SEMANTIC METHODS

FOR

FUNCTIONAL HYBRID MODELLING

JOHN CAPPER, BSc.

Thesis submitted to the University of Nottingham

for the degree of Doctor of Philosophy

November 2014

Abstract

Equation-based modelling languages have become a vital tool in many areas of science and engineering. Functional Hybrid Modelling (FHM) is an approach to equation-based modelling that allows the behaviour of a physical system to be expressed as a *modular hierarchy* of undirected equations. FHM supports a variety of advanced language features — such as higher-order models and variable system structure — that sets it apart from the majority of other modelling languages. However, the inception of these new features has not been accompanied by the semantic tools required to effectively use and understand them. Specifically, there is a lack of static safety assurances for dynamic models and the semantics of the aforementioned language features are poorly understood.

Static safety guarantees are highly desirable as they allow problems that may cause an equation system to become unsolvable to be detected early, during compilation. As a result, the use of static analysis techniques to enforce structural invariants (e.g. that there are the same number of equations as unknowns) is now in use in main-stream equation-based languages like Modelica. Unfortunately, the techniques employed by these languages are somewhat limited, both in their capacity to deal with advanced language features and also by the spectrum of invariants they are able to enforce.

Formalising the semantics of equation-based languages is also important. Semantics allow us to better understand what a program is doing during execution, and to *prove* that this behaviour meets with our expectation. They also allow different implementations of a language to agree with one another, and can be used to demonstrate the correctness of a compiler or interpreter. However, current attempts to formalise such semantics typically fall short of describing advanced features, are not compositional, and/or fail to show correctness.

This thesis provides two major contributions to equation-based languages. Firstly, we develop a refined type system for FHM capable of capturing a larger number of structural anomalies than is currently possible with existing methods. Secondly, we construct a compositional semantics for the discrete aspects of FHM, and prove a number of key correctness properties.

Acknowledgements

Writing this thesis has been a long and difficult task, and I am certain that it would not have been possible without a great deal of support and friendship from my friends, family, and colleagues. First and foremost I would like to thank my supervisor Henrik Nilsson whose guidance, assistance, and constant enthusiasm has been invaluable. This thesis would not have been possible without him.

It has been a pleasure to work in the Functional Programming Laboratory and I wish to thank everyone there for making the last five years so interesting and enjoyable. In particular, I would like to acknowledge Graham Hutton and Thorsten Altenkirch who have been teaching me since my first year as an undergraduate. Florent Balestrieri, Iain Lane, Darin Morrison, Neil Sculthorpe, George Giorgidze, Laurence Day, and Bas van Gijzel also deserve many thanks for providing feedback and help with this thesis or earlier papers. I also thank my examiners Venanzio Capretta and Walid Taha for their time and constructive comments.

Finally, special thanks go to my parents who made this thesis possible with their continual support.

List of Figures

Contents

Chapter 1

Introduction

Systems of equations, also known as simultaneous equations, are a fundamental concept in many areas of mathematics, science, and engineering. Equations are used in many ways, such as to describe the behaviour of a physical process or to state the laws of a logical theory. Modelling and simulation, optimisation problems, and artificial intelligence are but a few possible applications. There are numerous *types* of equation systems. For example, *linear* systems of equations (i.e. equations that permit only linear operations such as addition and multiplication by a constant) play a vital role in many engineering and computer science problems. Another example is *differential algebraic* systems, which permit a much larger variety of operations (e.g. computing derivatives) and are key to the field of physical modelling and simulation.

The introduction of digital computers with ever improving performance has made it feasible to handle increasingly large and complex systems of equations. This in turn has spurred the creation of dedicated programming languages designed specifically to aid in the construction of equation systems that would otherwise be unmanageable. These languages typically support a modular approach, allowing systems to be described in a hierarchical fashion via the composition of individual equation system fragments. The systems can also be parameterised, describing not just a specific problem instance but a set of problems. In this thesis we investigate the semantics of equation-based languages.

One such class of languages that makes extensive use of equations are the physical modelling languages, which have received a significant amount of attention in the last few decades as they have been shown to be very useful for modelling a broad spectrum of physical phenomena [Cellier, 1996, Elmqvist, 1978, Mod, 2012, Nilsson et al., 2003, Simulink, 1992]. Examples include electronics, chemical reaction rates, population distribution, and the spread of infectious diseases [Cellier, 1991]. Moreover, there is ongoing research into a variety of more advanced features, such as support for first-class models, variable model structure, and iteratively-staged execution [Giorgidze, 2011, Zimmer, 2007].

For these reasons our work focuses on modelling and simulation languages as they provide a concrete setting that is representative of the wider field. While we wish to keep the scope of the thesis as broad as possible, a concrete setting has a number of advantages, and in light of our previous remarks, modelling languages provide a suitable such setting:

- The syntax and language features of existing modelling and simulation languages provide a solid foundation to begin semantic investigation.

- Our work can be phrased in terms of existing language design and implementation problems, which makes it easier to demonstrate that our work is of immediate practical value.

- The contributions of the thesis will be more accessible to those already familiar with the field of modelling and simulation languages.

- We are able to make contributions directly to the field of modelling and simulation languages. For example, the implementation of our refined type system (see Chap. 4) is immediately of use to modelling languages such as Modelica [Mod, 2012] and MKL [Broman and Fritzson, 2008].

1.1 Static Semantics for Equation-based Languages

In modern, high-level programming languages, types play a crucial role. Types help to create safe programs that conform with their specification. The strength of type systems can vary greatly, from the weak systems found in languages such as C [Kernigham and Ritchie, 1978], to the more powerful systems found in language such as Agda [Bove et al., 2009]. Stronger type systems allow more precise invariants of programs to be expressed, enabling a larger set of invalid programs to be statically rejected as ill-typed.

Equation-based languages are also often typed, with the types playing much the same roles as in conventional programming languages. Additionally, simple invariants relating to the structure of an equation system may be enforced, such as there being an equal number of variables and equations. The hope is that it becomes possible to statically detect structural problems that are likely to render a system ill-formed and thus unsolvable.

However, the development of advanced modelling language features has made the detection of structural anomalies a more difficult task. For example, first-class models mean that the fully-assembled structure of an equation system cannot be determined prior to elaboration (the process of "flattening" a hierarchical system of equations into a set). Detecting structural anomalies is even more challenging in a structurally-dynamic setting (i.e. where the equations describing the behaviour of a system may change during simulation) as structural invariants may only be violated at certain points in time *during* simulation.

Thus, there is considerable scope for improving the type systems of current equation-based languages. In one respect, the type systems can be generalised to a setting were equation system fragments have first-class status and where the systems of equations may be structurally dynamic. In the other respect, the type systems can be improved by refining the enforced structural invariants, thus allowing more potential problems to be detected early during compilation.

1.2 Dynamic Semantics for Equation-based Languages

Dynamic semantics express the computation of a language by explaining how the terms of the language are evaluated. Formalising a semantics is important as it allows us to better understand what a program is doing during execution, and to *prove* that this behaviour meets with our expectation. Semantics also allow different implementations of a language to agree with one another, and can be used to demonstrate the correctness of a compiler or interpreter.

Recent advances in equation-based language features have created an expressive framework for describing systems of equations. However, the semantics of these features are often poorly understood, which creates a rift between the user's perception of a program and the actual program behaviour. Structural dynamism, for example, introduces difficulties as it may be a long time before a structural configuration arises that exhibits unexpected behaviour.

To make matters worse, existing attempts often conflate the semantics of *computing* a set of equations with the semantics of *solving* the equation system [Giorgidze, 2011, Pepper et al., 2011]. This coupling is undesirable as it makes the semantics non-modular, and means it is often difficult to separate the two aspects. This may become a problem when we wish to change the type of equations (e.g. from linear to differential algebraic) as the semantics may be tied to a specific approach to describing a particular type of equation. Many of these approaches also produce *non-compositional* semantics [Broman, 2010, Henzinger, 1996, Kågedal, 1998, Kågedal and Fritzson, 1998], which means that language features are not always described independently, and that the introduction of new features alongside existing functionality can be a difficult task.

Thus, there is a need for a compositional semantics for equation-based languages that separates the act of computing equations and solving them. The semantics must be capable of describing advanced equation-based language features, such as first-class equation system fragments and structural dynamism.

There is then the opportunity to use the semantics to prove correctness properties of the language; for example, showing that a program cannot get "stuck" when attempting to compute a new system configuration.

1.3 Contributions of this Thesis

The motivation for the work in this thesis was born out of a desire to improve and better understand equation-based modelling languages such as Functional Hybrid Modelling (FHM). Advanced modelling techniques have been a very active area of research in recent years, yet the tools needed to understand and work with them have received relatively little attention. In this thesis, we advance the current state-of-the-art in two ways: by improving upon existing static methods for detecting structural anomalies in equation systems, and by mechanically formalising the discrete aspects of a fully-featured equation-based modelling language. As such, the contributions of this thesis fall into two categories.

The first part of this thesis investigates how the type system of FHM can be refined to include additional structural information. The development of the type system is broken down into two phases. Firstly, the global property of "equation-variable balance" is captured, allowing the type system to enforce that there are globally the same number of equations as variables, without requiring that this property be enforced locally for every subcomponent. Secondly, the balance criterion is further refined such that the type system is able to capture a much broader spectrum of structural anomalies. In summary, the contributions to static checking are as follows:

1. A novel type system for modular systems of equations supporting first-class components and structural dynamism.

2. A refined set of structural invariants based on classification of equations, allowing a larger class of structural anomalies to be prevented compared

with existing approaches.

3. A concise small-step semantics for the core of FHM, capturing the subtle behaviour of variables in a modular system of equations.

4. A proof of correctness for the simple balance type system.

The second part of the thesis is dedicated to finding a semantic model for the discrete parts of an FHM-like modelling language. The semantics are parameterised by the continuous aspects, allowing the continuous behaviour (i.e. finding a solution to a flat set of equations) to be described in whatever way is most appropriate for a given domain.

FHM supports many advanced modelling language features, and thus we argue that it is an ideal candidate for metatheoretical study. We develop a non-standard semantic model that captures aspects which are common to equation-based languages but that are rarely seen in other domains. In addition to describing how an initial flat set of equations is computed, we also describe how the system responds (at simulation runtime) to discrete events to produce new system configurations. In summary, the contributions to dynamic semantics of equation-based languages are as follows:

1. A compositional semantics for the discrete aspects of an acausal, hybrid, structurally dynamic modelling language expressed in dependent type theory using Normalisation by Evaluation. The discrete semantics encompass not only the translation of a hierarchical system of equations into a flat set, but also handle structural changes in response to events during simulation.

2. A novel formalisation of dynamism and the generation of new structural configurations that is declarative, avoiding the traditional imperative bias common to other approaches to dynamism.

3. A semantics that is carefully structured so as to allow the continuous aspects to be described separately, in whatever way is most appropriate

for the purpose at hand, while retaining the ability to describe precisely how a system evolves in response to discrete events.

4. Mechanised proofs of type preservation, termination and totality, and *completeness* for the semantics.

1.4 Overview of Peer-Reviewed Publications

The work in this thesis draws on three earlier publications [Capper and Nilsson, 2010, 2012, 2013] co-authored by myself and Henrik Nilsson. These publications have been superseded by this thesis. This thesis was written by myself and presents my own contributions. I have implemented the type systems and semantics described in this thesis, and the source code is available online from my personal webpage [1].

