.10: Plot showing how x and y coordinates of the body on the breaking pendulum change over time.

the position and velocity as discussed above. The simulation results obtained by the *simulate* function can be seen in Figure 4.10

We have already demonstrated how the *switch* combinator can be used to dynamically add and remove signal variables and noncausal equations. This flexibility is useful even when the number of equations and variables remain unchanged during the simulation. The book by Cellier and Kofman [2006] gives one such example: the half-wave rectifier circuit with an ideal diode and an in-line inductor that is depicted in Figure 4.11.

The half-wave rectifier circuit can be modelled easily in languages like Modelica. However, any attempt to simulate this model assuming fixed causality, as current mainstream noncausal language implementations tend to, will fail as the causalised model will lead to a division by zero when the switch is open: there simply is no one fixed causality model that is valid both when the switch is open and closed.

One reason is that noncausal modelling languages tend to be designed and implemented on the assumption that the causality of the model does not change during simulation. This assumption simplifies the language design and facilitates the generation of efficient simulation code. In particular, the causality can be analysed and code can be generated once and for all, at compile time, paving the way for using a fast, explicit solver for simulation as demonstrated in Chapter 2.

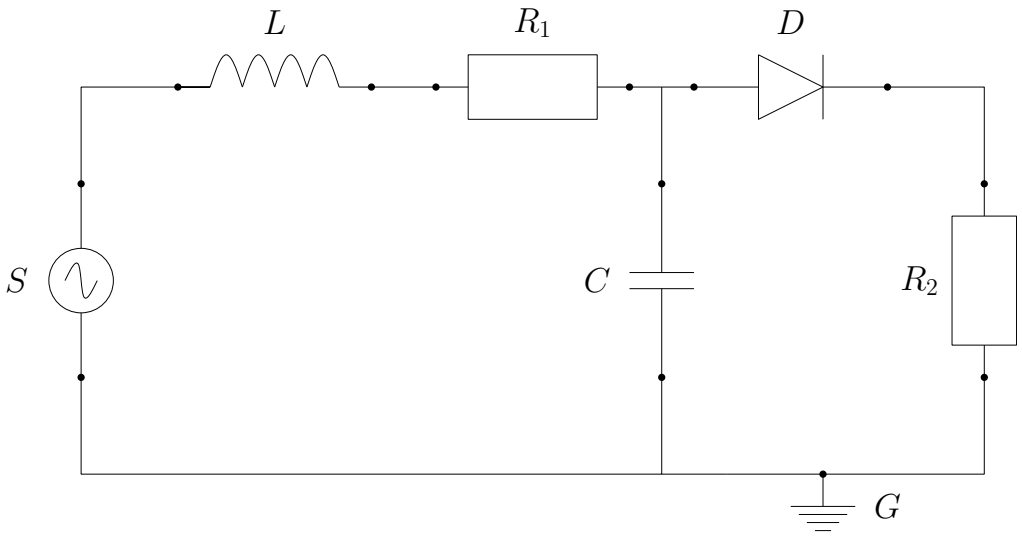

Figure 4.11: Half-wave rectifier circuit with an ideal diode and an in-line inductor.

One common solution to the division-by-zero problem in models involving ideal diodes is to avoid the ideal model and opt for a leaky diode model instead. This works, but often leads to very stiff equations. Thus, if an ideal model would suffice for the purpose at hand, that would be preferable [Cellier and Kofman, 2006].

In the following we model the half-wave rectifier circuit in Hydra. Since the Hydra language is not predicated on compilation of simulation all at once and allows for runtime symbolic processing it addresses the shortcomings of main-stream, noncausal modelling languages discussed earlier in this section.

The following two signal relations model initially opened (*ioDiode*) and initially closed (*icDiode*) ideal diodes. Note the use of the host language feature of mutual recursion in the following definitions allowing for signal relations to switch into each other.

$$
\begin{aligned}
&ioDiode :: SR\ (Pin, Pin)\\
&ioDiode = switch\ nowire\ [fun \mid ((\_, p_v), (\_, n_v)) \rightarrow p_v - n_v \mid]\ (\lambda\_ \rightarrow icDiode)\\
&icDiode :: SR\ (Pin, Pin)\\
&icDiode = switch\ wire \quad [fun \mid ((p_i, \_), (\_, \_)) \rightarrow p_i \quad \mid]\ (\lambda\_ \rightarrow ioDiode)
\end{aligned}
$$

The switches are controlled by the polarity of the voltage and the current through the component. Now we can assemble the half-wave rectified circuit into a single signal relation using the higher-order connection combinators defined earlier in this chapter.

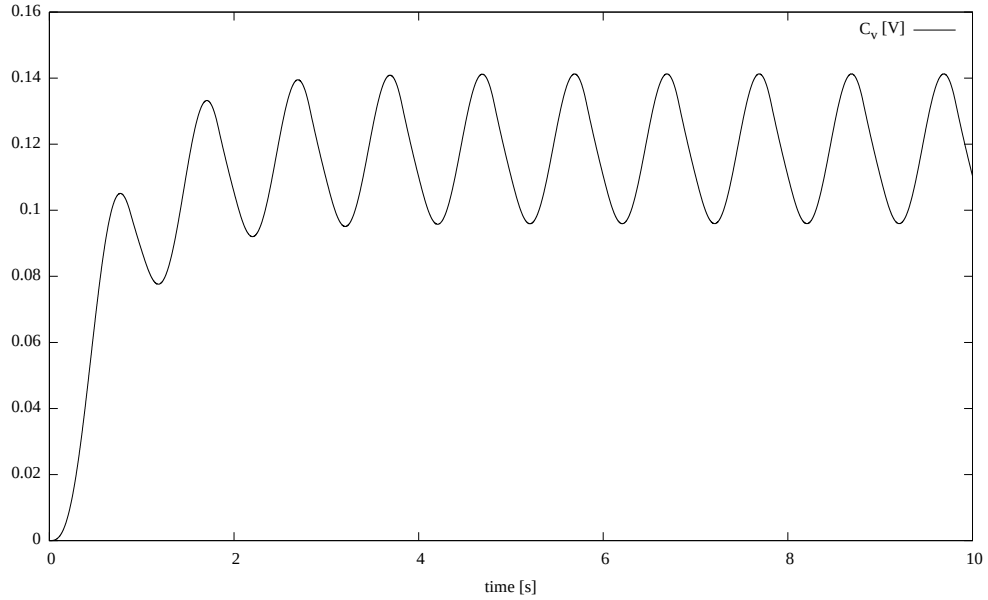

Figure 4.12: Voltage across the capacitor in the half-wave rectifier circuit with in-line inductor.

```
halfWaveRectifier :: SR ()
halfWaveRectifier =
  groundedCircuit (vSourceAC 1 1)
                  (serialise [iInductor 0 1
                             , resistor 1
                             , icDiode
                             , parallel (iCapacitor 0 1) (resistor 1)
                             ]
                  )
```

Partial simulation results of the *halfWaveRectifier* signal relation obtained by using the *simulate* function are presented in Figure 4.12 and in Figure 4.13

4.8 Highly Structurally Dynamic Modelling

The breaking pendulum and half-wave rectifier examples feature only two modes of operation. In principle (with a suitable language design and implementation) it is feasible to generate code for two modes of operation prior to simulation. However, despite their simplicity, these are examples with which main-stream noncausal languages such as Modelica struggle, as mentioned earlier.

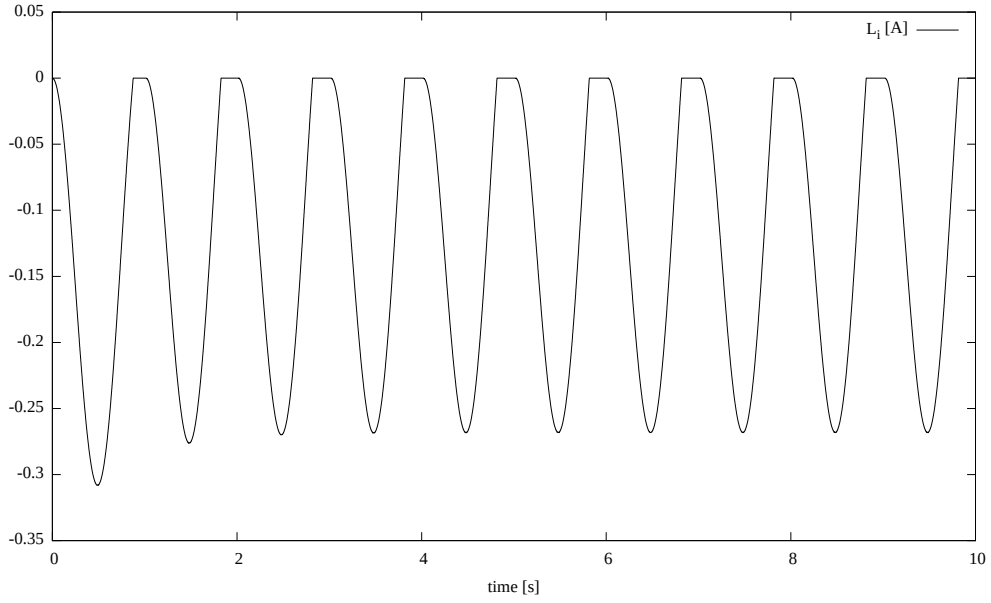

Figure 4.13: Current through the inductor in the half-wave rectifier circuit with in-line inductor.

Because of the expressivity of the *switch* combinator, in general, it is not possible to compile Hydra models prior to simulation. For example, given a parametrised signal relation $sr' :: \mathbb{R} \to SR\ \mathbb{R}$ and a signal function $sf :: SF\ \mathbb{R}\ \mathbb{R}$ one can recursively define a signal relation sr that describes an overall behaviour by "stringing together" the behaviours described by sr':

$$sr :: \mathbb{R} \to SR\ \mathbb{R}$$
$$sr\ x = switch\ (sr'\ x)\ sf\ sr$$

In this case, because the number of instantiations of sr' in general cannot be determined statically and because each instantiation can depend on the parameter in arbitrarily complex ways, there is no way to generate all code prior to simulation.

Perhaps the example involving the *sr* signal relation is a bit abstract. In the following sections we emphasise the same point by using a variation on the familiar bouncing-ball example. Assuming the elastic collision with the floor, the bouncing ball system can be modelled in Hydra as follows.

$$bouncingBall :: Body \to SR\ Body$$
$$bouncingBall\ b = switch\ (freeFall\ b)$$
$$[fun \mid ((\_, y), \_) \to y \mid]$$
$$(\lambda(p, (vx, vy)) \to bouncingBall\ (p, (vx, -vy))$$

This example involves stringing of the *bouncingBall* signal relation. But even here, in principal, it is possible to generate the code prior to simulation, because the active equations always remain the same; that is, only initial the condition is changing.

The following code models a variation of the bouncing ball example where the ball breaks at every collision with the floor.

$$
\begin{aligned}
&\mathit{bouncingBall'} :: \mathit{Body} \to \mathit{SR}\ \mathit{Body} \\
&\mathit{bouncingBall'}\ b = \mathit{switch}\ (\mathit{freeFall}\ b) \\
&\qquad\qquad\qquad\qquad [\mathit{fun} \mid ((\_, y), \_) \to y \mid] \\
&\qquad\qquad\qquad\qquad (\lambda(p, v) \to \mathit{divide}\ (p, v)) \\
&\mathit{divide} :: \mathit{Body} \to \mathit{SR}\ \mathit{Body} \\
&\mathit{divide}\ ((x0, y0), (vx0, vy0)) = [\mathit{rel} \mid ((x, y), (vx, vy)) \to \\
&\quad \$\ \mathit{bouncingBall'}\ ((x0, y0), (\ \ vx0\ /\ 2, -\ vy0\ /\ 2))\ \$\ \diamond\ ((x, y), (vx, vy)) \\
&\quad \mathbf{local}\ x'\ y'\ vx'\ vy' \\
&\quad \$\ \mathit{bouncingBall'}\ ((x0, y0), (-vx0\ /\ 2, -\ vy0\ /\ 2))\ \$\ \diamond\ ((x', y'), (vx', vy')) \\
&\mid]
\end{aligned}
$$

The model assumes that the kinetic energy is not lost and the balls divide the initial kinetic energy by bouncing in opposite directions. This is an example of a highly structurally dynamic system; the number of modes cannot be determined prior to simulation and it is not feasible to generate the code prior to simulation.