The first peer-reviewed publication [Capper and Nilsson, 2010] describes the preliminary investigation into the refinement type system. The work introduces the concept of adding structural information to types and the structural criteria used to generate constraints. However, the simple balance type system is not formalised and structural dynamism is not considered. The paper also makes no attempt to prove any metatheoretical properties, and the formalisation of the constraint-based system is more complicated than it appears in this thesis.

The second peer-reviewed publication [Capper and Nilsson, 2012] covers the initial work into constructing a semantic model for FHM. The paper uses the same approach as found in this thesis, but is less comprehensive. As in this thesis, the model is expressed in Agda and is thus known to be both total and terminating. No attempt is made to show any other meta-theoretical properties in this publication. The approach to structural dynamism is incomplete, and the semantics of local variables are not considered.

The third peer-reviewed publication [Capper and Nilsson, 2013] builds upon the earlier work on the refinement type systems by simplifying the presentation

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

and adding support for structural dynamism. It also provides a more thorough case study into the applications of the type system.

1.5 Prerequisites

This thesis assumes that readers are familiar with functional programming and also, though not essential, are familiar with Haskell in particular. Readers that are unfamiliar with functional programming or Haskell should refer to Peyton Jones [2003], Hutton [2007], or Thompson [1996].

The thesis also assumes a familiarity with elementary type theory and set theory. For an introduction to both see Pierce [2002].

Finally, Chap. 5 assumes that the reader is familiar with dependent type theory and mechanised theorem proving in Agda. An understanding of similar languages, such as Coq [Bertot and Castéran, 2004] is likely to be sufficient, but readers may also wish to refer to Norell [2009] for a tutorial on Agda.

1.6 Structure of this Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 introduces the fundamental concepts of equation-based modelling such as modularity, causality, and structural dynamism. The chapter then describes two different approaches to equation-based modelling, namely object-oriented modelling as illustrated by Modelica, and functional modelling as illustrated by FHM.

- Chapter 3 provides an introduction to Agda and other specialised notation used throughout the thesis.

- Chapter 4 develops the concept of a refined type system for detecting structural anomalies in equation-based languages. The chapter begins by describing a type system that tracks simple equation-variable balance.

The correctness of the balance type system is given with respect to an operational semantics. The refinements are then further developed to create a type system capable of detecting more kinds of structural errors. The chapter ends with a case study that demonstrates the applications of the refined type systems.

- Chapter 5 develops a semantic model of a core language based on FHM. Certain correctness properties are then established for the model and its interpreter. The chapter then presents a model of structural dynamism, and concludes by considering a number of extensions to the base model, such as delayed branch evaluation.
- Chapter 6 provides an overview of work closely related to our own.
- Finally, chapter 7 summarises the conclusions of our work and discusses possible future work.

Chapter 2

Equation-Based Modelling

This chapter introduces the fundamental concepts of equation-based modelling. Two different approaches are discussed: *object-oriented* modelling as illustrated by Modelica, and *functional* modelling as illustrated by Functional Hybrid Modelling (FHM). We pay particular attention to the latter approach as it forms the basis for much of the work in this thesis.

2.1 Preliminaries

2.1.1 Systems of Equations

A core principle of equational modelling is describing the behaviour of a physical system via a *system of equations*. A system of equations is a set of equations over a set of *variables* or *unknowns*. It has a solution if every variable in the system can be instantiated with a value such that all the equations are simultaneously satisfied. Moreover, if only one such instantiation exists, then the system has a *unique solution*. The following is an example of a system consisting of 2 equations and 2 unknowns:

$$x^2 + y = 0 \tag{2.1a}$$

$$3x = 10 \tag{2.1b}$$

The domain of the variables and signatures for equations is mostly orthogonal to the work presented in this thesis. The dimensions of an object are important to the kind of analyses that we investigate, and thus we will assume that all variables are of zero dimensions. Furthermore, for reasons of presentation, we will use the domains of reals or time-varying reals unless stated otherwise.

Returning to the equation system above, it can be solved by using (2.1b) to solve for x, substituting the value of x into (2.1a), thus enabling the latter to be used to solve for y.

Now consider the following parametrised version of the system instead. The solvability of the system now depends on the value of the *coefficient* $\boldsymbol{c}$. For example, when $\boldsymbol{c} = 0$, no solution exists.

$$x^2 + y = 0 \tag{2.2a}$$

$$\boldsymbol{c}x = 10 \tag{2.2b}$$

Whether or not a system of equations has a solution is an important property. For example, if a system of equations is intended to model a physical system, unsolvability would be indicative of a modelling fault. However, as the trivial example above illustrates, unless all aspects of the system are known, it may not be possible to answer this question, at least not directly. Moreover, depending on the domain, the question is in general undecidable (e.g. [Matiyasevich, 1993]).

Figure 2.1 shows a representation of a simple electrical circuit. The circuit is a typical example of a physical system. It consists of a number of two-pinned electrical components and a ground component. Much like any other domain,

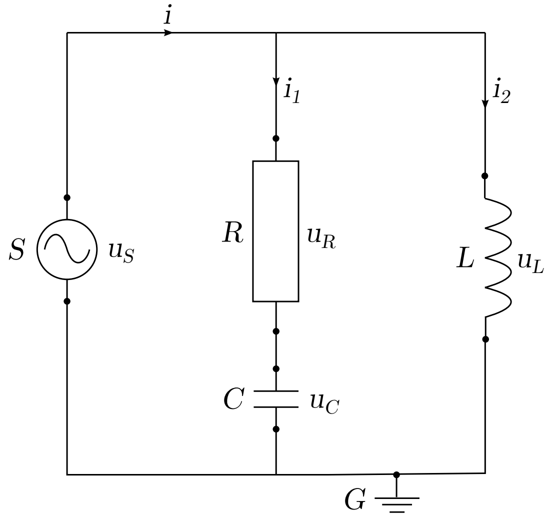

Figure 2.1: A simple electrical circuit.

the behaviour of an electrical system can be modelled via a system of equations. Four equations can be formulated to describe the components (2.3a – 2.3d) and three equations describe the circuit topology (2.3e – 2.3g). The interested reader can consult Cellier [1991] for further information on how an equational model can be derived from a physical system.

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

$$u_R = Ri_1 \tag{2.3b}$$

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

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

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

$$u_R + u_C = u_S \tag{2.3f}$$

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

In this example, the signature of the equations is given by implicit *differential* *algebraic equations* (DAEs) [Cellier and Kofman, 2006] and the domain of the variables is the time-varying reals. As a reminder, the signature and variable domain are not of particular importance to our work; for example, we could just as easily describe a linear system of equations over the integers instead.

2.1.2 Causality

The DAE system above is said to be *implicit* as the cause-and-effect relationship between variables is not made explicit. The equations are *undirected*: both known and unknown variables may appear on both sides of the equality, and thus no specific order in which to solve the equations is provided. By contrast, an explicit system can be seen as a series of assignments, whereby known (at that point) variables appear on one side of the equation with a single unknown appearing on the other (which then becomes known in subsequent equations). *Ordinary differential equations* (ODEs) are one example of an explicit system. A language that only provides support for explicit equation systems, such as Simulink [1992], is known as *causal*. Conversely, a language such as Hydra that allows for implicit equations is known as a *noncausal* (or *acausal*) language.

Consider Pell's equation [Barbeau, 2003] over the two unknowns x and y and parametrised by an integer n:

$$x^2 - ny^2 = \pm 1 \tag{2.4}$$

The equation is implicit: depending on which variable is known in a specific context, the equation can be translated into two different *assignments*:

$$x := \sqrt{ny^2 \pm 1} \tag{2.5a}$$

$$y := \sqrt{(x^2 \pm 1) \div n} \tag{2.5b}$$

There are a number of advantages of noncausal modelling [Cellier, 1996]:

1. The equations are more reusable: Pell's equation can be used in two

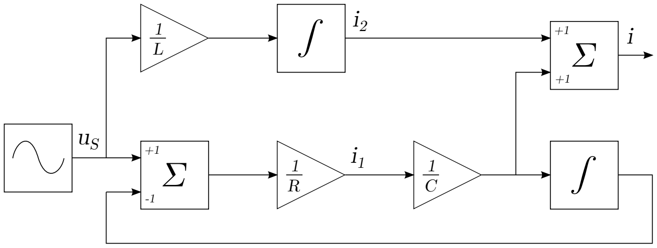

Figure 2.2: Block diagram modelling the simple electrical circuit.

ways, but neither individual assignment is sufficient for capturing the full behaviour of the equation (i.e. depending upon whether x or y is known).

2. Implicit equations are more declarative: the modeller is free to express an equation in whatever way is most clear, without undue concerns about how the larger system is going to be solved.

3. In practice, causal models can be hard to maintain: a small change in the physical structure of the system may have global consequences in the causality of the equations.

To further highlight the advantages of expressing a system implicitly consider a model of the simple electrical circuit in a causal language such as Simulink [Simulink, 1992] by transforming the circuit into the block diagram in Fig. 2.2. The diagram bears little structural resemblance to the physical circuit that it models, and the process of deriving the causal model is in general a difficult task, with the burden of translation resting entirely with the modeller. Consider also the impact of introducing a second resistor to the circuit (see Fig. 2.3), which alters the diagram in a nontrivial and, crucially, *non-compositional* way.

2.1.3 Modularity

The equation systems needed to describe real-world problems are usually large and complex. However, there tends to be a lot of repetitive structure, making it beneficial to describe the systems in terms of reusable equation system fragments

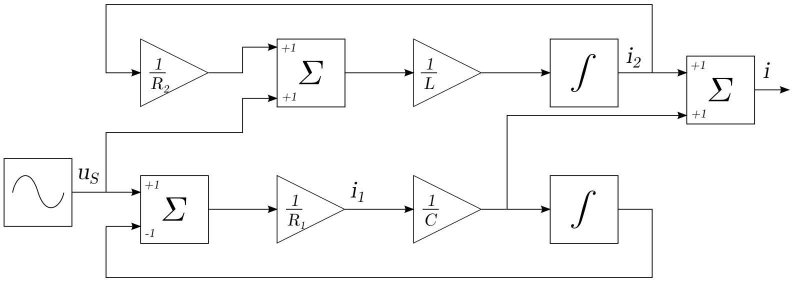

Figure 2.3: Block diagram modelling an electrical circuit with two resistors.

[Cellier and Kofman, 2006]. For example, in the simple electrical circuit, each component can be described by a small equation system, and the entire circuit can then be described *modularly* by composition of *instances* of these for specific values of any parameters. Moreover, the aspects common to each component — for example, the equations that are inherent to any two-pinned electrical component — can be abstracted over and reused.

While the exact syntactic details vary between languages, the idea is to encapsulate a set of equations as a component with a well-defined interface. Let us illustrate with an example, temporarily borrowing the syntax of the λ-calculus for the abstraction mechanism:

$$\lambda(x, y) \rightarrow \begin{array}{rcl} x + y + z & = & 0 \\ x - z & = & 1 \end{array}$$

This abstraction is a relation that constrains the possible values of the two *interface variables* x and y according to the encapsulated equations. The variable z is *local* to the abstraction. If we call the above relation *rel*, it can now be used as a building block by *instantiating* it: substituting expressions for the interface variables and renaming local variables as necessary to avoid name clashes:

$$u + v + w = 10 \tag{2.6a}$$

$$rel(u, v) \tag{2.6b}$$

$$rel(v, w + 7) \tag{2.6c}$$

After unfolding and renaming, often referred to as *flattening* or *elaboration*, the following unstructured (as opposed to modular) set of equations is obtained:

$$u + v + w = 10 \tag{2.7a}$$

$$u + v + z_1 = 0 \tag{2.7b}$$

$$u - z_1 = 1 \tag{2.7c}$$

$$v + (w + 7) + z_2 = 0 \tag{2.7d}$$

$$v - z_2 = 1 \tag{2.7e}$$

The relation *rel* contributes 2 equations for each application. Including the top-level equation, the fully elaborated system thus consists of 5 equations in total over 5 unknowns. Note the need to rename the local variable z when unfolding.

2.1.4 First-class Components

A complete model can be constructed by manipulating and composing individual equation-system fragments (or simply *components*) programmatically. If components are elevated to a first-class status it creates a far more expressive language for *higher-order* and *structurally dynamic* modelling.

The precise meaning of a first-class language entity varies somewhat from field to field, and hence for our purposes we will define it as follows: a language entity is first class if it can be passed as a parameter to functions, returned as a result from functions, constructed at runtime, and be placed in data structures [Scott, 2009]. Thus, for the remainder of this thesis, when referring to language entities as first-class, we will be adhering to the above definition.

To our knowledge, the notion of first-class was first introduced by Christopher Strachey [Burstall, 2000] in reference to functions being first-class values in higher-order, functional programming languages.

2.1.5 Hybrid Systems

A *hybrid system* is a general term for any system that exhibits both continuous-time and discrete-time behaviour. Hybrid systems are very useful in practice as they allow dramatic changes in the behaviour of a system to be expressed easily [Mosterman and Biswas, 1997]. A *cyber-physical* system is an example of a hybrid system that allows digital computers to interact with a continuous physical system to effectuate discrete changes [Lee, 2008].

In order to *model* a hybrid system it must contain both continuous and discrete values. The continuous and discrete parts of the model interact via discrete transitions at distinct points in time. These interactions are known as *events*. In between events, the model evolves continuously: all discrete values remain fixed. Since the model may depend conditionally on the discrete values, each discrete value assignment defines a potentially unique configuration or *mode* of continuous operation. In general, the total number of modes can be enormous, or even unbounded, and often cannot be predicted a priori.

Hybrid systems encompass a broad spectrum of modelling behaviours. For example, as we will discuss in the following section (Sect. 2.1.6), *structural dynamism* provides a means to mix the continuous and discrete by allowing the very equations that model the system to change over time [Nilsson et al., 2003]. *Dirac impulses*, as explored by Nilsson [2003], also allow a form of hybrid modelling for systems that are only piecewise continuous.

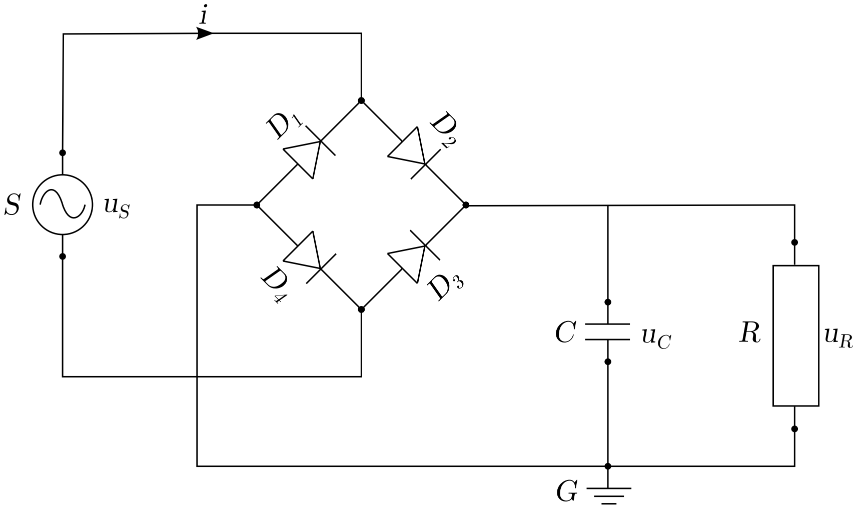

Figure 2.4: Full-wave rectifier modelled using ideal diodes.

2.1.6 Structural Dynamism

In a temporal setting, where equations express relations among time-varying entities, structural dynamism allows the equations of a system to change at various points in time to capture changes in the system configuration. As an example, consider the system in Fig. 2.4 of a full-wave rectifier [Nilsson and Giorgidze, 2010]. The modeller has chosen an ideal model for the diodes: an electrical switch that is closed (diode conducting) whenever the voltage across it is positive, and open otherwise (diode not conducting). Depending on which switches are open and which are closed, there are up to $2^4 = 16$ structural configurations, each corresponding to a distinct system of equations.

Structural dynamism offers a form of *temporal composition* as opposed to the *spatial composition* that arises from constructing an equation system in a modular fashion. This form of temporal composition has been shown to be very useful in practice, particularly in a setting that supports *first-class* components [Giorgidze and Nilsson, 2009]. However, the greater expressivity that comes with structural dynamism also create many problems [Giorgidze and Nilsson, 2009, Nilsson et al., 2003, Nytsch-Geusen et al., 2005, Zimmer, 2010]. Of particular concern in this thesis is the meaning of structurally dynamic systems and the early detection of structural errors. In the latter case, an error in a system with a large — or possibly even unbounded — number of configurations may take a very long time to surface if the error only manifests itself when specific system configurations become active. There are also issues that we do not investigate in the thesis, such as the re-initialisation of variables between events that allow continuity assumptions to be preserved. For example, if one were to model a bouncing ball using a structural configuration for each trajectory, it would be necessary to preserve the position of the ball between events. Initialisation of hybrid systems is known to be a difficult problem in general [Pantelides, 1988], but we do not consider it here as it is not of immediate relevance to our work.

2.2 Modelica

Before we introduce FHM it is both useful and informative to take a look at an alternative approach to noncausal modelling, as illustrated by Modelica. Modelica is a high-level, declarative language for noncausal modelling and simulation of physical systems, and it is the current industry state-of-the-art.

While a familiarity with Modelica is not essential for understanding the work presented in this thesis, it provides a useful reference point for the state of contemporary, noncausal modelling languages. Furthermore, this section demonstrates that there are many core principles in equation-based languages that are not specific to FHM. Thus, we hope that a wider understanding of the field will give the reader a fuller appreciation of the contributions of this thesis.

2.2.1 An *Object-Oriented* Approach

Modelica programs are structured using an *object-oriented* paradigm. It features a class system that is reminiscent of the type systems found in many other modern object-oriented languages, such as C++, Java, or C#. However, unlike these languages where the behaviour of an object is specified via methods according to a particular interface, in Modelica the behaviour of an object is given via equations. Thus, the variables that the equations constrain are the fields of the object along with any parameters that may be passed into it.

As with FHM, equations in Modelica are given via implicit DAEs. A Modelica program can be "run" by compiling the hierarchal equation system into simulation code using a number of sophisticated symbolic techniques, which is then executed to simulate the system.

The approach taken by Modelica is very successful, and understandably so as it builds on many years of research into structuring programs using objects. However, Modelica suffers from a lack of first-class equation system fragments, making it difficult to express higher-order models in a straightforward manner. Modelica has some support for parameterised models by allowing the fields of a model to be replaced in a derived class. For example, a model of a circuit containing a resistor could be refined by deriving a new model that replaces the resistor with a thermistor. However, this notion of parameterisation is very restrictive as the replacement mechanism is entirely static and prevents models from being created dynamically. The lack of first-class models in noncausal languages was one of the primary motivations for FHM [Giorgidze, 2011].

Modelica is also limited in its ability to express structurally dynamic models. It is possible to express simple dynamism through various control flow mechanisms, but Modelica statically rules out the dynamic addition and removal of time-varying variables. It also lacks the ability to perform runtime symbolic processing or to generate new simulation code in response to discrete events during simulation. This means that all possible modes must be statically computable, and the number of modes must be relatively small in order to generate simulation code upfront for every possible structural configuration. This shortcoming was another motivating factor in the design of FHM.

2.2.2 A Simple Model in Modelica

We introduce the Modelica language by way of a small example: the electrical circuit depicted in Fig. 2.1. In keeping with object-oriented philosophy, Modelica makes it easy to create abstractions, and we make use of this by isolating the behaviour that all two-pinned, electrical components have in common:

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

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

A two-pinned component has a positive and negative pin, denoted p and n respectively. A *Pin* is given by a **connector** record that represents electrical connectors. The connector record introduces the variable i, representing the current flowing into the connector, and the variable v, representing the voltage across the connector. A connector does not introduce equations itself, but instead is used in conjunction with **connect** statements to specify the topology of a model. Connectors can be used in any physical domain where both flow variables and potential variables are present; for example, in electrical, hydraulic, and mechanical domains. A discussion of how equations are generated from **connector**s and **connect** statements is deferred until the end of the section.

In addition to a positive and negative pin, the *TwoPin* model also contains a further two quantities: the variable u represents the voltage drop across the component, and i represents the current flowing into the positive pin. Finally, *TwoPin* defines the noncausal equations that these variables satisfy, which correspond to Kirchhoff's laws for electrical circuits [Serway, 2004].

It is now possible to derive specific two-pinned components by *extend*ing *TwoPin* and adding any new quantities and equations as required. This mechanism is a form of *inheritance* that allows models to be highly reusable. Below we define the models representing a resistor, a capacitor, an inductor, and a voltage source. Variables marked as **parameter**s can be set when instantiating a new object of the class, otherwise, parameters take the specified default value.

```
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;
```

We define one more component with a single physical connection to represent the ground component:

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

Finally, the simple circuit can be assembled by composing instances of the components. The Modelica compiler analyses the **connect** statements and appropriate *connect equations* are generated. Connected flow variables generate sum-to-zero equations, and connected potential variables generate equality constraints stating that all connected potential variables are equal at any point in time. For the *SimpleCircuit* model, a total of three sum-to-zero equations and six equality constraints are generated.

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

Modelica is specifically designed to make it easy to abstract over the repetitive structure of large, complex models. The use of noncausal equations makes it easy to express the topology of a circuit, as demonstrated in the above example. In a causal setting, the user would be required to manually causalise the circuit model. Furthermore, components representing the same physical entity may not even be reusable if a different causality is required for the equations in each instance. However, note that Modelica performs all symbolic translations once prior to simulation (i.e. causalisation). In the next section we will see an example of a model that causes changes in causality *during* simulation. In contrast to FHM, such models are rejected at compile time by Modelica as there is not one valid causalisation for every mode.

2.3 Functional Hybrid Modelling

FHM is a high-level functional framework for equation-based modelling that provides a concrete setting for the work presented in this thesis. Hydra is an example of an FHM language and we adopt its syntax to express functional hybrid models until a formal core language is introduced in Chap. 4 and 5. The work in this thesis is applicable to the more general notion of equation-based languages, and much of it is not specific to equation-based *modelling*. Nevertheless, FHM has first-class equation system fragments, spatial and temporal composition, and is embedded in a pure functional language, making it both representative of equation-based languages and also ideal for formal study.