Unfortunately, due to the limitations of main-stream noncausal modelling languages, declarative equational modelling of (highly) structurally dynamic systems remains an elusive application. We believe the adoption of the Hydra features described in this chapter will remedy this unfortunate situation.

Chapter 5

Definition of Hydra

This is a technical chapter giving a formal definition of the Hydra language. Note that this chapters defines Hydra's signal-level sublanguage. The functional-level sublanguage is provided by Haskell. The definition of Haskell is given in the book by Peyton Jones [2003].

The language definition is given in four steps. Firstly, we define Hydra's lexical structure and concrete syntax by using regular expression and BNF notations, respectively. Secondly, we give Hydra's untyped abstract syntax as a Haskell algebraic data type (ADT) definition. Thirdly, we define Hydra's typed abstract syntax as a Haskell generalised algebraic data type (GADT) [Jones et al., 2006] definition and give a translation from the untyped abstract syntax to the typed abstract syntax. The typed representation fully embodies Hydra's type system and can be seen as a definition of Hydra's type system in terms of the Haskell type system. In other words, Hydra's type system is embedded into Haskell's type system. Finally, we give ideal denotational semantics of Hydra by giving meaning to the typed abstract syntax in terms of second-order logic.

5.1 Concrete Syntax

The syntactic structure of Hydra is given in Figure 5.1, which uses BNF notation. Nonterminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to BNF notation. All other symbols are terminals.

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ and ', reserved words excluded. The reserved words used in Hydra are **init** and **local**.

⟨*SigRel*⟩ ::= ⟨*Pattern*⟩ −> { ⟨*ListEquation*⟩ }

⟨*SigFun*⟩ ::= ⟨*Pattern*⟩ −> { ⟨*Expr*⟩ }

⟨*Pattern*⟩ ::= _
| ⟨*Ident*⟩
| `()`
| `(` ⟨*Pattern*⟩ `,` ⟨*Pattern*⟩ `)`

⟨*Equation*⟩ ::= ⟨*Expr*⟩ = ⟨*Expr*⟩
| `init` ⟨*Expr*⟩ = ⟨*Expr*⟩
| `local` ⟨*Ident*⟩
| ⟨*HsExpr*⟩ <> ⟨*Expr*⟩

⟨*Expr1*⟩ ::= ⟨*Expr1*⟩ + ⟨*Expr2*⟩
| ⟨*Expr1*⟩ − ⟨*Expr2*⟩
| ⟨*Expr2*⟩

⟨*Expr2*⟩ ::= ⟨*Expr2*⟩ `/` ⟨*Expr3*⟩
| ⟨*Expr2*⟩ `*` ⟨*Expr3*⟩
| ⟨*Expr3*⟩

⟨*Expr3*⟩ ::= ⟨*Expr3*⟩ `^` ⟨*Expr4*⟩
| − ⟨*Expr4*⟩
| ⟨*Expr4*⟩

⟨*Expr4*⟩ ::= ⟨*Expr4*⟩ ⟨*Expr5*⟩
| ⟨*Expr5*⟩

⟨*Expr5*⟩ ::= ⟨*Ident*⟩
| ⟨*HsExpr*⟩
| ⟨*Integer*⟩
| ⟨*Double*⟩
| `()`
| `(` ⟨*Expr*⟩ `,` ⟨*Expr*⟩ `)`
| `(` ⟨*Expr*⟩ `)`

⟨*Expr*⟩ ::= ⟨*Expr1*⟩

⟨*ListEquation*⟩ ::= ϵ
| ⟨*Equation*⟩
| ⟨*Equation*⟩ `;` ⟨*ListEquation*⟩

Figure 5.1: Syntactic structure of Hydra.

```
->   {    }
_    ()   (
,    )    =
<>   +    -
/    *    ^
;
```

Figure 5.2: Symbols used in Hydra.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits. Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + '.'⟨*digit*⟩ + ('e'-?⟨*digit*⟩+)?; that is, two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

HsExpr literals represent antiquoted Haskell expressions and are recognised by the regular expression '$'(⟨*anychar*⟩ − '$') ∗ '$'

The symbols used in Hydra are given in Figure 5.2. In Hydra, single-line comments begin with −− and multiple-line comments are enclosed with {− and −}.

5.2 Abstract Syntax

Hydra's abstract syntax is given in Figure 5.3. The ADT definition is derived from the concrete syntax defined in previous section. The representation is untyped; that is, it allows for terms that are syntactically correct but not necessarily type correct.

The data type *Ident* is used to represent identifiers, specifically, signal variable and built-in signal function identifiers. The data type *HsExpr* is used to represent antiquoted Haskell expressions.

The data type *SigRel* is used to represent signal relations. The data type has a single constructor. Given a pattern and a list of equations the constructor constructs the corresponding signal relation.

The data type *SigFun* is used to represent signal functions. The data type has a single constructor. Given a pattern and a signal expression the constructor constructs the corresponding signal function.

The data type *Pattern* is used to represent patterns that bind signal variables. There are four ways to construct a pattern. The constructor *PatWild* constructs the wild card pattern. Given an identifier the constructor *PatName* constructs a pattern that binds the corresponding single signal variable. The constructor *PatUnit* constructs the

```
data Ident    = Ident    String
data HsExpr = HsExpr String

data SigRel  = SigRel  Pattern [Equation]
data SigFun = SigFun Pattern Expr

data Pattern = PatWild
             | PatName Ident
             | PatUnit
             | PatPair   Pattern Pattern

data Equation = EquEqual       Expr    Expr
              | EquInit        Expr    Expr
              | EquLocal       Ident
              | EquSigRelApp HsExpr Expr

data Expr = ExprAdd     Expr Expr
          | ExprSub     Expr Expr
          | ExprDiv     Expr Expr
          | ExprMul     Expr Expr
          | ExprPow     Expr Expr
          | ExprNeg     Expr
          | ExprApp     Expr Expr
          | ExprVar     Ident
          | ExprAnti    HsExpr
          | ExprInteger Integer
          | ExprDouble  Double
          | ExprUnit
          | ExprPair    Expr Expr
```

Figure 5.3: Abstract syntax of Hydra.

$$
\begin{aligned}
&[\![\cdot]\!] && :: SigRel \to SigRel \\
&[\![SigRel\ pat\ eqs]\!] &&= SigRel\ pat\ (concat\ [[\![eq]\!]_{eq} \mid eq \leftarrow eqs])
\end{aligned}
$$

$$
\begin{aligned}
&[\![\cdot]\!]_{eq} && :: Equation \to [Equation] \\
&[\![EquEqual\ (Pair\ e_1\ e_2)\ (Pair\ e3\ e4)]\!]_{eq} &&= [\![EquEqual\ e_1\ e3]\!]_{eq} +\!\!+ [\![EquEqual\ e_2\ e4]\!]_{eq} \\
&[\![EquInit\ \ (Pair\ e_1\ e_2)\ (Pair\ e3\ e4)]\!]_{eq} &&= [\![EquInit\ \ e_1\ e3]\!]_{eq} +\!\!+ [\![EquInit\ \ e_2\ e4]\!]_{eq} \\
&[\![eq]\!]_{eq} &&= [eq]
\end{aligned}
$$

Figure 5.4: Desugaring translation of Hydra.

pattern that only matches unit signals. The constructor *PatPair* constructs a pattern that matches a pair of patterns.

The data type *Equation* is used to represent noncausal equations and local signal variable declarations. The constructor *EquEqual* constructs an equation that asserts equality of two signal expressions. The constructor *EquInit* constructs an initialisation equation that asserts equality of two signal expressions. The constructor *EquLocal* constructs a local variable declaration. The constructor *EquSigRelApp* constructs a signal relation application that applies the signal relation referred in the antiquoted Haskell expression to the given signal expression.

The data type *Expr* is used to represent signal expressions. Common mathematical operations, identifiers, antiquoted Haskell expressions, integer and real constants, unit signals, and pairs of signals can be used to construct signal expressions (see Figure 5.3 for details).

5.3 Desugaring

Before we turn our attention to the translation of the untyped abstract syntax into typed abstract syntax, we describe a translation that desugars all equations that assert equality of signal pairs into equations asserting equality of scalar signals. This translation allows for a simpler typed representation as we show in the following section. The translation is given in Figure 5.4 as a Haskell function working with the untyped abstract syntax of Hydra.

5.4 Typed Abstract Syntax

The typed abstract syntax that embodies the type system of Hydra is given in Figure 5.5 as a GADT definition. Note that the types *Signal* α and *PrimSF* α β are genuine GADTs, while the data types *SR* α, *SF* α β and *Equation* are ADTs that use the GADT notation for consistency.

5.5 From Untyped to Typed Abstract Syntax

The translation rules that transform a model in the untyped representation into the corresponding model in the typed representation are given in Figure 5.6 and Figure 5.7. These rules translate an untyped term into Haskell code that builds the corresponding typed term. The pattern matching semantics in the left-hand side of the translation rules are that of Haskell.

You may have noticed that there are no translation rules that generate the *Switch* constructor. The functional-level combinator *switch*, which was introduced in Chapter 4, generates the *Switch* constructor of the typed abstract syntax. Specifically, the *switch* combinator is defined as follows.

$$switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$$
$$switch = Switch$$

The typed abstract syntax embodies Hydra's type system features that were only informally introduced in earlier sections of the thesis. Let us outline several key features. The type of a signal relation is determined by its pattern. A type of a structurally dynamic signal relation remains unchanged despite the structural changes. Signal relation and signal function applications must be well typed. This includes the application of the built-in equality signal relation.

Note that the type system says nothing about the solvability of signal relations. It is possible to define a type correct signal relation that does not have a solution or has more than one solution. It is the responsibility of the modeller to define a signal relation that has an unique solution. Recent work in the context of the FHM framework has made progress in the direction of more expressive type systems incorporating the solvability aspect of noncausal models [Nilsson, 2008, Capper and Nilsson, 2010]. Incorporation of the aforementioned work in Hydra is a subject of future research.

$$
\begin{aligned}
&\textbf{data } \mathit{SR}\ \alpha \textbf{ where}\\
&\quad \mathit{SR} \quad :: (\mathit{Signal}\ \alpha \to [\mathit{Equation}]) \to \mathit{SR}\ \alpha\\
&\quad \mathit{Switch} :: \mathit{SR}\ \alpha \to \mathit{SF}\ \alpha\ \mathbb{R} \to (\alpha \to \mathit{SR}\ \alpha) \to \mathit{SR}\ \alpha\\
&\textbf{data } \mathit{SF}\ \alpha\ \beta \textbf{ where}\\
&\quad \mathit{SF} :: (\mathit{Signal}\ \alpha \to \mathit{Signal}\ \beta) \to \mathit{SF}\ \alpha\ \beta\\
&\textbf{data } \mathit{Equation} \textbf{ where}\\
&\quad \mathit{Local} \ :: (\mathit{Signal}\ \mathbb{R} \to [\mathit{Equation}]) \to \mathit{Equation}\\
&\quad \mathit{Equal} :: \mathit{Signal}\ \mathbb{R} \to \mathit{Signal}\ \mathbb{R} \to \mathit{Equation}\\
&\quad \mathit{Init} \quad :: \mathit{Signal}\ \mathbb{R} \to \mathit{Signal}\ \mathbb{R} \to \mathit{Equation}\\
&\quad \mathit{App} \quad :: \mathit{SR}\ \alpha \to \mathit{Signal}\ \alpha \to \mathit{Equation}\\
&\textbf{data } \mathit{Signal}\ \alpha \textbf{ where}\\
&\quad \mathit{Unit} \quad :: \mathit{Signal}\ ()\\
&\quad \mathit{Time} \quad :: \mathit{Signal}\ \mathbb{R}\\
&\quad \mathit{Const} \quad :: \alpha \to \mathit{Signal}\ \alpha\\
&\quad \mathit{Pair} \quad :: \mathit{Signal}\ \alpha \to \mathit{Signal}\ \beta \to \mathit{Signal}\ (\alpha, \beta)\\
&\quad \mathit{PrimApp} :: \mathit{PrimSF}\ \alpha\ \beta \to \mathit{Signal}\ \alpha \to \mathit{Signal}\ \beta\\
&\textbf{data } \mathit{PrimSF}\ \alpha\ \beta \textbf{ where}\\
&\quad \mathit{Der} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Exp} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Sqrt} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Log} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Sin} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Tan} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Cos} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Asin} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Atan} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Acos} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Sinh} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Tanh} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Cosh} \quad :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Asinh} :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Atanh} :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Acosh} :: \mathit{PrimSF}\ \mathbb{R}\ \mathbb{R}\\
&\quad \mathit{Add} \quad :: \mathit{PrimSF}\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}\\
&\quad \mathit{Mul} \quad :: \mathit{PrimSF}\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}\\
&\quad \mathit{Div} \quad :: \mathit{PrimSF}\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}\\
&\quad \mathit{Pow} \quad :: \mathit{PrimSF}\ (\mathbb{R}, \mathbb{R})\ \mathbb{R}
\end{aligned}
$$