2.3.1 A *Functional* Approach

There are a number of good reasons for why purely functional programming provides a great setting for hybrid modelling. Functional languages typically have more powerful abstraction capabilities compared with their imperative counterparts, such as first-class functions, parametric datatypes, and ad-hoc polymorphism (though many of these features are now finding their way into mainstream imperative languages [Kiselyov, 1999]).

Purely functional languages are usually easier to formally manipulate and reason about, which is particularly relevant in this thesis. Much like Modelica, functional programming is also declarative, which lightens the burden on the end user by simplifying the translation of physical systems into models.

The functional languages that we consider are also equipped with powerful, static type systems. For example, parametricity is useful when trying to ensure that components are used consistently in a system topology that may be changing during simulation. Indeed, a key contribution of this thesis is to refine the type system of a functional modelling language to check a variety of structural properties during compilation.

There are also a number of advantages specific to Haskell, the host language for Hydra. Haskell is well-established as a host language for modelling *causal* physical systems (i.e. Functional Reactive Programming [Hudak, 1999]), due, at least in part, to its support for Embedded Domain Specific Languages (EDSLs). It provides meta-programming and customisable syntax through Template Haskell and Quasi-quoting [Mainland, 2007, Sheard and Peyton Jones, 2002], and provides powerful abstraction mechanisms such as monads for programming with effects [Wadler, 1993].

2.3.2 Functional Reactive Programming

FHM is strongly influenced by Functional Reactive Programming (FRP), and in particular the *Yampa* framework. In many ways, FHM can be viewed as a generalisation of FRP. Thus, we will first cover the fundamentals of FRP.

An FRP language can be considered to have two levels: a time-invariant *functional* level and a time-varying *reactive* level [Wan et al., 2001]. The functional level, typically provided by a functional host language, is a pure functional language into which the reactive level is embedded. The reactive level is concerned with time-varying values called *signals*. At this level, signal combinators are provided to construct a *directed*, synchronous dataflow network. The levels are mutually dependent: the reactive level uses the functional level to compute time-invariant values and perform point-wise computations on signals, while certain reactive objects appear as first-class entities at the functional level. Choosing which reactive objects to promote to first-class functional entities is a matter of design [Sculthorpe, 2011]. In Yampa, it is the functions on signals, rather than the signals themselves, that are given first-class status.

FRP can be understood by a conceptual model. Signals are modelled as functions from continuous time to a value. As time is taken to be continuous we represent it by the set of nonnegative real numbers:

$$\begin{aligned} \mathit{Time} &\simeq \{\, t \in \mathbb{R} \mid t \geqslant 0 \,\} \\ \mathit{Signal}\ \alpha &\simeq \mathit{Time} \rightarrow \alpha \end{aligned}$$

The type parameter α specifies the type of values carried by the signal; for example, *Signal* $\mathbb{N}$ might represent the number of seconds since execution began, or *Signal* $(\mathbb{R}, \mathbb{R})$ might represent the change in position of a moving ball.

$$\mathit{SF}\ \alpha\ \beta \simeq \mathit{Signal}\ \alpha \rightarrow \mathit{Signal}\ \beta$$

Conceptually, a *signal function* is a function on signals. It is these signal functions that are taken as the first-class abstraction in Yampa, while signals have no independent status of their own. A signal function of type $\mathit{SF}\ \alpha\ \beta$ can be applied to an input signal of type $\mathit{Signal}\ \alpha$ to produce an output signal of type $\mathit{Signal}\ \beta$. Interestingly, as a pair of signals (i.e. $(\mathit{Signal}\ \alpha, \mathit{Signal}\ \beta)$) is isomorphic to a signal of a pair of the same types (i.e. $\mathit{Signal}\ (\alpha, \beta)$), unary signal functions are sufficient for handling signal functions of any arity.

It is important to stress the conceptual nature of this model. In a typical digital implementation the continuous time needs to be approximated by sampling the signal over a discrete sequence of time steps. Nevertheless, a conceptual model allows us to abstract away from such details. It allows us to make no assumptions about the rate of sampling, whether the sampling rate is fixed, how sampling is performed, or how to handle numerical inaccuracies. Instead, it provides an *ideal* model of FRP and describes a simple semantics that can be easily understood. Moreover, it can be used as a benchmark, with the expectation that any "reasonable implementation" converges toward the ideal semantics as the sampling interval tends to zero [Wan and Hudak, 2000].

There is also the further caveat that a signal function must be *temporally causal*. This idea should not be confused with equational causality, but instead states that the value of the signal function at any given point in time may only depend on earlier values, and cannot depend on the future.

2.3.3 First-class Signal Relations

The causality of a signal function — or even an ordinary function for that matter — is fixed: it takes a *known* signal as *input* and produces a previously *unknown* signal as *output*. This causality can be eliminated if we generalise the notion of a signal function to a *signal relation*. Rather than specifying which variables are inputs and which are outputs, we simply state that some signals are in a particular relation to each other, imposing constraints on the signals.

Just as an ordinary relation can be seen as a predicate that determines whether some given values are related, a signal relation can also be viewed as a predicate on signals. Here, *Prop* is taken to be the type of propositions:

$$SR\ \alpha \simeq Signal\ \alpha \ \rightarrow\ Prop$$

Solving a relation amounts to finding a valuation for each unknown, such that the constraints imposed by a signal relation are satisfied. As the unknowns are time-varying entities, finding a solution is equivalent to finding a value of type α for all points in time. Thus, the *solution* to a signal relation is a signal. The pairing isomorphism that held for signal functions is also true for signal relations. Thus, solving a relation between many signals is no different from finding the set of signals that satisfy the n-ary predicate.

$$\begin{aligned}&\mathit{equal} :: \mathit{SR}\ (\mathbb{R}, \mathbb{R})\\&\mathit{equal}\ s\ =\ \forall\ t\ :\ \mathit{Time}.\,\mathit{fst}\ (s\ t) \equiv \mathit{snd}\ (s\ t)\end{aligned}$$

The above binary signal relation states that, for all points in time, the two signals are equal. Or equivalently, that the two components of the one signal are equal at all points in time.

2.3.4 Hydra: an FHM Language

Hydra is an FHM language that is heavily inspired by Yampa. It is currently implemented as a Haskell EDSL using quasiquoting [Giorgidze, 2012], but for convenience, we will use an idealised Hydra syntax that avoids the implementation-specific details.

Just like Yampa, Hydra is a two-levelled language embedded in Haskell. However, the primary abstraction mechanism of the signal level is first-class signal relations, as opposed to the signal functions of Yampa. Definitions at the signal level may freely refer to entities defined at the functional level, but signal-level objects are not permitted to escape to the functional level, with the exception of instantaneous values of signals, which may be fed back to the functional level at the time of discrete events. This allows future system configurations to depend on earlier results.

In Hydra, signal relations are constructed using the **sigrel** primitive:

sigrel *pattern* **where** *equations*

This syntax constructs a first-class, time-invariant, function-level object that encapsulates a set of equations. The equations range over signal variables introduced by the pattern, similar to the abstraction mechanism presented in

Sect. 2.1.3. We refer to these signal variables as *interface variables.* Signal variables that occur in the set of equations but not in the pattern are referred to as *local variables.* They do not occur anywhere else in the system.

There are two basic forms of equation:

atomic equation:	$s_1 \;=\; s_2$
signal relation application:	$sr \;\diamond\; s_3$

Here, sr is a time-invariant expression (signal variables must not occur in it) denoting a signal relation, and $\diamond$ denotes signal relation application. The symbols s_1, s_2, and s_3 denote *signal expressions*; that is, a time-varying expression that appears in an equation. Equations do not have types, but their subcomponents are required to be well typed. Thus, in the above, if sr has the type $SR\ \alpha$ then s_3 must have the type $Signal\ \alpha$. Taken together, the two types of equation form a hierarchical system of equations: the atomic equations are leaves representing simple equality constraints, and $\diamond$ allows us to instantiate an equation system fragment with expressions containing the in-scope signals.

To express structurally dynamic systems, Hydra employs a switch construct that allows equations to be introduced and removed from a model as needed:

```
**initially** [; **when** condition] ⇒
  equations_1
**when** condition ⇒
  equations_2
...
**when** condition_n ⇒
  equations_n
```

Only the equations from one branch are active at any one point in time. The equations of a branch are switched in whenever the condition guarding the branch *becomes* true, at which point those from the previously active branch are switched out. Practically speaking, this usually amounts to determining when the value of a signal expression would cross zero (i.e. when it is zero and its left derivative is nonzero).

The keyword **initially** designates the initially active branch. An optional condition allows for the initial branch to be re-activated later. Should more than one switch condition within a switch construct trigger simultaneously, the branches are prioritised syntactically from the top down.

Complications arise due to the need to properly *initialise* the new system of equations after each switch. This is a hard problem in general, but it can be addressed by providing separate initialisation and reinitialisation equations [Nilsson and Giorgidze, 2010].

2.3.5 Hydra by Comparison

The distinction between building and solving equations is an approach that Hydra shares with many other equation-based languages, such as Acumen [Taha et al., 2012], Modelica [Mod, 2012], and Simulink [Simulink, 1992]. Before demonstrating the language features of Hydra by example, it is useful to put Hydra in context by comparing it to other similar languages (see the table below). Our list of comparable languages is far from exhaustive as, at this point, it is not our intention to give a full review of other languages, this task is addressed in Chap. 6. Instead, we wish to show that Hydra occupies a fairly unique point in that it supports first-class components, acausal modelling, and highly-dynamic structure (unbounded dynamism).

Language	**Causality**	**First-class**	**Dynamism**
Hydra	Noncausal	Yes	Highly dynamic
Yampa	Causal	Yes	Highly dynamic
Modelica	Noncausal	Limited	Mostly static structure
Simulink	Causal	No	Static

Languages such as MOSILAB [Nytsch-Geusen et al., 2005] and Sol [Zimmer, 2013] that build upon the Modelica standard fall somewhere between Modelica and Hydra in their support for dynamic structure.

2.3.6 A Simple Model in Hydra

Consider once again the simple circuit example from Fig. 2.1. Following the same approach used for Modelica, we begin by defining an abstract two-pinned circuit component. A pin is represented by a tuple of reals for the quantities of current and potential difference, respectively. For convenience, we allow ourselves access to the subexpressions via the named record fields *i* and *v*, rather than the verbose *fst* and *snd* functions.

```
type Pin = (ℝ, ℝ)
twoPin : SR (Pin, Pin, Voltage)
twoPin = sigrel (p, n, u) where
  p.i + n.i = 0
  p.v − n.v = u
```

To clarify the above example, *twoPin* contains two atomic equations and the symbols *p*, *n*, and *u* are interface variables introduced by the relation. There are no local variables, and hence, the relation constrains a total of five signal variables (recall that each pin contains two signal variables).

```
resistor : Resistance → SR (Pin, Pin)
resistor r = sigrel (p, n) where
  local u
  twoPin ◇ (p, n, u)
  r ∗ p.i = u
```

It is now possible to derive concrete electrical components from the *twoPin* definition. In the example above, *resistor* takes a resistance *r* as a parameter and creates a relation between two pins. Note that a parameterised signal relation is just an ordinary function returning a signal relation. The syntax

local is used to make the quantification of the local variable u explicit, and to make it easily distinguishable from r, which is a time-invariant, functional-level parameter. The local signal variables are not exposed in the pattern. Consequently, u can only be constrained in this signal relation, unlike the rest of the variables in the pattern, which can be constrained further.

From here, we can define models for other two-pin components such as inductors and capacitors in the same way. Note how the *twoPin* signal relation is reused in each case. The keyword **der** indicates the time derivative of a signal.

```
$inductor : Inductance \to SR\ (Pin, Pin)$
$inductor\ i = \mathbf{sigrel}\ (p, n)\ \mathbf{where}$
  $\mathbf{local}\ u$
  $twoPin \diamond (p, n, u)$
  $l * \mathbf{der}\ p.i = u$

$capacitor : Capacitance \to SR\ (Pin, Pin)$
$capacitor\ c = \mathbf{sigrel}\ (p, n)\ \mathbf{where}$
  $\mathbf{local}\ u$
  $twoPin \diamond (p, n, u)$
  $c * \mathbf{der}\ u = p.i$
```

The complete circuit can now be assembled by applying the relevant signal relations and connecting the components together according to the circuit topology. As before, we adopt a special **connect** syntax to denote an electrical circuit junction. However, this time the **connect** statement accepts n arguments, and results in one sum-to-zero equation and $n - 1$ voltage equalities.

```
$simpleCircuit : SR\ ()$
$simpleCircuit = \mathbf{sigrel}\ ()\ \mathbf{where}$
  $\mathbf{local}\ rp, rn, cp, cn, gp$
  $\mathbf{local}\ lp, ln, sp, sn$
  $resistor\ 2200$         $\diamond (rp, rn)$
  $capacitor\ 0.00047$ $\diamond (cp, cn)$
```

$$
\begin{array}{ll}
\mathit{inductor}\ \mathit{0.01} & \diamond\ (\mathit{lp},\ \mathit{ln}) \\
\mathit{vSourceAC}\ \mathit{12} & \diamond\ (\mathit{sp},\ \mathit{sn}) \\
\mathit{ground} & \diamond\ \mathit{gp} \\
\mathbf{connect}\ \mathit{sp}\ \mathit{rp}\ \mathit{lp} & \\
\mathbf{connect}\ \mathit{rn}\ \mathit{cp} & \\
\mathbf{connect}\ \mathit{sn}\ \mathit{cn}\ \mathit{ln}\ \mathit{gp} &
\end{array}
$$

Notice that in the *simpleCircuit* relation the pattern introduced no interface variables. This is because the relation represents a fully assembled system, and hence, all variables are effectively local to the relation.

Before moving on, it is important to understand how *twoPin* contributes to the definitions of the derived electrical components. Let us consider what happens when the *resistor* model is elaborated (or *flattened*). Elaboration proceeds by replacing the applied relation with its body, substituting interface variables for the applied expressions. Again, one needs to be careful to rename local variables during elaboration to avoid name clashes (see 2.1.3). The aim of elaborating a complete modular system of equations is to produce a flat list of equations (i.e. a flat DAE).

$$
\begin{array}{l}
\mathit{twoPin}\ \diamond\ (p, n, u) \\
\mathit{220} * p.i\ =\ u
\end{array}
$$

The above equations are the result of applying *resistor* to *220*. A single step of unfolding eliminates the relation application, producing a flat system of 3 equations: two originating from *twoPin*, and a third contributed by *resistor*.

$$
\begin{array}{lcl}
p.i\ +n.i & = & \mathit{0} \\
p.v\ -n.v & = & u \\
\mathit{220} *\ p.i & = & u
\end{array}
$$

2.3.7 Higher-order Modelling

A higher-order model is a model parameterised on other models. In a setting with first-class signal relations it is easy to express a higher-order model as a

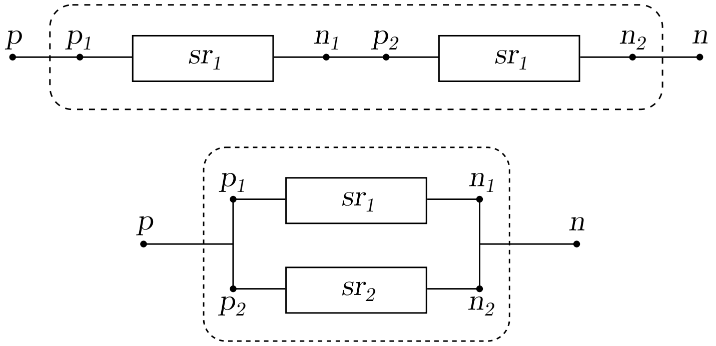

Figure 2.5: Components connected in series and parallel.

model that can be passed into or returned from a function like any other value.

Consider the two different methods of connecting circuit components together given in Fig. 2.5. Both methods share a common interface: they take a pair of two-pinned components as input and return a new two-pinned component as output representing the composite of the two inputs. It is straightforward to model these in Hydra, and the code for *serial* and *parallel* are given below.

```
serial : SR (Pin, Pin) → SR (Pin, Pin) → SR (Pin, Pin)
serial sr1 sr2 =
  sigrel (p, n) where
    local p1 p2 n1 n2
    sr1 ◇ (p1, n1)
    sr2 ◇ (p2, n2)
    connect p p1
    connect n1 p2
    connect n2 n

parallel : SR (Pin, Pin) → SR (Pin, Pin) → SR (Pin, Pin)
parallel sr1 sr2 =
  sigrel (p, n) where
    local p1 p2 n1 n2
    sr1 ◇ (p1, n1)
    sr2 ◇ (p2, n2)
```

connect $p\ p_1\ p_2$

connect $n\ n_1\ n_2$

2.3.8 Structurally Dynamic Modelling

In this final example of the chapter, we look at how Hydra handles structurally dynamic models. We model a "breaking pendulum", which consists of two modes: a swinging pendulum and a freefalling mass. The idea is to model a swinging pendulum until a specific point in time, at which point the pendulum's rod will brake and the pendulum will go into freefall. To highlight the structural aspects as opposed to the physics of a more realistic model, we make a number of simplifying assumptions.

The first mode consists of a point mass m at the end of a massless rod l swinging from a frictionless pivot in a vacuum. That is, we wish to model the simple swinging pendulum given in Fig. 2.6. The second mode consists of the mass m freefalling from an initial position and velocity.

Despite the relative simplicity of the example, the abrupt change in behaviour during simulation makes it very difficult for Modelica to handle. The issue is that a change in causality arises between the two modes. This is problematic as Modelica performs all symbolic transformations (e.g. causalisation) prior to simulation. While there are workarounds for this particular example (that involve the end user manually causalising the system), Modelica is unable to handle these sorts of dynamic models in general.

On the other hand, the breaking pendulum is relatively easy to express in Hydra. The two modes can be modelled as distinct, first-class components. These two components can then be temporally composed using the **when** syntax to express the transition conditions. The current implementation of Hydra [Giorgidze and Nilsson, 2011] uses just-in-time (JIT) compilation to ensure that new generations of equations are generated only when needed. As such, symbolic transformations are performed at the point of a discrete event, making it easy

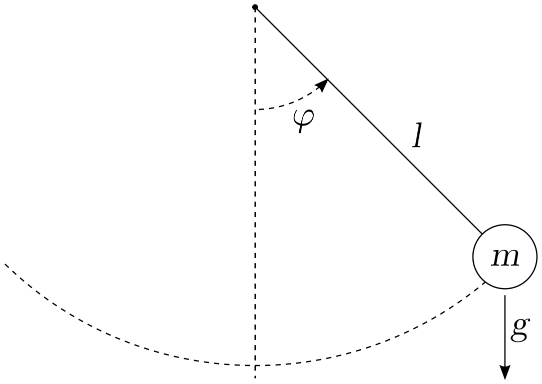

Figure 2.6: Breaking pendulum.

to handle a change in causality between modes.

```
**type** $Mass$ $= \mathbb{R}$
**type** $Pos$ $= (\mathbb{R}, \mathbb{R})$
**type** $Vel$ $= (\mathbb{R}, \mathbb{R})$
**type** $Body$ $= (Pos, Vel)$
$g : \mathbb{R}$
$g = 9.81$
```

Definitions are provide above for the mass, position, and velocity of the pendulum. The force of gravity is approximated by the constant g. The function *pendulum* takes the breaking time, the mass, the rod length, and the initial angle of deviation as parameters. It produces a relation on the body that represents the mass; that is, it provides constraints to determine the position and velocity of the mass at any given time, whether swinging or falling.

```
$pendulum : Time \to Mass \to \mathbb{R} \to \mathbb{R} \to SR\ Body$
$pendulum\ t\ m\ l\ \phi_0$ = **sigrel** $(pos, vel)$ **where**
  **local** $u, \phi, \phi_d$
  **initially** $\Rightarrow$
    **init** $\phi$     $= \phi_0$
    **init** $\phi_d$   $= 0$
    $pos$        $= (l * sin\ \phi, - l * cos\ \phi)$
```

$$
\begin{aligned}
&\quad \phi_d &&= \mathbf{der}\ \phi \\
&\quad m * l * l * \mathbf{der}\ \phi_d + m * g * l * \mathit{sin}\ \phi &&= u \\
&\mathbf{when}\ \mathit{time} - t > \mathit{0} \Rightarrow && \\
&\quad \mathbf{init}\ \mathit{pos} = \mathbf{pre}\ \mathit{pos} && \\
&\quad \mathbf{init}\ \mathit{vel} = \mathbf{pre}\ \mathit{vel} && \\
&\quad m * \mathbf{der}\ \mathit{vel} = m * (\mathit{0}, -g) &&
\end{aligned}
$$

The **initially** branch provides the behaviour of the swinging pendulum. As this branch has no reactivation conditions, it is impossible for the rod to become reattached after it has broken. The **when** branch provides the behaviour of the freefalling mass. In this case, the branch will only ever be activated once as the condition will only ever *become* true on one occasion.

The reader may have noticed that this example included the new syntax **init** and **pre**. The details of **init** and **pre** are not of particular importance in this thesis. They have been included in this example for the purpose of providing an illuminating and working model. The **init** syntax allows us to mark an equation as initial. An initial equation (not to be confused with the equations appearing in the **initially** block) is a constraint that only holds for the first time step in which it is active. For example, when a new branch is switched-in, any equations marked **init** will become active for the time step immediately following the event that caused the switch.

The syntax **pre** allows us to refer to instantaneous values immediately prior to an event. For example, **pre** x might be used in an **init** equation to give the value of x in the time step immediately before an event occurred. Taken together, **init** and **pre** allow continuity assumptions to be preserved. For example, the resistance of a component should not jump discontinuously between modes. In the *pendulum* example, **init** and **pre** are used to preserve the position and velocity of the mass as the pendulum breaks.

The dynamism in *pendulum* is quite simple: it consists of only two modes and only a single discrete event. Despite this, the *pendulum* example highlights the limitations of structural dynamism in Modelica, compared to the relative ease of expression in Hydra. It is entirely possible to create models in Hydra with many more modes, and we will see examples of this in later chapters.

Chapter 3

Agda and Notation

This chapter gives an overview of Agda and its role in this thesis. We discuss how Agda is used to implement and verify much of our work and why Agda is suitable for our purposes. We also discuss specialised notation that is adopted for the sake of presentation. This chapter is not intended to be a tutorial; readers not sufficiently familiar with Agda may consult an introductory text; for example Norell [2009], McBride [2012], or Bove et al. [2009].

3.1 Overview of Agda

Agda, developed by Ulf Norell for his doctoral thesis [Norell, 2007], is both a dependently typed programming language and a proof assistant. Agda is based on Martin-Löf Type Theory (MLTT) [Martin-Löf, 1975] and shares many similarities with other dependently typed proof assistants, such as Coq [Bertot and Castéran, 2004] and Epigram [McBride, 2004]. However, the syntax of Agda is similar to that of Haskell.