Figure 5.5: Typed intermediate representation of Hydra.

$$
\begin{aligned}
[\![SigRel\ \ pattern\ equations]\!]_{sr} &= SR\ (\lambda [\![pattern]\!]_{pat} \rightarrow [\![equations]\!]_{eqs}) \\
[\![SigFun\ pattern\ expression]\!]_{sf} &= SF\ (\lambda [\![pattern]\!]_{pat} \rightarrow [\![expression]\!]_{exp})
\end{aligned}
$$

$$
\begin{aligned}
[\![PatWild]\!]_{pat} &= \_ \\
[\![PatName\ (Ident\ s)]\!]_{pat} &= [\![s]\!]_{hs} \\
[\![PatUnit]\!]_{pat} &= Unit \\
[\![PatPair\ pat_1\ pat_2]\!]_{pat} &= Pair\ [\![pat_1]\!]_{pat}\ [\![pat_2]\!]_{pat}
\end{aligned}
$$

$$
\begin{aligned}
[\![[\,]]\!]_{eqs} &= [\,] \\
[\![(EquSigRelApp\ (HsExpr\ s)\ e) : eqs]\!]_{eqs} &= (App\ \ [\![s]\!]_{hs}\ \ [\![e]\!]_{exp}) : [\![eqs]\!]_{eqs} \\
[\![(EquEqual\ e_1\ e_2) : eqs]\!]_{eqs} &= (Equal\ [\![e_1]\!]_{exp}\ [\![e_2]\!]_{exp}) : [\![eqs]\!]_{eqs} \\
[\![(EquInit\ \ e_1\ e_2) : eqs]\!]_{eqs} &= (Init\ \ [\![e_1]\!]_{exp}\ [\![e_2]\!]_{exp}) : [\![eqs]\!]_{eqs} \\
[\![(EquLocal\ (Ident\ s)) : eqs]\!]_{eqs} &= [\,Local\ \ (\lambda [\![s]\!]_{hs} \rightarrow [\![eqs]\!]_{eqs})\,]
\end{aligned}
$$

Figure 5.6: Translation of untyped signal functions and signal relations into typed signal functions and signal relations. The translation rule $[\![\cdot]\!]_{hs}$ takes a string in the concrete syntax of Haskell and generates the corresponding Haskell code. The translation rules $[\![\cdot]\!]_{exp}$ and $[\![\cdot]\!]_{ident}$ are given in Figure 5.7.

5.6 Ideal Denotational Semantics

A formal language definition has a number of advantages over an informal presentation. A formal semantics does not leave room for ambiguity and allows different implementers to implement the same language. In addition, a formally defined semantics paves the way for proving useful statements about the language.

There are a number of different approaches to the specification of formal semantics. Two most widely used approaches are operational semantics [Plotkin, 2004] and denotation semantics [Scott, 1982]. An operational semantics formally defines an abstract machine and how the language terms are executed on the machine. A denotational semantics formally defines translation of the language terms into terms in a formalism that is well understood (often a field of mathematics).

One characteristic of noncausal modelling languages setting them apart from traditional programming languages is that concrete implementations of noncausal languages only aim to approximate the model defined at the source level. For example, consider the system of equations modelling the simple electrical circuit given in Chapter 2. In the process of deriving the simulation code we introduced a number of approximations. The continuous real numbers were approximated using the double-precision machine floating-point numbers and the system of equations was approximated using the Haskell code implementing the forward Euler method.

$$
\begin{aligned}
[\![\mathit{ExprAnti}\ (\mathit{HsExpr}\ s_1)]\!]_{exp} &= \mathit{Const}\ [\![s_1]\!]_{hs} \\
[\![\mathit{ExprVar}\ (\mathit{Ident}\ s)]\!]_{exp} &= [\![s_1]\!]_{ident} \\
[\![\mathit{ExprAdd}\ e_1\ e_2]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Add}\ (\mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_1]\!]_{exp}) \\
[\![\mathit{ExprSub}\ e_1\ e_2]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Sub}\ (\mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_1]\!]_{exp}) \\
[\![\mathit{ExprDiv}\ e_1\ e_2]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Div}\ (\mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_1]\!]_{exp}) \\
[\![\mathit{ExprMul}\ e_1\ e_2]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Mul}\ (\mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_1]\!]_{exp}) \\
[\![\mathit{ExprPow}\ e_1\ e_2]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Pow}\ (\mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_1]\!]_{exp}) \\
[\![\mathit{ExprNeg}\ e_1]\!]_{exp} &= \mathit{PrimApp}\ \mathit{Neg}\ [\![e_1]\!]_{exp} \\
[\![\mathit{ExprApp}\ (\mathit{ExprAnti}\ (\mathit{HsExpr}\ s))\ e]\!]_{exp} &= (\mathbf{case}\ [\![s]\!]_{hs}\ \mathbf{of}\ \mathit{SF}\ f \to f)\ [\![e]\!]_{exp} \\
[\![\mathit{ExprApp}\ e_1\ e_2]\!]_{exp} &= [\![e_1]\!]_{exp}\ [\![e_2]\!]_{exp} \\
[\![\mathit{ExprInteger}\ i_1]\!]_{exp} &= \mathit{Const}\ (\mathit{fromIntegral}\ i_1) \\
[\![\mathit{ExprDouble}\ d_1]\!]_{exp} &= \mathit{Const}\ d_1 \\
[\![\mathit{ExprUnit}]\!]_{exp} &= \mathit{Unit} \\
[\![\mathit{ExprPair}\ e_1\ e_2]\!]_{exp} &= \mathit{Pair}\ [\![e_1]\!]_{exp}\ [\![e_2]\!]_{exp}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathit{Ident}\ \texttt{"time"}]\!]_{ident} &= \mathit{Time} \\
[\![\mathit{Ident}\ \texttt{"der"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Der} \\
[\![\mathit{Ident}\ \texttt{"exp"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Exp} \\
[\![\mathit{Ident}\ \texttt{"sqrt"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Sqrt} \\
[\![\mathit{Ident}\ \texttt{"log"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Log} \\
[\![\mathit{Ident}\ \texttt{"sin"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Sin} \\
[\![\mathit{Ident}\ \texttt{"tan"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Tan} \\
[\![\mathit{Ident}\ \texttt{"cos"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Cos} \\
[\![\mathit{Ident}\ \texttt{"asin"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Asin} \\
[\![\mathit{Ident}\ \texttt{"atan"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Atan} \\
[\![\mathit{Ident}\ \texttt{"acos"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Acos} \\
[\![\mathit{Ident}\ \texttt{"sinh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Sinh} \\
[\![\mathit{Ident}\ \texttt{"tanh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Tanh} \\
[\![\mathit{Ident}\ \texttt{"cosh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Cosh} \\
[\![\mathit{Ident}\ \texttt{"asinh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Asinh} \\
[\![\mathit{Ident}\ \texttt{"atanh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Atanh} \\
[\![\mathit{Ident}\ \texttt{"acosh"}]\!]_{ident} &= \mathit{PrimApp}\ \mathit{Acosh} \\
[\![\mathit{Ident}\ s]\!]_{ident} &= [\![s]\!]_{hs}
\end{aligned}
$$

Figure 5.7: Translation of untyped signal expressions into typed signal expressions.

Implementations of noncausal modelling languages allow modellers to choose floating-point representations (e.g., single or double precision), symbolic processing methods and numerical simulation methods that needs to be used during the simulation. This amounts to allowing modellers to choose a combination of approximations prior to simulation.

The fact that the implementations are only expected to approximate noncausal models needs to be taken into account when defining a formal semantics for a noncausal language. In particular, definition of operational semantics is problematic as it is hard to account for the myriad of approximation combinations that were outlined earlier. One option is to parameterise the operational semantics on approximations. This is feasible, but leaves the bulk of operational details unspecified defeating the purpose of an operational semantics.

For the reasons outlined above, and because the concept of first-class models, which allows for higher-order and structurally dynamic modelling, is not predicated on particular approximations used during simulation, we opted to use *ideal* denotational semantics for formally defining the Hydra language. By referring to the semantics as ideal, we emphasise that concrete implementations are only expected to approximate the denotational semantics.

The primary goal of the denotational semantics that is given in this section is to precisely and concisely communicate Hydra's definition to modelling language designers and implementers, in order to facilitate incorporation of Hydra's key features in other noncausal modelling languages.

Although not considered in this thesis, the ideal denotational semantics of Hydra can also be used to verify concrete implementations of Hydra with certain approximations. In addition, the denotational semantics can be used to check whether concrete simulation results correspond to the source-level noncausal model, again under certain approximation; for example, by using the absolute error tolerance of the numerical simulation. These two applications of the ideal denotational semantics are subjects of future work.

The ideal denotational semantics of Hydra are given in Figure 5.8 and in Figure 5.9. Note that the domains of the denotational semantics are the same as the conceptual definitions of signals, signal functions, and signal relations given in Chapter 4. Specifically, signal relations are mapped to functions from starting time and signal to second-order

$$
\begin{array}{l}
[\![SR\ f]\!]_{sr} \qquad\qquad\qquad = \lambda t_1\ t_2\ s \to [\![(0, t_1, t_2, f\ s)]\!]_{eqs} \\
[\![Switch\ sr\ sf\ f]\!]_{sr} = \lambda t_1\ t_2\ s \to \\
\quad ([\![sr]\!]_{sr}\ t_1\ t_2\ s) \wedge (\forall\, t \in \mathbb{R}.\, t_1 < t \leqslant t_2 \Rightarrow \neg\, [\![(sf, s, t)]\!]_{zc}) \\
\quad \vee \\
\quad (\exists\, t_e \in \mathbb{R}.\, (t_1 < t_e \leqslant t_2) \\
\qquad\qquad \wedge \\
\qquad\qquad ([\![sr]\!]_{sr}\ t_1\ t_e\ s) \wedge [\![(sf, s, t_e)]\!]_{zc} \wedge (\forall\, t \in \mathbb{R}.\, t_1 < t < t_e \Rightarrow \neg\, [\![(sf, s, t)]\!]_{zc}) \\
\qquad\qquad \wedge \\
\qquad\qquad ([\![f\ (s\ t_e)]\!]_{sr}\ t_e\ t_2\ s))
\end{array}
$$

$$
[\![(sf, s, t)]\!]_{zc} = [\![sf]\!]_{sf}\ s\ t \equiv 0 \wedge \tfrac{d\_}{dt}([\![sf]\!]_{sf}\ s)\ t \not\equiv 0
$$

$$
[\![SF\ sf]\!]_{sf} = sf
$$

$$
\begin{array}{lll}
[\![\_, \_, \_, [\,]]\!]_{eqs} & & = \top \\
[\![i\,, t_1, t_2, (Local\ f) & : eqs]\!]_{eqs} & = (\exists\, s_i \in \mathbb{R} \to \mathbb{R}.\, [\![i+1, t_0, f\ s_i +\!\!+ eqs]\!]_{eqs}) \\
[\![i\,, t_1, t_2, (App\ sr\ s) & : eqs]\!]_{eqs} & = ([\![sr]\!]_{sr}\ t_0\ s) \wedge [\![i, t_0, eqs]\!]_{eqs} \\
[\![i\,, t_1, t_2, (Equal\ s_1\ s_2) & : eqs]\!]_{eqs} & = \\
\quad (\forall\, t \in \mathbb{R}.\, (t \geqslant t_1 \wedge t \leqslant t_2) & \Rightarrow [\![s_1]\!]_{sig}\ t \equiv [\![s_2]\!]_{sig}\ t) \wedge [\![i, t_0, eqs]\!]_{eqs} & \\
[\![i\,, t_1, t_2, (Init\ s_1\ s_2) & : eqs]\!]_{eqs} & = \\
\quad (\forall\, t \in \mathbb{R}.\, (t \equiv t_1) & \Rightarrow [\![s_1]\!]_{sig}\ t \equiv [\![s_2]\!]_{sig}\ t) \wedge [\![i, t_0, eqs]\!]_{eqs} &
\end{array}
$$

Figure 5.8: Denotations for signal relations, signal functions and equations.

logic proposition, signal functions are mapped to functions from signal to signal, and signals are mapped to function from time to value. Time is represented as a real number.