The essence of dependent types is that the type of the result of a function may *depend* on the value of its argument (i.e. a dependent function space), and that the inhabitants of a datatype may *depend* on the values of its parameters (i.e. an inductive family). The standard example is the type of vectors of a given length, which are specified via an inductive family indexed on a natural number (i.e. a value). Thus, functions that operate on vectors typically depend on the length of the input vector to compute the output; for example, appending an element to a vector increments its length.

As Agda allows arbitrary terms to appear in types, the language is restricted to allow only *total* and *terminating* programs. This restriction means that, while the language is no longer Turing complete, type checking in Agda remains decidable. Moreover, this feature means that Agda can exploit the Curry-Howard Correspondence [Howard, 1980]. The correspondence states that types can be treated as propositions, and that well-typed programs can be treated as proofs. Uninhabited types represent false propositions and inhabited types represent true propositions, with the constituent programs serving as a witness to this fact. Termination is key here as it rules out non-terminating proofs.

3.2 Agda in this Thesis

Agda is used throughout the remainder of this thesis for several reasons. It has a powerful termination checker, which makes it particularly convenient for showing termination of our programs, and, along with other decision procedures such as pattern coverage checking, it contributes toward showing totality. In many situations, no special considerations need to be made and the checker is able to automatically decide termination without user intervention. In situations where the termination of a program is not apparent to the checker, there exist techniques for providing additional information (e.g. using well-founded recursion [Saaman and Malcolm, 1987], or sized types [Abel, 2010]).

Agda is based on a strong theory of types, which makes it easy to specify and prove theorems about our programs. This is particularly evident in Chap. 5 where we construct a mathematical model of an FHM-like language and use this to prove metatheoretical results, all within Agda itself.

Agda has a flexible syntax that permits Unicode symbols and mixfix operators, meaning that the syntax of our implementations very closely resemble that of Hydra and FHM. Agda is also similar to other popular, contemporary functional languages, such as Haskell, hopefully making the code presented in this thesis accessible to a wide audience.

3.2.1 Implementations and Mechanised Proofs

Most of the work in this thesis has been formalised in Agda. In Chap. 4 we develop two type systems, both of which have been implemented in Agda. These implementations ensure that the type systems are total and terminating. The safety of the refined type systems has not been formalised in Agda.

In Chap. 5 we develop a semantic model and interpreter for an FHM-like language. We prove that this interpreter obeys the equational theory of the language and show a number of corollaries. We provide a semantics of structural dynamism and conclude with two extensions. The implementation and all proofs from this chapter have been formalised in Agda. A complete archive of all code relating to this thesis can be found on the author's website [1].

3.2.2 Notation

The program code in this thesis uses a variety of specialised notations to make it easier to express ideas and to prevent code fragments from becoming too verbose. Our notational conventions are as follows:

- Where appropriate, the introduction of implicit arguments is omitted. This includes dependent function spaces (e.g. $\{a : \mathsf{A}\} \rightarrow \mathsf{B}\ a$ is simplified to $\mathsf{B}\ a$) and inductive families (e.g. data X $\{a : \mathsf{A}\}$ $(b : \mathsf{B}\ a)$: Set where becomes just data X $(b : \mathsf{B}\ a)$: Set where). Any free identifiers present should be assumed to be universally quantified at the top level.

- Implicit arguments that can be inferred by the reader are omitted even if Agda cannot infer them.

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

- We allow ourselves to overload symbols when the definition being used is obvious from the context. For example, we use $\simeq$ to denote program equivalence for both functional terms and equation system fragments.

- We allow ourselves to use ... notation to denote uninteresting code, as is already the case for Agda's `with` notation.

- Haskell uses the :: symbol to denote the "has type" operator, whereas Agda uses the : symbol. We use the latter throughout this thesis for consistency, even when presenting FHM code (a Haskell EDSL). Where the list concatenation operator is needed, we use the :: symbol instead.

Chapter 4

Structural Types

4.1 Preliminaries

4.1.1 Outline

Broadly speaking, the contributions of this thesis can be split into two groups: those contributions concerned with novel type systems for equation-based languages (i.e. enriching the static semantics), and those contributions related to finding a semantic model for equation-based languages (i.e. investigating the dynamics semantics). While these two aspects are deeply interconnected, this chapter will focus on investigating the former, leaving it up to the following chapter (Chap. 5) to explore the latter. However, the relationship between the static and dynamic semantics is such that a discussion of both topics to at least some extent is inevitable in both chapters.

The starting point for the type system developed in this chapter is a system capable of tracking so-called *equation-variable balance*. From here we develop a more sophisticated type system, which captures a much larger class of structural anomalies in equation-based languages. However, even the simple type system advances the state-of-the-art regarding balance checking of equation-based languages. Specifically, this chapter makes the following contributions:

1. A novel type system for modular systems of equations supporting first-class components and structural dynamism.
2. A refined set of structural invariants based on classification of equations, allowing a larger class of structural anomalies to be prevented compared with existing approaches.
3. A concise small-step semantics for the core of FHM, capturing the subtle behaviour of variables in a modular system of equations.
4. A proof of correctness for the simple balance type system.

The basis of the work in this chapter was first published in Capper and Nilsson [2010], the work was then further developed in Capper and Nilsson [2013]. A prerequisite for this chapter is Chap. 2.

4.1.2 Structural Properties

An important question regarding a system of equations is whether or not it has a solution, and if one exists, if said solution is unique. In general, one can only answer this question by studying a complete system of equations where all coefficients are known. Unfortunately, this is in direct opposition to the modular approach discussed in Sect. 2.1.3 as it would rule out the checking of components in isolation. Furthermore, as typical application domains, such as physical systems modelling, necessitate that the form of equations is not unduly restricted, one cannot in general hope to construct a decidable type theory capable of determining if an arbitrary modular system of equations has a solution. For example, a modelling language for physical systems that restricted the systems of equations to be linear would be of very limited practical use.

However, there are simple criteria that while neither necessary nor sufficient for *guaranteeing* solvability, are such that violations of them are likely to be indicative of problems. Indeed, they may even be necessary preconditions for the *specific* approach to solving equations used by a particular tool. Thus, enforcing that such criteria be met through the static semantics of an equation-based language can be very useful, and is in fact often done in practice. The following are two commonly used criteria for checking the well-formedness of systems (see Broman et al. [2006], Bunus and Fritzson [2002], Capper and Nilsson [2010], Mod [2012], Nilsson [2008]):

1. Balanced system: the number of equations and variables are equal.

2. Structurally non-singular system: there is a bijection between the variables and the equations such that each variable is paired with an equation in which it occurs.

Property 2 implies property 1. Note that these properties are strictly structural: no information beyond which variables occur, and in which equations, is assumed. For illustration, consider the following system:

$$x + y = z \qquad (4.1a)$$

$$x + 3 = 12 \qquad (4.1b)$$

$$y^2 + 9 = z^2 \qquad (4.1c)$$

The bijection $\{x \mapsto 4.1b, y \mapsto 4.1c, z \mapsto 4.1a\}$ between the set of variables $\{x, y, z\}$ and the set of equations $\{4.1a, 4.1b, 4.1c\}$ pairs each variable with an equation in which it occurs. Therefore, this system is both balanced and structurally non-singular. Furthermore, if we assume that the above is a conventional algebraic system of equations over the real numbers then it has a solution at $x = 9$, $y = -4$, $z = 5$, and this solution is unique.

On the other hand, it is easy to construct a system that violates the above criteria, but yet still possesses a solution. Consider:

$$x = 2 \qquad (4.2a)$$

$$x^2 + 1 = 5 \qquad (4.2b)$$

This system is neither structurally non-singular nor is it balanced. Yet $x = 2$ is clearly a solution. This shows that the above criteria are not necessary for the existence of a solution. It is also easy to demonstrate that the criteria are not sufficient either. For example, the system below is structurally non-singular, yet possesses no solution:

$$x^2 + y = 4 \tag{4.3a}$$

$$y = 5 \tag{4.3b}$$

Given the above examples, it is reasonable to ask what is it that makes these two criteria useful? The criteria stem from the fact that a *linear* system of equations has a unique solution if and only if the equations are independent and the number of equations and variables agree. If a linear system of equations has more variables than independent equations, it is said to be *underdetermined*. Conversely, if there are more independent equations than variables, it is said to be *overdetermined*. Intuitively, one could interpret each variable as a degree of freedom, and each equation as a constraint that eliminates a degree of freedom (i.e. is used to solve for a variable).

Broadly speaking, this latter intuition is also valid for general systems of equations. In particular, structural non-singularity, which says that there is an equation that can be used to solve for each variable, is *exactly* what is needed for a number of (symbolic and/or numerical) methods that *attempt* to solve general systems of equations. Thus, if a system is structurally singular, commonly used methods will definitely fail to find a solution.

The balance criterion is a coarse approximation of structural non-singularity, essentially assuming that any equation can be used to solve for any variable. However, the criterion is easy to check, and if violated it implies that the system is certainly structurally singular. On the other hand, even though neither criterion is necessary for the existence of a solution, insisting that the criteria be met is not overly restrictive in practice. For rare cases where an over-constrained system is a practical necessity (e.g. see work by Nilsson and Giorgidze [2010] on initialisation problems), one can imagine relaxing the rules by providing simple language mechanisms that allow the user to selectively turn off structural checks. Such an approach would not undermine the benefit of continuing to check the rest of the system. Consequently, both criteria constitute useful static checks that can help find errors early during compilation.

4.2 A Type System for Simple Balance

4.2.1 Key Ideas

In this section we develop a type system that captures the first structural requirement outlined earlier in Sect. 4.1.2, that of *equation-variable balance*.

In Chap. 2 the equation-based modelling language framework of Functional Hybrid Modelling was presented, along with a rough outline of its type system. In particular, a syntax for encapsulating a modular set of equations is introduced and associated with a new type constructor *SR*. By precisely formulating the type system of FHM, we show in this section how the type constructor *SR* can be refined to capture the desired structural properties.

The key idea is to annotate *SR* with an additional parameter that denotes the number of equations a signal relation is capable of *contributing* to the wider system. Intuitively, the *contribution* of a relation is the number of excess equations that are not required to solve for the local variables. At this point we are not concerned with which variables occur in which equations. Instead, it is assumed that any equation may solve for any variable. Thus, a signal relation composed of n equations and m local variables will contribute $n - m$ equations when $n > m$, and otherwise will be under-determined.

As FHM permits higher-order models a signal relation may be parameterised on other relations, and thus, its type may have a parametric contribution. As a result, a notion of *balance variables* is required to express polymorphic signal relations that contribute a varying number of equations depending on the context in which they are used.