A signal relation denotation may involve existentially quantified function symbols (i.e., signals). This is what makes the denotations second-order logic propositions (i.e., not expressible in first-order logic). In other words, solving of a signal relation can be understood as proving of *existence* of signals that satisfy the given constraints (see Figure 5.8 for details).

$$
\begin{aligned}
[\![Unit]\!]_{sig} &= \lambda\_ \to () \\
[\![Time]\!]_{sig} &= \lambda t \to t \\
[\![Const\ d]\!]_{sig} &= \lambda\_ \to d \\
[\![Pair\ s_1\ s_2]\!]_{sig} &= \lambda t \to ([\![s_1]\!]_{sig}\ t, [\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Der\ s]\!]_{sig} &= \lambda t \to \tfrac{d}{dt}[\![s]\!]_{sig}\ t \\
[\![PrimApp\ Exp\ s]\!]_{sig} &= \lambda t \to exp\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sqrt\ s]\!]_{sig} &= \lambda t \to sqrt\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Log\ s]\!]_{sig} &= \lambda t \to log\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sin\ s]\!]_{sig} &= \lambda t \to sin\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Tan\ s]\!]_{sig} &= \lambda t \to tan\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Cos\ s]\!]_{sig} &= \lambda t \to cos\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Asin\ s]\!]_{sig} &= \lambda t \to asin\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Atan\ s]\!]_{sig} &= \lambda t \to atan\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Acos\ s]\!]_{sig} &= \lambda t \to acos\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Sinh\ s]\!]_{sig} &= \lambda t \to sinh\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Tanh\ s]\!]_{sig} &= \lambda t \to tanh\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Cosh\ s]\!]_{sig} &= \lambda t \to cosh\ \ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Asinh\ s]\!]_{sig} &= \lambda t \to asinh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Atanh\ s]\!]_{sig} &= \lambda t \to atanh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Acosh\ s]\!]_{sig} &= \lambda t \to acosh\ ([\![s]\!]_{sig}\ t) \\
[\![PrimApp\ Add\ \ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \to ([\![s_1]\!]_{sig}\ t) + ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Mul\ \ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \to ([\![s_1]\!]_{sig}\ t) * ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Div\ \ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \to ([\![s_1]\!]_{sig}\ t)\ /\ ([\![s_2]\!]_{sig}\ t) \\
[\![PrimApp\ Pow\ (Pair\ s_1\ s_2)]\!]_{sig} &= \lambda t \to ([\![s_1]\!]_{sig}\ t) \uparrow ([\![s_2]\!]_{sig}\ t)
\end{aligned}
$$

Figure 5.9: Denotations for signals.

Chapter 6

Implementation of Hydra

This chapter describes how Hydra is embedded in Haskell and how embedded noncausal models are simulated. Performance of the simulator is evaluated by focussing on the implementation aspects that are absent from main-stream noncausal modelling language implementations (i.e., runtime symbolic processing and JIT compilation).

6.1 Embedding

Hydra is implemented as a Haskell embedded DSL. We use quasiquoting, a recent Haskell extension implemented in Glasgow Haskell Compiler (GHC)[1], to provide a convenient surface (i.e., concrete) syntax for Hydra. The implementation uses quasiquoting to generate the typed representation of Hydra models from strings in the concrete syntax. An opening quasiquote specifies a function (a so-called quasiquoter) that performs the aforementioned transformation. In GHC, a quasiquoter generates Haskell code using Template Haskell [Sheard and Jones, 2002], a compile-time meta-programming facility implemented in GHC.

GHC executes quasiquoters at Haskell compile time, before type checking. As the typed abstract syntax of Hydra fully embodies the type system of Hydra, we effectively delegate the task of type checking to the host language type checker. This approach reduces the language specification and implementation effort by reusing the host language type system and the host language type checker. However, the disadvantage of this approach is the fact that typing errors are not domain specific.

[1] `http://www.haskell.org/ghc`

The implementation of Hydra provides two quasiquoters: the *rel* quasiquoter for generating typed signal relations, and the *fun* quasiquoter for generating typed signal functions. The implementation of the quasiquoters is broken down into three stages: parsing, desugaring and translation into the typed abstract syntax.

Firstly, the string in the concrete syntax of Hydra is parsed and the corresponding untyped representation is generated as an abstract syntax tree (AST). The BNF Converter (BNFC), a compiler front-end generator from a labelled BNF grammar [Pellauer et al., 2004], is used to generate the parser and the AST data type. The labelled BNF grammar of Hydra is given in Figure 6.1. The generated AST data type and the syntax that the generated parser implements are exactly the same as given in the language definition in Chapter 5. In addition, we use BNFC to generate Hydra's layout resolver allowing for a list of equations in *rel* quasiquotes to be constructed without curly braces and semicolons. The layout rules are the same as for Haskell.

Secondly, the untyped representation is desugared exactly as it is presented in the language definition (see Chapter 5).

Finally, the desugared untyped representation is translated into the typed representation. This step implements the corresponding translation rules given in Chapter 5. Note that the translation rules generate Haskell code. This is implemented by using the Template Haskell facility of GHC.

We illustrate the quasiquoting process by using a signal relation that models a parametrised van der Pol oscillator. The oscillator model is given in Figure 6.2. After the parsing stage the quasiquoted signal relation is turned into the AST that is given in Figure 6.3. After the desugaring stage we get the AST that is given in Figure 6.4. After translation into the typed representation we get the typed AST that is given in Figure 6.5.

Let us briefly overview the typed abstract syntax used in the implementation of Hydra. This is to highlight a minor difference from the typed abstract syntax presented in the language definition and to draw your attention to the mixed-level embedding techniques used in the implementation.

The typed abstract syntax allows for two ways to form a signal relation: either from equations that constrain a given signal, or by composing two signal relations temporally:

```
entrypoints SigRel, SigFun;

SigRel. SigRel ::= Pattern "->" "{" [Equation] "}" ;

SigFun. SigFun ::= Pattern "->" "{" Expr       "}" ;

PatWild. Pattern ::= "_" ;
PatName. Pattern ::= Ident ;
PatUnit. Pattern ::= "()" ;
PatPair. Pattern ::= "(" Pattern "," Pattern ")" ;

EquEqual.     Equation ::=        Expr "=" Expr ;
EquInit.      Equation ::= "init" Expr "=" Expr ;
EquLocal.     Equation ::= "local" Ident;
EquSigRelApp. Equation ::= HsExpr "<>" Expr ;

ExprAdd.     Expr1 ::= Expr1 "+"  Expr2 ;
ExprSub.     Expr1 ::= Expr1 "-"  Expr2 ;
ExprDiv.     Expr2 ::= Expr2 "/"  Expr3 ;
ExprMul.     Expr2 ::= Expr2 "*"  Expr3 ;
ExprPow.     Expr3 ::= Expr3 "^"  Expr4 ;
ExprNeg.     Expr3 ::= "-" Expr4 ;
ExprApp.     Expr4 ::= Expr4 Expr5 ;
ExprVar.     Expr5 ::= Ident ;
ExprAnti.    Expr5 ::= HsExpr ;
ExprInteger. Expr5 ::= Integer ;
ExprDouble.  Expr5 ::= Double ;
ExprUnit.    Expr5 ::= "()" ;
ExprPair.    Expr5 ::= "(" Expr "," Expr ")" ;
_.           Expr  ::= Expr1 ;
_.           Expr1 ::= Expr2 ;
_.           Expr2 ::= Expr3 ;
_.           Expr3 ::= Expr4 ;
_.           Expr4 ::= Expr5 ;
_.           Expr5 ::= "(" Expr ")" ;

separator Equation ";" ;

comment "--" ;
comment "{-" "-}" ;

token HsExpr ('$' (char - '$')* '$') ;

layout "->" ;
```

Figure 6.1: Labelled BNF grammar of Hydra. This labelled BNF grammar is used to generate Hydra's parser, untyped abstract syntax and layout resolver.

```
vanDerPol :: ℝ → SR ()
vanDerPol μ = [rel | () →
  local x y
  init (x, y) = (1, 1)
  der x = y
  der y = − x + $μ $ *(1 − x * x) * y
|]
```

Figure 6.2: Signal relation modelling a parametrised van der Pol oscillator.

```
SigRel PatUnit
    [ EquLocal (Ident "x") [Ident "y"]
    , EquInit  (ExprPair (ExprVar (Ident "x")) (ExprVar (Ident "y")))
               (ExprPair (ExprInteger 1) (ExprInteger 1))
    , EquEqual (ExprApp (ExprVar (Ident "der")) (ExprVar (Ident "x")))
               (ExprVar (Ident "y"))
    , EquEqual (ExprApp (ExprVar (Ident "der")) (ExprVar (Ident "y")))
               (ExprAdd (ExprNeg (ExprVar (Ident "x")))
                        (ExprMul (ExprMul (ExprAnti (HsExpr "$mu$"))
                                          (ExprSub (ExprInteger 1)
                                                   (ExprMul
                                                     (ExprVar (Ident "x"))
                                                     (ExprVar (Ident "x")))))
                                 (ExprVar (Ident "x"))))
    ]
```

Figure 6.3: Untyped abstract syntax tree representing the *vanDerPol* signal relation.

```
SigRel PatUnit
    [ EquLocal (Ident "x") []
    , EquLocal (Ident "y") []
    , EquInit  (ExprVar (Ident "x")) (ExprInteger 1)
    , EquInit  (ExprVar (Ident "y")) (ExprInteger 1)
    , EquEqual (ExprApp (ExprVar (Ident "der")) (ExprVar (Ident "x")))
               (ExprVar (Ident "y"))
    , EquEqual (ExprApp (ExprVar (Ident "der")) (ExprVar (Ident "y")))
               (ExprAdd (ExprNeg (ExprVar (Ident "x")))
                        (ExprMul (ExprMul (ExprAnti (HsExpr "$mu$"))
                                          (ExprSub (ExprInteger 1)
                                                   (ExprMul
                                                     (ExprVar (Ident "x"))
                                                     (ExprVar (Ident "x")))))
                                 (ExprVar (Ident "x"))))
    ]
```

Figure 6.4: Desugared, untyped abstract syntax tree representing the *vanDerPol* signal relation.

```
SR (λ() →
      [Local (λx →
        [Local (λy →
          [ Init x (Const 1.0)
          , Init y (Const 1.0)
          , Equal (PrimApp Der x) y
          , Equal (PrimApp Der y)
                  (PrimApp
                    Add
                    (Pair (PrimApp Neg x)
                          (PrimApp
                            Mul
                            (Pair (PrimApp
                                    Mul
                                    (Pair (Const μ)
                                          (PrimApp
                                            Sub
                                            (Pair (Const 1.0)
                                                  (PrimApp Mul (Pair x x)))))))
                                  y)))
          ])])])
```

Figure 6.5: Typed abstract syntax tree representing the *vanDerPol* signal relation.

data $SR\ a$ **where**

$$SR \quad :: (Signal\ a \to [Equation]) \to SR\ a$$

$$Switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$$

The constructor *SR* forms a signal relation from a function that takes a signal and returns a list of equations constraining the given signal. This list of equations constitutes a system of DAEs that defines the signal relation by expressing constraints on the signal. The system of equations is not necessarily a static one as the equations may refer to signal relations that contain switches.

The *switch* combinator, which was introduced in Chapter 4, forms a signal relation by temporal composition of two signal relations. Internally, in the implementation of Hydra, such a temporal composition is represented by a signal relation formed by the *Switch* constructor:

$$switch :: SR\ a \to SF\ a\ \mathbb{R} \to (a \to SR\ a) \to SR\ a$$

$$switch = Switch$$

Note that, in the implementation of Hydra, the type $\mathbb{R}$ is a type synonym of *Double*, which is a standard double-precision floating-point type of Haskell.

There are four kinds of equations:

```
data Equation where
  Local :: (Signal ℝ → [Equation]) → Equation
  Equal :: Signal ℝ → Signal ℝ → Equation
  Init  :: Signal ℝ → Signal ℝ → Equation
  App   :: SR a → Signal a → Equation
```

The *Local* constructor forms equations that merely introduce local signals. As it is evident from the language definition, such signals can be constrained only by the equations that are returned by the function that is the first argument of the *Local* constructor. In contrast, equation generating functions in the *SR* constructor are allowed to be passed a signal that is constrained elsewhere. This distinction is enforced by the language implementation, as we will see later in this chapter.

Initialisation equations, formed by the *Init* constructor, state initial conditions. They are only in force when a signal relation instance first becomes active.

Equations formed by the *Equal* constructor are basic equations imposing the constraint that the valuations of the two signals have to be equal for as long as the signal relation instance that contains the equation is active.

The fourth kind of equation is signal relation application, *App*; that is, an equation such as $sr \diamond (x, y + 2)$. The application constrains the given signals by the equations defined by the signal relation.

The following code defines the typed representation of signals used in the implementation of Hydra:

```
data Signal a where
  Unit    :: Signal ()
  Time    :: Signal ℝ
  Const   :: a → Signal a
  Pair    :: Signal a → Signal b → Signal (a, b)
  PrimApp :: PrimSF a b → Signal a → Signal b
  Var     :: Integer → Signal ℝ
```

As you can see, this data type definition contains one constructor that is not featured in the language definition, namely the *Var* constructor. This constructor is not used at the stage of quasiquoting. Instead, the constructor is used later at the stage of runtime symbolic processing to instantiate each local signal variable to an unique signal variable by using the constructor's *Integer* field.

The implementation of Hydra supports the same set of primitive functions as defined in the language definition. Hence, in the implementation we use the same *PrimSF* data type as given in the language definition.

The implementation of Hydra uses a mixture of shallow and deep techniques of embedding. The embedded functions in the *SR*, *Switch*, *Local* and *App* constructors correspond to the shallow part of the embedding. The rest of the data constructors, namely, *Equal*, *Init*, and all constructors of the *Signal* data type correspond to the deep part of the embedding, providing an explicit representation of language terms for further symbolic processing and ultimately compilation. As we will see in more detail below, each mode of operation can be described as a flat list of equations where each equation is constructed, either, by the *Init* constructor or by the *Equal* constructor. It is this representation that allows for generation of efficient simulation code. This combination of the two embedding techniques allowed us to leverage shallow embedding for high-level aspects of the embedded language, such as equation generation and temporal composition, and deep embedding for low-level aspects of the embedded language, such as simulation code generation for efficiency.

6.2 Simulation

In this section we describe how iteratively staged Hydra models are simulated. The process is conceptually divided into three stages as illustrated in Figure 6.6. In the first stage, a signal relation is flattened and subsequently transformed into a mathematical representation suitable for numerical simulation. In the second stage, this representation is JIT compiled into efficient machine code. In the third stage, the compiled code is passed to a numerical solver that simulates the system until the end of simulation or an event occurrence. In the case of an event occurrence, the process is repeated from the first stage by staring the new iteration.

Before we describe the three stages of the simulation in detail, let us briefly overview a function that performs these three stages. The simulator performs the aforementioned three stages iteratively at each structural change. A function that performs simulation has the following type signature:

$$simulate :: SR\ () \rightarrow Experiment \rightarrow IO\ ()$$

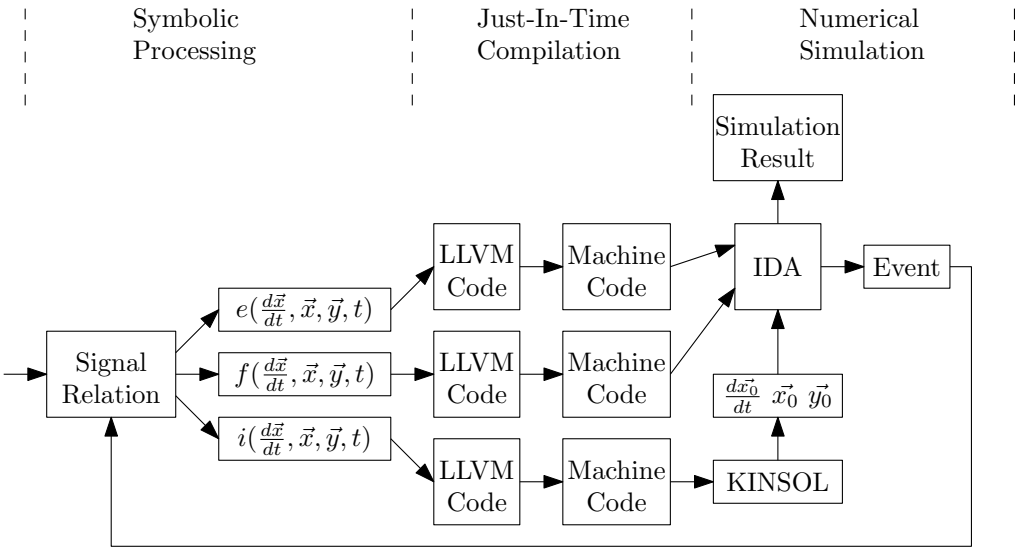

Figure 6.6: Execution model of Hydra.

The function takes a signal relation and an experiment description and simulates the system. The *Experiment* data type is defined in Figure 6.7. The *timeStart* field specifies the simulation starting time. The *timeStop* field specifies the simulation stopping time. The *timeStep* field specifies the simulation time step. The *symbolicProcessor* specifies the simulator's runtime symbolic processor. The *numericalSolver* specifies the simulator's numerical solver. The *trajectoryVisualiser* specifies how to visualise the simulation results (i.e., change of signal values over time). The data type definitions for *SymTab*, *NumericalSover* and *TrajectoryVisualiser* are given later in this chapter.

The implementation of Hydra provides the default experiment configuration that is given in Figure 6.8. Note that the last three fields of the experiment description record are expected to be modified by expert users willing to provide their own runtime symbolic processor, numerical solvers, or trajectory visualisers. The behaviour of the *defaultSymbolicProcessor*, *defaultNumericalSolver* and *defaultTrajectoryVisualiser* are described in detail later in this chapter.

6.3 Symbolic Processing

In this section we describe the first stage performed by the simulator: symbolic processing. A symbolic processor is a function from a symbol table to a symbol table. The symbol table data type that is used in the implementation of Hydra is given in Figure 6.9. The symbol table record has five fields.

```
data Experiment = Experiment {
    timeStart           :: ℝ
  , timeStop            :: ℝ
  , timeStep            :: ℝ
  , symbolicProcessor   :: SymTab → SymTab
  , numericalSolver     :: NumericalSolver
  , trajectoryVisualiser :: TrajectoryVisualiser
  }
```

Figure 6.7: Data type for experiment descriptions.

```
defaultExperiment :: Experiment
defaultExperiment = Experiment {
    timeStart           = 0
  , timeStop            = 10
  , timeStep            = 0.001
  , symbolicProcessor   = defaultSymbolicProcessor
  , numericalSolver     = defaultNumericalSolver
  , trajectoryVisualiser = defaultTrajectoryVisualiser
  }
```

Figure 6.8: Default experiment description.

The *model* field is for a top-level signal relation that is active. At the start of the simulation, the *simulate* function places application of its first argument of type *SR* () to the *Unit* signal. In other words, the *model* field contains currently active system of hierarchical equations that contains signal relation applications and temporal compositions.

The *equations* field is for a flat list of equations that describe an active mode of operation. By flat we mean that the list of equations only contain *Init* and *Equal* equations. At the start of the simulation, the *simulate* function places an empty list in this field.

The *events* field is for a list of zero-crossing signals defining the event occurrences. Recall the type signature of the *switch* combinator given in Section 6.1. A signal function that detects events returns a real valued signal. The simulator places the signal expressions that describe an event occurrence at each structural change. Initially, at the start of the simulation, the simulator places an empty list in the *events* field of the symbol table.

The *time* field is for current time. Initially the simulator places the starting time given in the experiment description in this field. The *time* fieled is modified at each structural change with the time of an event occurrence.

```
data SymTab = SymTab {
    model     :: [Equation]
  , equations :: [Equation]
  , events    :: [Signal ℝ]
  , time      :: ℝ
  , instants  :: Array Integer (ℝ, ℝ)
  }
```

Figure 6.9: Data type for symbol tables.

The *instants* field is for storing instantaneous values of signals. The simulator stores instantaneous values of active signals at each structural change. The instantaneous real values are stored as an array of pairs of reals. The first field is for storing the instantaneous signal values, while the second field is for storing the instantaneous values of signal differentials.

The task of the symbolic processor is to handle occurred events by modifying the *model* field of the symbol table, to generate a flat list of events that may occur in the active mode of operation by updating the *events* field of the symbol table, and to generate the flat list of equations describing the active mode of operation by updating the *equations* field of the symbol table. The implementation of Hydra provides the default symbolic processor that is defined as follows:

$$defaultSymbolicProcessor :: SymTab \rightarrow SymTab$$
$$defaultSymbolicProcessor = flattenEquations \circ flattenEvents \circ handleEvents$$

The default symbolic processor is defined as a composition of three symbolic processing steps. The first step handles occurred events by modifying the *model* field of the symbol table. The event handler is defined in Figure 6.10. The second step generates a list of signal expressions representing the list of possible events in the active mode of operation as defined in Figure 6.11. Note that this step involves evaluation of the instantaneous signal values by using the $[\![\cdot]\!]_{sig}$ function. The $[\![\cdot]\!]_{sig}$ function is defined in Figure 6.12. The third step flattens the hierarchical system of equations placed in the *model* field of the symbol table into the *equations* field of the symbol table. The flat list only contains *Init* and *Equal* equations. The *Equal* equations define the DAE that describes the active mode of operation. The *Init* equations describe the initial conditions for the DAE. The flattening transformation is given in Figure 6.13.