To illustrate the above, consider the resistor example from Chap. 2:

$$
\begin{aligned}
&\mathit{resistor} \;:\; \mathit{Resistance} \;\to\; \mathit{SR}\;(\mathit{Pin}, \mathit{Pin})\;\mathit{2}\\
&\mathit{resistor}\;r \;=\; \textbf{sigrel}\;(p, n)\;\textbf{where}\\
&\quad \textbf{local}\;u\\
&\quad \mathit{twoPin} \;\diamond\; (p, n, u)\\
&\quad r * p.i \;=\; u
\end{aligned}
$$

A contribution of two equations is determined for resistor, which is easy to justify: the application of *twoPin* contributes two equations, the body of *resistor* adds a single equation, but one equation must be deducted from the overall contribution to account for the local variable u. To see balance variables in action we need to consider a higher-order model. For example, take the *par* function below, which performs the parallel composition of a two-pinned electrical component with itself:

$$
\begin{aligned}
&\mathit{par} \;:\; \mathit{SR}\;(\mathit{Pin}, \mathit{Pin})\;n \;\to\; \mathit{SR}\;(\mathit{Pin}, \mathit{Pin})\;(\mathit{2}n - \mathit{2})\\
&\mathit{par}\;\mathit{sr} \;=\; \textbf{sigrel}\;(p, n)\;\textbf{where}\\
&\quad \textbf{local}\;p_1\;p_2\;n_1\;n_2\\
&\quad \mathit{sr} \;\diamond\; (p_1, n_1)\\
&\quad \mathit{sr} \;\diamond\; (p_2, n_2)\\
&\quad p.i \;+\; p_1.i + p_2.i \;=\; \mathit{0}\\
&\quad n.i \;+\; n_1.i + n_2.i \;=\; \mathit{0}\\
&\quad p.v \;=\; p_1.v\\
&\quad p.v \;=\; p_2.v\\
&\quad n.v \;=\; n_1.v\\
&\quad n.v \;=\; n_2.v
\end{aligned}
$$

Given a relation *sr* contributing n equations, *par* returns a new relation contributing $\mathit{2}n - \mathit{2}$ equations. As before, this can just be computed in a bottom-up fashion: $\mathit{2}n$ is contributed from the two applications of *sr* and $-\mathit{2}$

equations are contributed from the number of atomic equations minus the number of local variables. Recall that each pin contains two variables and as such the variables p_1, p_2, n_1, and n_2 account for *8* local unknowns, which when subtracted from the *6* atomic equations, result in a contribution of $-\mathit{2}$.

The remainder of this section is dedicated to describing and formalising the simple balance type system. We develop a new core language, giving a rigorous account of the semantics and typing rules. Moreover, we show novel metatheoretical results for the new type system, specifically, we present a proof for preservation of balance with respect to the semantics.

4.2.2 H_Δ: A Core Language for Simple Balance

A precise account of a type system first demands a precise account of the programming language upon which it is built. To that end, the first step to formalising the above intuition is to make precise the object language of study; the goal of this subsection. This guideline holds true not only for the simple balance system but for type systems in general. Hence, the language presented in this section will be the first of several in this thesis, each one designed to focus only on aspects that are necessary and relevant to their respective type systems. We designate the language of simple balance H_Δ.

The simple balance type system aims to be applicable not just to FHM but to equation-based modelling languages in general. Where then should one start when designing a syntax for such a language? A formalisation of Hydra alone would be complex, monotonous, and likely impenetrable as a resource to understand the type system (e.g. one would need to formalise module systems, pattern matching, and expansion of syntactic sugar, to name but a few difficult aspects). A more reasonable compromise would be to design a *core* language that captures the essence of FHM without introducing unnecessary details.

Notionally, there is still a tension between the desire to design a core language and theory of *general applicability*, and opting for a *FHM-like* core language. However, an FHM-like language with support for both spatial composition (modular equation fragments) and temporal composition (equations that evolve over time) is a very general setting, covering a large number of conceivable concrete domains. Despite our choice to use a syntax specific to FHM, for example, the use of relations on signals, it should be straightforward to translate our methods to similar languages such as Modelica or MKL [Broman, 2010].

The first simplification we will make here is an obvious one: to replace the functional host language of Haskell with the λ-calculus. In Sect. 2 we saw how Hydra was partitioned into two levels: the time-invariant functional host level (now portrayed by the λ-calculus), and the time-variant signal level. A language of equations mediates between these two levels, allowing functional values to be embedded as constants at the signal level. Accordingly, a mediating syntactic layer of equations is also included in this core language. As it turns out, a syntax for signal expressions is unnecessary in H_Δ. At first, this might seem like a significant departure from the languages envisioned by FHM. However, the simple balance type system is unconcerned with the shape of signal expressions. In fact, it is entirely unconcerned about signal expressions at all, but instead only about the number of equations and local variables that occur in a signal relation. In later sections, core languages that incorporate a notion of signal expressions are explored. However, even in these languages the shape of such expressions is mostly orthogonal. After all, this thesis is primarily interested in techniques that are applicable to equation-based languages in general, and not just to specific classes of equation systems (e.g. linear systems of equations)

A typed language certainly needs a language of types, which we spell out in Fig. 4.1. At the functional level we permit either function spaces or signal relation types. Ordinarily, signal relations would be required to carry the type of the signal that they constrain (e.g. Pin). However, by doing away with signal expressions we also obviate the need for signal types. To reiterate, in later sections we present a core language that is not excused from such details, as they will be relevant and interesting in such a context. In the current context,

$$
\begin{array}{lr}
\tau ::= & \text{type:} \\
\quad \tau_1 \;\rightarrow\; \tau_2 & \text{function space} \\
\quad \mathit{SR}\ e & \text{signal relation} \\
\quad \forall\ b\,.\,\tau & \text{balance abstraction} \\
 & \\
\nu ::= & \text{equation type:} \\
\quad \mathit{Eq}\ e & \text{equation} \\
 & \\
e ::= & \text{balance:} \\
\quad \mathbb{Z} & \text{integer} \\
\quad b & \text{variable} \\
\quad e_1 + e_2 & \text{addition}
\end{array}
$$

Figure 4.1: H_Δ types.

however, we are interested in the contribution of a signal relation, which is described by the language of *constraint expressions* e.

Functional types also permit us to abstract over balance variables, denoted by the $\forall$ symbol, allowing a system to be polymorphic in its balance. While this concept is notionally similar to type abstraction found in second-order calculi such as System F, the quantification in our system does not range over the language of types and thus the usual difficulties associated with proving properties of these systems do not arise (see Girard [1972]).

Constraint expressions are either integers, variables or addition. In other words, we use the monoid $\langle\ \mathbb{Z}, +\ \rangle$, as such a uniform structure will prove to be very useful in later sections when reasoning about equality.

The language of types also add something that was not seen in FHM: the category ν provides types for equations. Previously, equations were only required to be well-typed insofar as requiring that the constituent components be well-typed. However, in H_Δ, equations are also annotated with their contribution, in much the same way as signal relations.

Looking at the language of terms and equations given by t and q respectively, we can see how the language is partitioned. The terms consist of the standard productions from the λ-calculus (variables, abstraction, application, and let bindings) and additionally a syntax for constructing signal relations. In the

$t ::=$	term:
x	variable
$t_1\ t_2$	application
$\lambda\ x\ :\ \tau\,.\,t$	abstraction
$\textbf{let}\ x\ =\ t_1\ \textbf{in}\ t_2$	let binding
$\textbf{sigrel}\ i\ l\ \textbf{where}\ q$	signal relation
$\Lambda\ b\,.\,t$	balance abstraction
$t\ [e]$	balance application
$q ::=$	equation:
$\textbf{atomic}$	atomic
$q_1 \wedge q_2$	pairing
$t\ \diamond$	application
sw	switch block
$sw ::=$	switch:
$\textbf{initially}\ q$	initial branch
$sw\ \textbf{when}\ q$	event branch
$i, l ::=$	variable accumulator:
$\mathbb{Z}$	integer

Figure 4.2: H_Δ terms.

absence of signal expressions, we provide **sigrel** with two new parameters that record the number of interface variables and local variables occurring in the body of the relation; information that is easy to compute statically.

The syntax of terms also provides constructs for abstracting over balance expressions ($\Lambda\ b\ .\ t$) and balance application ($t\ [e]$). These constructs are related to their abstraction and application counterparts in System F, in that they permit balance variables to occur in signal types. However, to reiterate our earlier remark, the abstraction mechanism in our system is much simpler as it only allows abstraction over balance expressions, and thus does not create an impredicative hierarchy of types, as is the case in System F.

The category q describes equations as a non-empty tree of atomic equations, relation applications, and switches, the syntax of which is provided by sw. However, we have simplified the syntax as we do not consider signal expressions in this formulation. Instead, Hydra equations of the form $s_1\ =\ s_2$ will simply be written **atomic**, and equations of the form $t\ \diamond\ (s_1, ..., s_n)$ will simply be written as $t\ \diamond$. Switch blocks describe a list of equations. The production **initially** denotes the initially active branch, while **when** denotes the branch that may become active during simulation. Continuing to follow our policy of omitting signal expressions, we can omit the signal expression responsible for describing the switching condition. We are not interested at this stage with describing the semantics of switching, and so defer this discussion to Chap. 5.

Fig. 4.3 provides the syntax of values. In both halves of the partition, the grammars have done away with application, as these will either have been redexes that are eliminated, or they will appear under a binder. The latter case is acceptable as we will not be attempting reduction under binders.

A switch value requires only that the **initially** branch be reduced. There are a number of reasons why it might be desirable to defer reduction of a branch until it becomes active, which are discussed in depth in Sect. 5.5.2 when we consider the semantics of dynamism in earnest. For the purposes of this chapter, it is sufficient to point out that, from an operational standpoint, premature reduction of branches is in general wasteful as a branch may never be activated. Moreover, the number of possible structural configurations may be very large, and in a setting supporting general recursion (note that our **let**-expressions are not recursive) even unbounded [Giorgidze and Nilsson, 2009].

A Comparison to Hydra

In the pursuit of dispelling any confusion about H_Δ, we will take a quick detour to relate our new core language back to Hydra and FHM. In Fig. 4.4 the implementation of the *par* function in each language can be seen side-by-side.

Immediately of note is the extent to which the core language simplifies the equations appearing in the body of the signal relation. Nonetheless, the core language still captures the structure of Hydra. Indeed, H_Δ is in fact imposing slightly more structure on the equations. The pairing operation ($\wedge$) implicitly associates to the left creating a list of equations as opposed to a set. This will

v ::=	value:
$\lambda\, x\ :\ \tau\, .\, t$	abstraction
$\Lambda\, b\, .\, t$	balance abstraction
$\mathbf{sigrel}\ i\ l\ \mathbf{where}\ qv$	signal relation
qv ::=	equation value:
atomic	atomic equation
$qv_1 \wedge qv_2$	pairing
sv	switch block
sv ::=	switch value:
$\mathbf{initially}\ qv$	initial branch
$sw\ \mathbf{when}\ q$	event branch

Figure 4.3: H_Δ values.

allow us to prescribe a simple notion of reduction to signal relation application.

Also of note is the need to explicitly abstract over the balance variable that is used as a parameter to the incoming signal relation. Furthermore, the explicit introduction of local variables has been replaced by the pair of natural numbers (i.e. *4* and *8*) that denote the number of signal variables and local variables in scope in the body of the relation.

4.2.3 Semantics

Describing the meaning of a program can be rife with difficulties, not least, due to the number of different approaches available, each with their own advantages and disadvantages. Of particular interest to this thesis are denotational semantics and operational semantics. The former describe the meaning of a program by constructing a mathematical object, whereas the latter describe how a program can be interpreted as a sequence of computational steps. Of course, the distinction is not always so clear cut and both topics have been of great interest to computer scientists for several decades.