Each of the three steps of the default symbolic processor has a compact and self-explanatory definition, especially, the *flattenEquations* function. To my knowledge, this

$$
\begin{array}{l}
\mathit{handleEvents} \quad :: \mathit{SymTab} \to \mathit{SymTab} \\
\mathit{handleEvents}\ \mathit{st} = \mathit{st}\ \{\mathit{model} = [\![(\mathit{symtab}, \mathit{events}\ \mathit{st}, \mathit{model}\ \mathit{st})]\!]_{\mathit{eqs}}\}
\end{array}
$$

$$
\begin{array}{ll}
[\![\cdot]\!]_{\mathit{eqs}} :: (\mathit{SymTab}, [\mathit{Signal}\ \mathbb{R}], [\mathit{Equation}]) \to [\mathit{Equation}] & \\
[\![(\_, \_, [\,])]\!]_{\mathit{eqs}} & = [\,] \\
[\![(\mathit{st}, \mathit{evs}, (\mathit{Equal}\ \_\ \_) : \mathit{eqs})]\!]_{\mathit{eqs}} & = \mathit{eq} : [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(\mathit{st}, \mathit{evs}, (\mathit{Init}\quad \_\ \_) : \mathit{eqs})]\!]_{\mathit{eqs}} & = [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(\mathit{st}, \mathit{evs}, (\mathit{Local}\ f) : \mathit{eqs})]\!]_{\mathit{eqs}} & = \\
\quad \mathit{Local}\ (\lambda s \to [\![(\mathit{st}, \mathit{evs}, f\ s)]\!]_{\mathit{eqs}}) : [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} & \\
[\![(\mathit{st}, \mathit{evs}, (\mathit{App}\ (\mathit{SR}\ \mathit{sr})\ s_1\ f) : \mathit{eqs})]\!]_{\mathit{eqs}} & = \\
\quad \mathit{App}\ (\mathit{SR}\ (\lambda s_2 \to [\![(\mathit{st}, \mathit{evs}, f\ s_2)]\!]_{\mathit{eqs}}))\ s_1 : [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} & \\
[\![(\mathit{st}, \mathit{evs}, (\mathit{App}\ (\mathit{Switch}\ \mathit{sr}\ (\mathit{SF}\ \mathit{sf})\ f)\ s) : \mathit{eqs})]\!]_{\mathit{eqs}} = & \\
\quad \textbf{if}\ \mathit{elem}\ (\mathit{sf}\ s)\ \mathit{evs} & \\
\qquad \textbf{then}\quad \mathit{App}\ (f\ [\![(\mathit{time}\ \mathit{st}, \mathit{instants}\ \mathit{st}, s)]\!]_{\mathit{sig}})\ s : [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} & \\
\qquad \textbf{else}\quad \mathit{App}\ (\mathit{Switch}\ (\mathit{SR}\ (\lambda\_ \to [\![(\mathit{st}, \mathit{evs}, [\mathit{App}\ \mathit{sr}\ s])]\!]_{\mathit{eqs}}))\ (\mathit{SF}\ \mathit{sf})\ f)\ s & \\
\qquad\quad : [\![(\mathit{st}, \mathit{evs}, \mathit{eqs})]\!]_{\mathit{eqs}} &
\end{array}
$$

Figure 6.10: Function that handles events.

$$
\begin{array}{l}
\mathit{flattenEvents} :: \mathit{SymTab} \to \mathit{SymTab} \\
\mathit{flattenEvents}\ \mathit{st} = \mathit{st}\ \{\mathit{events} = [\![(0, \mathit{st}\ \{\mathit{events} = [\,]\}, \mathit{model}\ \mathit{st})]\!]_{\mathit{eqs}}\}
\end{array}
$$

$$
\begin{array}{ll}
[\![\cdot]\!]_{\mathit{eqs}} :: (\mathit{Integer}, \mathit{SymTab}, [\mathit{Equation}]) \to \mathit{SymTab} & \\
[\![(\_, \mathit{st}, [\,])]\!]_{\mathit{eqs}} & = \mathit{st} \\
[\![(i, \mathit{st}, (\mathit{Local}\ f) : \mathit{eqs})]\!]_{\mathit{eqs}} & = [\![(i+1, \mathit{st}, f\ (\mathit{Var}\ i) +\!\!+ \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(i, \mathit{st}, (\mathit{Equal}\ \_\ \_) : \mathit{eqs})]\!]_{\mathit{eqs}} & = [\![(i, \mathit{st}, \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(i, \mathit{st}, (\mathit{Init}\quad \_\ \_) : \mathit{eqs})]\!]_{\mathit{eqs}} & = [\![(i, \mathit{st}, \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(i, \mathit{st}, (\mathit{App}\ (\mathit{SR}\ \mathit{sr})\ s) : \mathit{eqs})]\!]_{\mathit{eqs}} & = [\![(i, \mathit{st}, \mathit{sr}\ s +\!\!+ \mathit{eqs})]\!]_{\mathit{eqs}} \\
[\![(i, \mathit{st}, (\mathit{App}\ (\mathit{Switch}\ \mathit{sr}\ (\mathit{SF}\ \mathit{sf})\ \_)\ s) : \mathit{eqs})]\!]_{\mathit{eqs}} = & \\
\quad [\![(i, \mathit{st}\ \{\mathit{events} = (\mathit{sf}\ s) : (\mathit{events}\ \mathit{st})\}, (\mathit{App}\ \mathit{sr}\ s) : \mathit{eqs})]\!]_{\mathit{eqs}} &
\end{array}
$$

Figure 6.11: Function that generates the flat list of events that may occur in the active mode of operation.

$$
\begin{aligned}
&[\![\cdot]\!]_{sig} :: (\mathbb{R}, \mathit{Array\ Integer}\ (\mathbb{R},\mathbb{R}), \mathit{Signal}\ a) \to a \\
&[\![(\_, \_, \mathit{Unit})]\!]_{sig} &&= () \\
&[\![(t, \_, \mathit{Time})]\!]_{sig} &&= t \\
&[\![(\_, \_, \mathit{Const}\ c)]\!]_{sig} &&= c \\
&[\![(\_, v, \mathit{Var}\ i)]\!]_{sig} &&= \mathit{fst}\ (v\ !\ i) \\
&[\![(t, v, \mathit{Pair}\ e_1\ e_2)]\!]_{sig} &&= ([\![(t, v, e_1)]\!]_{sig}, [\![(t, v, e_2)]\!]_{sig}) \\
&[\![(\_, v, \mathit{PrimApp\ Der}\ (\mathit{Var}\ i))]\!]_{sig} &&= \mathit{snd}\ (v\ !\ i) \\
&[\![(t, v, \mathit{PrimApp\ sf}\ e)]\!]_{sig} &&= [\![sf]\!]_{sf}\ [\![(t, v, e)]\!]_{sig}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\cdot]\!]_{sf} :: \mathit{PrimSF}\ a\ b \to (a \to b) \\
&[\![\mathit{Exp}]\!]_{sf} &&= \mathit{exp} \\
&[\![\mathit{Sqrt}]\!]_{sf} &&= \mathit{sqrt} \\
&[\![\mathit{Log}]\!]_{sf} &&= \mathit{log} \\
&[\![\mathit{Sin}]\!]_{sf} &&= \mathit{sin} \\
&[\![\mathit{Tan}]\!]_{sf} &&= \mathit{tan} \\
&[\![\mathit{Cos}]\!]_{sf} &&= \mathit{cos} \\
&[\![\mathit{Asin}]\!]_{sf} &&= \mathit{asin} \\
&[\![\mathit{Atan}]\!]_{sf} &&= \mathit{atan} \\
&[\![\mathit{Acos}]\!]_{sf} &&= \mathit{acos} \\
&[\![\mathit{Sinh}]\!]_{sf} &&= \mathit{sinh} \\
&[\![\mathit{Tanh}]\!]_{sf} &&= \mathit{tanh} \\
&[\![\mathit{Cosh}]\!]_{sf} &&= \mathit{cosh} \\
&[\![\mathit{Asinh}]\!]_{sf} &&= \mathit{asinh} \\
&[\![\mathit{Atanh}]\!]_{sf} &&= \mathit{atanh} \\
&[\![\mathit{Acosh}]\!]_{sf} &&= \mathit{acosh} \\
&[\![\mathit{Add}]\!]_{sf} &&= \mathit{uncurry}\ (+) \\
&[\![\mathit{Mul}]\!]_{sf} &&= \mathit{uncurry}\ (*) \\
&[\![\mathit{Div}]\!]_{sf} &&= \mathit{uncurry}\ (/) \\
&[\![\mathit{Pow}]\!]_{sf} &&= \mathit{uncurry}\ (**) \\
&[\![\mathit{Lt}]\!]_{sf} &&= \lambda d \to (d < 0) \\
&[\![\mathit{Lte}]\!]_{sf} &&= \lambda d \to (d \leqslant 0) \\
&[\![\mathit{Gt}]\!]_{sf} &&= \lambda d \to (d > 0) \\
&[\![\mathit{Gte}]\!]_{sf} &&= \lambda d \to (d \geqslant 0) \\
&[\![\mathit{Or}]\!]_{sf} &&= \mathit{uncurry}\ (\vee) \\
&[\![\mathit{And}]\!]_{sf} &&= \mathit{uncurry}\ (\wedge) \\
&[\![\mathit{Not}]\!]_{sf} &&= \neg
\end{aligned}
$$

Figure 6.12: Functions that evaluate instantaneous signal values.

$$
\begin{aligned}
&\mathit{flattenEquations} :: \mathit{SymTab} \to \mathit{SymTab} \\
&\mathit{flattenEquations}\ st = st\ \{\, \mathit{equations} = [\![(0, \mathit{model}\ st)]\!]_{eqs}\,\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\cdot]\!]_{eqs} && :: (\mathit{Integer}, [\mathit{Equation}]) \to [\mathit{Equation}] \\
&[\![(\_, [\,])]\!]_{eqs} && = [\,] \\
&[\![(i, (\mathit{App}\ (\mathit{SR}\ sr)\ s) : eqs)]\!]_{eqs} && = [\![(i, sr\ s +\!\!+ eqs)]\!]_{eqs} \\
&[\![(i, (\mathit{App}\ (\mathit{Switch}\ sr\ \_\ \_)\ s) : eqs)]\!]_{eqs} && = [\![(i, (\mathit{App}\ sr\ s) : eqs)]\!]_{eqs} \\
&[\![(i, (\mathit{Local}\ f) : eqs)]\!]_{eqs} && = [\![(i+1, f\ (\mathit{Var}\ i) +\!\!+ eqs)]\!]_{eqs} \\
&[\![(i, (\mathit{Equal}\ \_\ \_) : eqs)]\!]_{eqs} && = eq : [\![(i, eqs)]\!]_{eqs} \\
&[\![(i, (\mathit{Init}\ \_\ \_) : eqs)]\!]_{eqs} && = eq : [\![(i, eqs)]\!]_{eqs}
\end{aligned}
$$

Figure 6.13: Functions that flatten hierarchical systems of equations.

is the shortest formal and executable definition of the flattening process for a noncausal modelling language. This is partly due to the simple abstract syntax and utilisation of shallow embedding techniques, specifically, embedded functions in the *SR* and *Switch* constructors.

The default symbolic processor that is described in this section can be extended by modellers. This extensibility is especially useful for providing further symbolic processing steps that operate on flat systems of equations. For example, the default symbolic processor does not implement an index reduction transformation [Cellier and Kofman, 2006]. Index reduction transformations minimise algebraic dependencies between equations involved in flat systems of DAEs. This allows numerical solvers to more efficiently simulate DAEs [Brenan et al., 1996]. An overview of index reduction algorithms is given in the book by Cellier and Kofman [2006]. One of those algorithms can be used to extend the default symbolic processor by introducing an index reduction step after the *flattenEquations* step.

As an example of a symbolic processor extension the implementation of Hydra provides a processor that handles higher-order derivatives and derivatives of complex signal expressions (i.e., not just signal variables). Equations that involve higher-order derivatives are translated into equivalent set of equations involving only first-order derivatives. Derivatives of complex signal expressions are simplified using symbolic differentiation.

6.4 Just-in-time Compilation

Mathematically the end result of the stage of symbolic processing is the following list of equations:

$$i(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_i} \quad (6.1)$$

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_f} \quad (6.2)$$

$$e(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = \vec{r_e} \quad (6.3)$$

Here, $\vec{x}$ is a vector of differential variables, $\vec{y}$ is a vector of algebraic variables, t is time, $\vec{r_i}$ is a residual vector of initialisation equations, $\vec{r_f}$ is a residual vector of differential algebraic equations, and $\vec{r_e}$ is a vector of zero-crossing signal values. The aforementioned vectors are signals; that is, time-varying vectors.

Equation 6.1 corresponds to the *Init* equations that are placed in the *equations* field of the symbol table and determines the initial conditions for Equation 6.2; that is, the values of $\frac{d\vec{x}}{dt}$,$\vec{x}$ and $\vec{y}$ at the starting time of the active mode of operation. Equation 6.2 corresponds to the *Equal* equations that are placed in the *equations* field of the symbol table, and thus is the main DAE of the system that is integrated over time starting from the initial conditions. Equation 6.3 corresponds to the zero-crossing signals placed in the *events* field of the symbol table and specifies event conditions.

The task of a DAE solver is to find time varying valuations of $\vec{x}$ and $\vec{y}$ such that the residual vectors are zero. In addition, a DAE solver is required to detect points in time when the vector $\vec{r_e}$ changes and report it as an event occurrence.

The generated equations are implicitly formulated ones. In general, it is not possible to transform these implicit equations into explicit ones; that is, to completely causalise them [Brenan et al., 1996]. Consequently, a system of implicit equations needs to be solved at the start of the simulation of each mode of operation and at every integration step. For example, a numerical solution of the implicitly formulated DAE given in Equation 6.2 involves evaluation of the function f a number of times (sometimes hundreds or more at each integration step), with varying arguments, until it converges to zero. The number of executions of f depends on various factors including the required precision, the initial guess, the degree of nonlinearity of the DAE and so on.

As the functions i, f and e are evaluated from within inner loops of the solver, they have to be compiled into machine code for efficiency. Any interpretive overhead here would be considered intolerable by practitioners for most applications. However, as Hydra allows the equations to be changed in arbitrary ways *during* simulation, the

equations have to be compiled whenever they change, as opposed to only prior to simulation. As an optimisation, the code compiled for equations might be cached for future, possible reuse (see Chapter 8). The implementation of Hydra employs JIT machine code generation using the compiler infrastructure provided by LLVM. The functions i, f and e are compiled into LLVM instructions that in turn are compiled by the LLVM JIT compiler into native machine code. Function pointers to the generated machine code are then passed to the numerical solver.

The function pointers have the following Haskell type:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Void}\\
&\textbf{type}\ \mathit{Residual} = \mathit{FunPtr}\ (\quad \mathbb{R}\\
&\qquad\qquad\qquad\qquad \rightarrow \mathit{Ptr}\ \mathbb{R}\\
&\qquad\qquad\qquad\qquad \rightarrow \mathit{Ptr}\ \mathbb{R}\\
&\qquad\qquad\qquad\qquad \rightarrow \mathit{Ptr}\ \mathbb{R}\\
&\qquad\qquad\qquad\qquad \rightarrow \mathit{IO}\ \ \mathit{Void})
\end{aligned}
$$

The first function argument is time. The second argument is a vector of real valued signal. The third argument is a vector of differentials of real-valued signals. The forth argument is a vector of residuals, or in the case of the event specification vector of zero-crossing signal values. The residual functions read the first three arguments and write the residual values in the fourth argument. As these functions are passed to numerical solvers it is critical to allow for fast positional access of vector elements and in-place vector updates. Hence the use of C-style arrays.

Figure 6.14 gives the unoptimised LLVM code that is generated for the parametrised van der Pol oscillator. The corresponding optimised LLVM is given in Figure 6.15.

6.5 Numerical Simulation

The default numerical solver used in the current implementation of Hydra is SUNDIALS [Hindmarsh et al., 2005]. The solve components we use are KINSOL, a nonlinear algebraic equation systems solver, and IDA, a differential algebraic equation systems solver. The code for the function i is passed to KINSOL that numerically solves the system and returns initial values (at time t_0) of $\frac{d\vec{x}}{dt}$,$\vec{x}$ and $\vec{y}$. These vectors together with the code for the functions f and e are passed to IDA that proceeds to solve the DAE by numerical integration. This continues until either the simulation is complete

```
define void @hydra_residual_main(double, double*, double*, double*) {
entry:
  %4  = getelementptr double* %2, i32 1
  %5  = load double* %4
  %6  = getelementptr double* %1, i32 0
  %7  = load double* %6
  %8  = fmul double -1.000000e+00, %7
  %9  = getelementptr double* %1, i32 0
  %10 = load double* %9
  %11 = getelementptr double* %1, i32 0
  %12 = load double* %11
  %13 = fmul double %10, %12
  %14 = fmul double -1.000000e+00, %13
  %15 = fadd double 1.000000e+00, %14
  %16 = fmul double 3.000000e+00, %15
  %17 = getelementptr double* %1, i32 1
  %18 = load double* %17
  %19 = fmul double %16, %18
  %20 = fadd double %8, %19
  %21 = fmul double -1.000000e+00, %20
  %22 = fadd double %5, %21
  %23 = getelementptr double* %3, i32 0
  store double %22, double* %23
  br label %BB_0

BB_0:
  %24 = getelementptr double* %1, i32 1
  %25 = load double* %24
  %26 = getelementptr double* %2, i32 0
  %27 = load double* %26
  %28 = fmul double -1.000000e+00, %27
  %29 = fadd double %25, %28
  %30 = getelementptr double* %3, i32 1
  store double %29, double* %30
  br label %BB_1

BB_1:
  ret void
}
```

Figure 6.14: Unoptimised LLVM code for the parametrised van der Pol oscillator.

```
define void @hydra_residual_main(double, double*, double*, double*) {
entry:
  %4  = getelementptr double* %2, i32 1
  %5  = load double* %4
  %6  = load double* %1
  %7  = fmul double %6, -1.000000e+00
  %8  = fmul double %6, %6
  %9  = fmul double %8, -1.000000e+00
  %10 = fadd double %9, 1.000000e+00
  %11 = fmul double %10, 3.000000e+00
  %12 = getelementptr double* %1, i32 1
  %13 = load double* %12
  %14 = fmul double %11, %13
  %15 = fadd double %7, %14
  %16 = fmul double %15, -1.000000e+00
  %17 = fadd double %5, %16
  store double %17, double* %3
  %18 = load double* %12
  %19 = load double* %2
  %20 = fmul double %19, -1.000000e+00
  %21 = fadd double %18, %20
  %22 = getelementptr double* %3, i32 1
  store double %21, double* %22
  ret void
}
```

Figure 6.15: Optimised LLVM code for the parametrised van der Pol oscillator.

or until one of the events defined by the function e occurs. Event detection facilities are provided by IDA.

Modellers are allowed to replace the default numerical solver. In fact, any solver that implements the interface that is given in Figure 6.16 can be used. The default numerical solver implements this interface by using foreign function interface to the SUNDIALS library that is written in C.

After each integration step that calculates a numerical approximation of a vector of active signal variables the simulator calls the *defaultTrajectoryVisualiser* function that writes the simulation results into standard output. Hydra users can provide their own signal trajectory visualiser of the following type:

$$
\begin{aligned}
\textbf{type}\ \mathit{TrajectoryVisualiser} = {} & \mathbb{R} && \text{-- Time} \\
& \rightarrow \mathit{Int} && \text{-- Variable number} \\
& \rightarrow \mathit{Ptr}\ \mathbb{R} && \text{-- Variables} \\
& \rightarrow \mathit{IO}\ () &&
\end{aligned}
$$

For example, the user can animate the trajectories using a suitable graphical programming library.

```
type SolverHandle = Ptr Void
data NumericalSolver = NumericalSolver {
    createSolver   :: ℝ              -- Starting time
                   → ℝ              -- Stopping time
                   → Ptr ℝ          -- Current time
                   → ℝ              -- Absolute tolerance
                   → ℝ              -- Relative tolerance
                   → Int            -- Number of variables
                   → Ptr ℝ          -- Variables
                   → Ptr ℝ          -- Differentials
                   → Ptr Int        -- Constrained differentials
                   → Int            -- Number of events
                   → Ptr Int        -- Events
                   → Residual       -- Initialisation equations
                   → Residual       -- Main equations
                   → Residual       -- Event Equations
                   → IO SolverHandle
  , destroySolver  :: SolverHandle → IO ()
  , solve          :: SolverHandle → IO CInt
      -- Return value 0: Soulution has been obtained succesfully
      -- Return value 1: Event occurence
      -- Return value 2: Stopping time has been reached
  }
```

Figure 6.16: Numerical solver interface.

6.6 Performance

In this section we provide a performance evaluation of the implementation of Hydra. The aim of the evaluation is to communicate to noncausal modelling language designers and implementers performance overheads of Hydra's language constructs and implementation techniques that are absent from main-stream noncausal languages. Specifically, we are mainly concerned with the overheads of mode switching (computing new structural configurations at events, runtime symbolic processing of the equations, and JIT compilation) and how this scales when the size of the models grow in order to establish the feasibility of our approach. The time spent on numerical simulation is of less interest as we are using standard numerical solvers, and as our model equations are compiled down to native code with efficiency on par with statically generated code, this aspect of the overall performance should be roughly similar to what can be obtained from other compilation-based modelling and simulation language implementations. For this reason, and because other compilation-based, noncausal modelling and simulation language implementations do not carry out dynamic mode switching, we do not compare the performance to other simulation software. The results would not be very meaningful.

	200 Components 1000 Equations $t \epsilon [0, 10)$		400 Components 2000 Equations $t \epsilon [10, 20)$		600 Components 3000 Equations $t \epsilon [20, 30)$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.067	0.6	0.153	0.6	0.244	0.5
JIT Compilation	1.057	10.2	2.120	8.3	3.213	6.6
Numerical Simulation	9.273	89.2	23.228	91.1	45.140	92.9
Total	10.397	100.0	25.501	100.0	48.598	100.0

Table 6.1: Time profile of structurally dynamic RLC circuit simulation (part I).

The implementation of Hydra provides for user-defined symbolic processors and numerical solvers. It does not provide for a user-defined JIT compiler. In the following we evaluate the performance of the default symbolic processor, the default numerical solver and the built-in LLVM-based JIT compiler.

The evaluation setup is as follows. The numerical simulator integrates the system using variable-step, variable-order BDF (Backward Differentiation Formula) solver [Brenan et al., 1996]. Absolute and relative tolerances for numerical solution are set to 10^{-6} and trajectories are printed out at every point where $t = 10^{-3} * k, k \epsilon \mathbb{N}$. For static compilation Haskell-embedded models and JIT compilation we use GHC 6.10.4 and LLVM 2.5, respectively. Simulations are performed on a 2.0 GHz x86-64 Intel® Core™2 CPU. However, presently, we do not exploit any parallelism, running everything on a single core.

To evaluate how the performance of the implementation scales with an increasing number of equations, we constructed a structurally dynamic model of an RLC circuit (i.e., a circuit consisting of resistors, inductors and capacitors) with dynamic structure. In the initial mode of operation the circuit contains 200 components, described by 1000 equations in total (5 equations for each component). Every time $t = 10 * k$, where $k \epsilon \mathbb{N}$, the number of circuit components is increased by 200 (and thus the number of equations by 1000) by switching the additional components into the circuit.

Tables 6.1 and 6.2 show the amount of time spent in each mode of the system and in each conceptual stage of simulation of the structurally dynamic RLC circuit. In absolute terms, it is evident that the extra time spent on the mode switches becomes significant as the system grows. However, in relative terms, the overheads of our dynamic code generation approach remains low at about 10 % or less of the overall simulation time.

While JIT compilation remains the dominating part of the time spent at mode switches, Figure 6.17 demonstrates that the performance of the JIT compiler scales well.

	800 Components 4000 Equations $t \in [30, 40)$		1000 Components 5000 Equations $t \in [40, 50)$		1200 Components 6000 Equations $t \in [50, 60]$	
	CPU Time		CPU Time		CPU Time	
	s	%	s	%	s	%
Symbolic Processing	0.339	0.4	0.454	0.4	0.534	0.3
JIT Compilation	4.506	4.9	5.660	5.1	6.840	4.3
Numerical Simulation	86.471	94.7	105.066	94.5	152.250	95.4
Total	91.317	100.0	111.179	100.0	159.624	100.0

Table 6.2: Time profile of structurally dynamic RLC circuit simulation (part II).

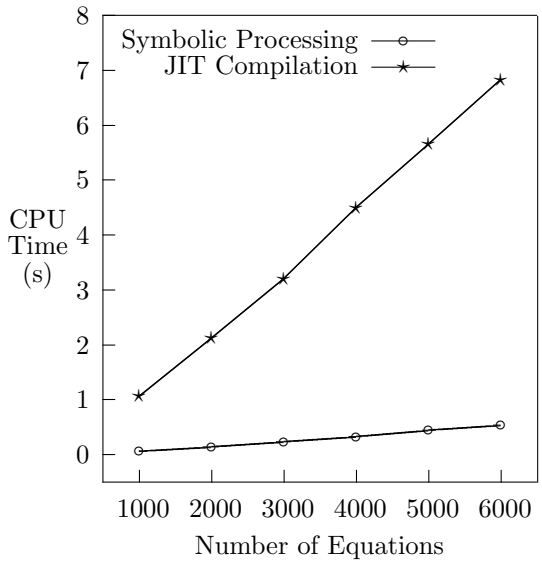

Figure 6.17: Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation.

In particular, compilation time increases roughly linearly in the number of equations. The time spent on symbolic processing and event handling remains encouragingly modest (both in relative and absolute terms) and grows slowly as model complexity increases.

In the current implementation of Hydra, a new flat system of equations is generated at each mode switch without reusing the equations of the previous mode. It may be useful to identify exactly what has changed at each mode switch, thus enabling the reuse of *unchanged* equations and associated code from the previous mode. In particular, information about the equations that remain unchanged during the mode switches provides opportunities for the JIT compiler to reuse the machine code from the previous mode, thus reducing the burden on the JIT compiler and consequently the compilation time during mode switches. We think it is worthwhile to investigate reusable code generation aspects in the context of noncausal modelling and simulation of structurally dynamic

systems, and the suitability of the proposed execution model for (soft) real-time simulation. Currently, for large structurally dynamic systems, the implementation is only suitable for offline simulation.

The implementation of Hydra offers new functionality in that it allows noncausal modelling and simulation of structurally dynamic systems that simply cannot be handled by static approaches. Thus, when evaluating the feasibility of our approach, one should weigh the inherent overheads against the limitation and inconvenience of not being able to model and simulate such systems noncausally.

Chapter 7

Related Work

7.1 Embedded Domain Specific Languages

The deep-embedding techniques used in the Hydra implementation for domain-specific optimisations and efficient code generation draws from the extensive work on compiling staged domain-specific embedded languages. Examples include Elliott et al. [2000] and Mainland et al. [2008]. However, these works are concerned with compiling programs all at once, meaning the host language is used only for meta-programming, not for running the actual programs.

The use of quasiquoting in the implementation of Hydra draws its inspiration from Flask, a domain-specific embedded language for programming sensor networks [Mainland et al., 2008]. However, we had to use a different approach to type checking. A Flask program is type checked by a domain-specific type checker after being generated, just before the subsequent compilation into the code to be deployed on the sensor network nodes. This happens at host-language run-time. Because Hydra is iteratively staged, we cannot use this approach: we need to move type checking back to host-language compile-time. The Hydra implementation thus translates embedded programs into typed combinators at the stage of quasiquoting, charging the host-language type checker with checking the embedded terms. This ensures only well-typed programs are generated at run-time.

The FHM design was originally inspired by Functional Reactive Programming (FRP) [Elliott and Hudak, 1997], particularly Yampa [Nilsson et al., 2002]. A key difference is that FRP provides functions on signals whereas FHM generalises this to relations on

signals. FRP can thus be seen as a framework for causal simulation, while FHM supports noncausal simulation. Signal functions are first class entities in most incarnations of FRP, and new ones can be computed and integrated into a running system dynamically. As we have seen, this capability has also been carried over to FHM. This means that these FRP versions, including Yampa, also are examples of iteratively staged languages. However, as all FRP versions supporting highly dynamic system structure so far have been interpreted, the program generation aspect is much less pronounced than what is the case for Hydra. That said, in Yampa, program fragments are generated and then optimised dynamically [Nilsson, 2005]. It would be interesting to try to apply the implementation approaches described in this thesis (i.e., runtime symbolic processing and JIT compilation) to a version of FRP, especially in the context of the recently proposed optimisations by Liu et al. [2009] and Sculthorpe [2011].

7.2 Noncausal Modelling and Simulation Languages

7.2.1 Sol

Sol is a Modelica-like language [Zimmer, 2007, 2008]. It introduces language constructs that enable the description of systems where objects are dynamically created and deleted, thus aiming at supporting modelling of highly structurally dynamic systems. So far, the research emphasis has been on the design of the language itself along with support for incremental dynamic recausalisation and dynamic handling of structural singularities. An interpreter is used for simulation. The work on Sol is thus complementary to ours: techniques for dynamic compilation would be of interest in the context of Sol to enable it to target high-end simulation tasks; conversely, algorithms for incremental recausalisation is of interest to us to minimise the amount of work needed to regenerate simulation code after structural changes.

7.2.2 MOSILAB

MOSILAB is an extension of the Modelica language that supports the description of structural changes using object-oriented statecharts [Nytsch-Geusen et al., 2005]. This enables modelling of structurally dynamic systems. The language extension has a compiled implementation. However, the statechart approach implies that all structural modes must be explicitly specified in advance, meaning that MOSILAB does not sup-

port highly structurally dynamic systems. Even so, if the number of possible configurations is large (perhaps generated mechanically by meta-modelling), higher-order and structurally dynamic modelling techniques and their implementations investigated here might be of interest also in the implementation of MOSILAB.

7.2.3 Modelling Kernel Language

Broman [Broman, 2007, Broman and Fritzson, 2008] developed Modelling Kernel Language (MKL). The language is intended to be a core language for noncausal modelling languages such as Modelica. Broman takes a functional approach to noncausal modelling, similar to the FHM approach proposed by Nilsson et al. [2003]. One of the main goals of MKL is to provide a formal semantics of the core language. The semantics is based on an untyped, effectful λ-calculus.

Similarly to Hydra, MKL provides a λ-abstraction for defining functions and an abstraction similar to *rel* for defining noncausal models. Both functions and noncausal models are first-class entities in MKL, enabling higher-order, noncausal modelling. The similarity of the basic abstractions in Hydra and MKL leads to a similar style of modelling in both languages.

Thus far, the work on MKL has not specifically considered support for structural dynamics, meaning that its expressive power in that respect is similar to current mainstream, noncausal modelling and simulation languages like Modelica. However, given the similarities between MKL and Hydra, MKL should be a good setting for exploring support for structural dynamics, which ultimately could carry over to better support for structural dynamics for any higher-level language that has a semantics defined by translation into MKL. Again, the implementation techniques discussed in this paper should be of interest in such a setting.

7.2.4 Acumen

[Zhu et al., 2010].

Chapter 8

Directions for Future Work and Conclusions

Bibliography

Kathryn Eleda Brenan, Stephen La Vern Campbell, and Linda Ruth Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. SIAM, Philadelphia, 1996.

David Broman. Flow Lambda Calculus for declarative physical connection semantics. Technical Reports in Computer and Information Science 1, Linköping University Electronic Press, 2007.

David Broman and Peter Fritzson. Higher-order acausal models. In Peter Fritzson, Fran cois Cellier, and David Broman, editors, *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, number 29 in Linköping Electronic Conference Proceedings, pages 59–69, Paphos, Cyprus, 2008. Linköping University Electronic Press.

John Joseph Capper and Henrik Nilsson. Static balance checking for first-class modular systems of equations. In *Proceedings of the 11th Symposium on Trends in Functional Programming*, Oklahoma City, Oklahoma, U.S.A, May 2010.

François E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

François E. Cellier. Object-oriented modelling: Means for dealing with system complexity. In *Proceedings of the 15th Benelux Meeting on Systems and Control*, pages 53–64, 1996.

François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag, 2006.

Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York,

NY, USA, 2003. ACM. ISBN 1-58113-758-3. doi: http://doi.acm.org/10.1145/871895.871897. URL http://doi.acm.org/10.1145/871895.871897.

Dymmola. Dynamic Modeling Laboratory, 2008. URL http://www.dynasim.se/.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of International Conference on Functional Programming*, pages 163–173, June 1997.

Conal Elliott, Sigbjørn Finne, and Oege de Moor. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *Lecture Notes in Computer Science*, pages 9–27, Montreal, Canada, September 2000. Springer-Verlag.

P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. *2006 IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1588–1595, October 2006. doi: 10.1109/CACSD.2006.285495.

George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, 2009. Linköping University Electronic Press. ISBN 978-91-7393-513-5. doi: http://dx.doi.org/10.3384/ecp09430137.

George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In *Proceedings of the 19th international workshop on Functional and (Constraint) Logic Programming*, Madrid, Spain, 2010. To appear in peer-reviewed proceedings published by Springer LNCS.

Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396, 2005. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1089014.1089020.

Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0521643384.

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming, 4th International School 2002*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2007. ISBN 0521871727.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming (ICFP '06)*, pages 50–61. ACM, 2006.

Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, December 2002.

Edward A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008. URL `http://chess.eecs.berkeley.edu/pubs/427.html`. Invited Paper.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596559. URL `http://doi.acm.org/10.1145/1596550.1596559`.

Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5. doi: http://doi.acm.org/10.1145/1291201.1291211.

Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming*, Victoria, British Columbia, Canada, September 2008. ACM Press.

Simulink. *Using Simulink Version 7*. The MathWorks, Inc., March 2008. URL `http://www.mathworks.com/products/simulink/`.

Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

Modelica Tutorial. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling: Tutorial Version 1.4*. The Modelica Association, December 2000.

Modelica. *A Unified Object-Oriented Language for Physical Systems Modeling : Language Specification Version 3.2*. The Modelica Association, March 2010. URL `https://www.modelica.org/documents/ModelicaSpec32.pdf`.

Pieter J. Mosterman. *Hybrid Dynamic Systems: A Hybrid Bond Graph Modeling Paradigm and its Application in Diagnosis*. PhD thesis, Graduate School of Vanderbilt University, Nashville, Tennessee, May 1997.

Pieter J. Mosterman. An overview of hybrid simulation phenomena and their support by simulation packages. In *HSCC '99: Proceedings of the Second International Workshop on Hybrid Systems*, pages 165–177, London, UK, 1999. Springer-Verlag. ISBN 3-540-65734-7.

Pieter J. Mosterman, Gautam Biswas, and Martin Otter. Simulation of discontinuities in physical system models based on conservation principles. In *Proceedings of SCS Summer Conference 1998*, pages 320–325, July 1998.

Henrik Nilsson. Functional automatic differentiation with Dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, August 2003. ACM Press.

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)*, pages 54–65, Tallinn, Estonia, September 2005. ACM Press.

Henrik Nilsson. Type-based structural analysis for modular systems of equations. In Peter Fritzson, François Cellier, and David Broman, editors, *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, number 29 in Linköping Electronic Conference Proceedings, pages 71–81, Paphos, Cyprus, July 2008. Linköping University Electronic Press.

Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of 5th International Workshop on Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling from an object-oriented perspective. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 71–87, Berlin, Germany, 2007. Linköping University Electronic Press.

NSF. Cyber-physical systems (CPS), 2008. URL `http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.pdf`.

Christoph Nytsch-Geusen, Thilo Ernst, André Nordwig, Peter Schwarz, Peter Schneider, Matthias Vetter, Christof Wittwer, Thierry Nouidui, Andreas Holm, Jürgen Leopold, Gerhard Schmidt, Alexander Mattes, and Ulrich Doll. MOSILAB: Development of a modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, pages 527–535, Hamburg, Germany, 2005.

Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008. ISBN 0596514980, 9780596514983.

Michael Pellauer, Markus Forsberg, and Aarne Ranta. BNF Converter: Multilingual front-end generation from labelled BNF grammars. Technical report, Computing Science at Chalmers University of Technology and Gothenburg University, Sep 2004.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Gordon Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

Dana Scott. Domains for denotational semantics. In *Automata, Languages and Programming*, pages 577–613, 1982.

Michael Scott. *Programming Language Pragmatics, Third Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009. ISBN 0123745144, 9780123745149.

Neil Sculthorpe. *Towards Safe and Efficient Functional Reactive Programming.* PhD thesis, School of Computer Science, University of Nottingham, 2011.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/636517.636528.

Don Stewart. Domain specific languages for domain specific problems. In *Workshop on Non-Traditional Programming Models for High-Performance Computing.* LACSS, 2009.

Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer Berlin/Heidelberg, 2004.

Simon Thompson. *The Haskell: The Craft of Functional Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201342758.

Günther Zauner, Daniel Leitner, and Felix Breitenecker. Modelling structural-dynamics systems in Modelica/Dymola, Modelica/MOSILAB, and AnyLogic. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 99–110, Berlin, Germany, 2007. Linköping University Electronic Press.

Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 1–11, New York, NY, USA, 2010. ACM.

ISBN 978-1-4503-0066-7. doi: http://doi.acm.org/10.1145/1795194.1795196. URL http://doi.acm.org/10.1145/1795194.1795196.

Dirk Zimmer. Enhancing Modelica towards variable structure systems. In Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, number 24 in Linköping Electronic Conference Proceedings, pages 61–70, Berlin, Germany, 2007. Linköping University Electronic Press.

Dirk Zimmer. Introducing Sol: A general methodology for equation-based modeling of variable-structure systems. In *Proceedings of the 6th International Modelica Conference*, pages 47–56, Bielefeld, Germany, 2008.