A number of factors are often considered when attempting to settle on a specific approach. For example, operational semantics tend to be quite simple and intuitive compared with their denotational counterpart, describing the valid

$$
\begin{array}{ll}
\mathit{par}\ \mathit{sr}\ =\ \textbf{sigrel}\ (p, n)\ \textbf{where} & \mathit{par}\ =\ \Lambda\ b\,.\,\lambda\ \mathit{sr}\ :\ \mathit{SR}\ b\,. \\
\quad \textbf{local}\ p_1\ p_2\ n_1\ n_2 & \quad \textbf{sigrel}\ \mathit{4}\ \mathit{8}\ \textbf{where} \\
\quad \mathit{sr}\ \diamond\ (p_1, n_1) & \qquad \mathit{sr}\ \diamond\ \wedge \\
\quad \mathit{sr}\ \diamond\ (p_2, n_2) & \qquad \mathit{sr}\ \diamond\ \wedge \\
\quad p.i\ +\ \ p_1.i + p_2.i\ =\ \mathit{0} & \qquad \textbf{atomic}\ \wedge \\
\quad n.i\ +\ \ n_1.i + n_2.i\ =\ \mathit{0} & \qquad \textbf{atomic}\ \wedge \\
\quad p.v\ =\ p_1.v & \qquad \textbf{atomic}\ \wedge \\
\quad p.v\ =\ p_2.v & \qquad \textbf{atomic}\ \wedge \\
\quad n.v\ =\ n_1.v & \qquad \textbf{atomic}\ \wedge \\
\quad n.v\ =\ n_2.v & \qquad \textbf{atomic}
\end{array}
$$

Figure 4.4: Comparison of Hydra and H_Δ.

computational steps on some suitably abstract machine. In a sense, operational semantics can be viewed as *interpreting* the terms of the object language. Conversely, denotational semantics use structural induction to define a valuation function that maps programs to elements of some suitably chosen model. An important characteristic of denotational semantics is that they are *compositional*: the meaning of a program fragment is defined directly in terms of the meaning of its respective components.

In this section we opt to use *small-step* operational semantics. The goal of the semantics is to help the reader understand the simple balance type system and its interaction with evaluation. Thus, the benefits of an operational semantics are a simple and clear presentation at the expense of some desirable mathematical properties (such as compositionality).

In Chap. 5 we revisit the problem of providing a semantics for Hydra where we opt for a denotational approach over the operational approach used in this chapter. Constructing a denotational model for Hydra is difficult due to the non-standard language features, the details of which need to be spelled out carefully. It is for this reason that we choose an operational semantics in this chapter as it provides an appropriate foundation for the simple balance type system. In particular, it makes for a simple formalisation that can be used to check the safety of the type system, allowing us to focus on the problems of immediate interest. However, unlike the denotational semantics in Chap. 5 the operational semantics is not compositional.

The semantics in Fig. 4.5 reveals an interesting peculiarity of FHM; the treatment of local variables and interface variables differs. In a richer language, such as Hydra, the interface variables are treated in much the same way as λ-bound variables in the λ-calculus, naming a token that will later be substituted for during reduction. On the other hand, local variables will never be substituted away but are instead accumulated as reduction proceeds. Precisely capturing the behaviour of variables is crucial for giving an honest account of the meaning of an FHM program. To this end, the semantics are specified via two relations: $t_1 \longrightarrow t_2$, the term t_1 reduces to t_2 in one step, and $q_1 \xrightarrow{l} q_2$, the equation q_1 reduces to q_2 and introduces l new local variables in the process.

The first relation is the usual notion of reduction in an operational semantics for the functional terms. To understand the second relation take the rule S-SigRel as an example, which reduces the equations contained within a relation and adds any new local variables discovered this way to the context. Recall that in the core language we are only interested in maintaining the *number* of local variables, and hence do not store the list of variables directly.

4.2.4 A H_Δ Type System

The purpose of the typing rules is to express a minimal set of axioms to type check the terms of our language. Moreover, the rules also suggest a simple algorithm for traversing a tree of equations and accumulating the number of equations that a compound equation is capable of contributing.

The rules (see Fig. 4.6) relate four aspects: the functional context Γ, the balance context Δ, a syntactic category of terms, and an associated category

$$\boxed{t_1 \ \longrightarrow\ t_2}$$

$$\frac{}{(\lambda\ x\ :\ \tau\,.\,t)\ v\ \longrightarrow\ [x\ \mapsto\ v]\ t}\quad(\text{S-AppAbs})$$

$$\frac{t_1\ \longrightarrow\ t_2}{t_1\ [e]\ \longrightarrow\ t_2\ [e]}\quad(\text{S-TApp})\qquad\qquad \frac{t_1\ \longrightarrow\ t_2}{t_1\ t_3\ \longrightarrow\ t_2\ t_3}\quad(\text{S-App1})$$

$$\frac{t_1\ \longrightarrow\ t_2}{v\ t_1\ \longrightarrow\ v\ t_2}\quad(\text{S-App2})\qquad\qquad \frac{}{(\Lambda\ b\,.\,t)\ [e]\ \longrightarrow\ [b\ \mapsto\ e]\ t}\quad(\text{S-TAppAbs})$$

$$\frac{q_1\ \xrightarrow{l_2}\ q_2}{\textbf{sigrel}\ i\ l_1\ \textbf{where}\ q_1\ \longrightarrow\ \textbf{sigrel}\ i\ (l_1 + l_2)\ \textbf{where}\ q_2}\quad(\text{S-SigRel})$$

$$\frac{t_1\ \longrightarrow\ t_2}{\textbf{let}\ x\ =\ t_1\ \textbf{in}\ t_3\ \longrightarrow\ \textbf{let}\ x\ =\ t_2\ \textbf{in}\ t_3}\quad(\text{S-Let})$$

$$\frac{}{\textbf{let}\ x\ =\ v\ \textbf{in}\ t\ \longrightarrow\ [x\ \mapsto\ v]\ t}\quad(\text{S-LetV})$$

$$\boxed{q_1\ \xrightarrow{l}\ q_2}$$

$$\frac{q_1\ \xrightarrow{l}\ q_3}{q_1\ \wedge\ q_2\ \xrightarrow{l}\ q_3\ \wedge\ q_2}\quad(\text{S-Pair1})$$

$$\frac{q_1\ \xrightarrow{l}\ q_2}{qv\ \wedge\ q_1\ \xrightarrow{l}\ qv\ \wedge\ q_2}\quad(\text{S-Pair2})\qquad\qquad \frac{t_1\ \longrightarrow\ t_2}{t_1 \diamond\ \xrightarrow{0}\ t_2 \diamond}\quad(\text{S-RApp})$$

$$\frac{}{(\textbf{sigrel}\ i\ l\ \textbf{where}\ qv) \diamond\ \xrightarrow{l}\ qv}\quad(\text{S-RAppAbs})$$

$$\frac{q_1\ \xrightarrow{l}\ q_2}{\textbf{initially}\ q_1\ \xrightarrow{l}\ \textbf{initially}\ q_2}\quad(\text{S-Initial})$$

$$\frac{sw_1\ \xrightarrow{l}\ sw_2}{sw_1\ \textbf{when}\ q\ \xrightarrow{l}\ sw_2\ \textbf{when}\ q}\quad(\text{S-When})$$

Figure 4.5: H_Δ small-step semantics.

of types. For example, the relation $\Gamma, \Delta \;\vdash\; t \;:\; \tau$ states that a functional term t has type τ in the contexts Γ and Δ. The context Γ is the usual typing context used to store λ-bound variables and their associated types. The context Δ records the list of balance variables currently in scope.

The strategy is exemplified by the rule T-SigRel that allows us to check the type of a signal relation by computing the contribution of its constituent equations and then removing enough equations to solve for the local variables. In a similar vein, T-Pair aggregates the contribution of its sub-equations, and T-RelApp states that an applied signal relation simply contributes the same number of equations as the relation being applied. The T-SigRel rules includes an additional assumption, specifically that $e \geqslant l$. As e is an expression containing bound balance variables, it may not be possible to determine upfront if this assumption holds. Rather than attempting to encode this in the rules, the problem is left as a quality of implementation issue. Thus, in any reasonable implementation, when applying the rule T-SigRel or T-TApp, a decision procedure can check that their still exist possible instantiations of the balance available such that the contribution of the relation would be non-negative.

The rules T-Initially and T-When provide a means to check the contribution of a switch block by requiring that each branch have an equal contribution. Rather than require that the contributions be syntactically equal or even definitionally equal, we submit a more flexible notion of equality (denoted by $\simeq_e$), which requires only that contributions be equal up to the laws of the abelian group of integers with addition. There are a number of suitable candidates for equality of contribution, and the story becomes even more complicated, and the choices more numerous, when we begin to consider the constraint-based approach in the following section. As a result, a thorough discussion of such matters is deferred until Sect. 4.3.2.

The rules of the functional aspect of the language should not come as much of a surprise as they are mostly identical to those of the simply-typed λ-calculus. This makes for a very simple presentation of the balance type system, something that will unfortunately not be preserved in the constraint-based approach.

4.2.5 Preservation of Balance

The formalisation of a mathematical system is often motivated by the desire to show that certain properties hold for that system. These metatheoretical properties, while not the only motivation for our work, give us assurances that the system we have designed is logical and well behaved.

The soundness of a type system is often specified via two properties: *progress* and *preservation* (also known as subject reduction) [Pierce, 2002, p. 95]. A type system has progress if, for every closed well-typed term t, either t is a value or else there exists some t' for which $t \longrightarrow t'$. A type system has subject reduction if the reduction of an expression preserves the type of that expression. Type equality, which we shall denote $\Delta \vdash \tau \simeq_\tau \sigma$ is an essential notion in stating the latter. In H_Δ we have designed type equality to express that the types τ and σ should be equal up to equality of balance expressions within the same context of balance variables (given by Δ), see Fig. 4.7. The rules are the normal laws of equivalence, congruences, and the monoidal laws of $\langle\, \mathbb{Z}, + \,\rangle$. The E-Forall rule allows terms to be equal up to α-equivalence of balance variables.

Due to the mutual dependencies between functional- and equational-level terms in H_Δ, proofs of the aforementioned safety properties for both levels are also necessarily mutually dependent. We will begin by formally stating both properties and then first present proofs for the equational level, making forward references to the functional-level proofs that follow. For each pair of proofs of this nature, we take special care to assume only induction hypotheses on structurally smaller terms, ensuring that our proofs remain terminating.

The following proofs make use of a few basic lemmas. Firstly, an inversion principle states that if a term is well typed, then its subterms are also well typed. The shape of the subterms corresponds to the premises of the typing rules, and as there is only one typing rule for each syntactic form, the inversion

$$\boxed{\Gamma, \Delta \vdash t : \tau}$$

$$\frac{\Gamma \triangleright x : \tau_1, \Delta \vdash t : \tau_2}{\Gamma, \Delta \vdash \lambda\, x : \tau_1 . t : \tau_1 \to \tau_2} \quad (\textsc{T-Abs})$$

$$\frac{\Gamma, \Delta \vdash t : \forall\, b . \tau}{\Gamma, \Delta \vdash t\,[e] : [b \mapsto e]\,\tau} \quad (\textsc{T-TApp}) \qquad \frac{\begin{array}{c}\Gamma, \Delta \vdash t_2 : \tau_1 \\ \Gamma, \Delta \vdash t_1 : \tau_1 \to \tau_2\end{array}}{\Gamma, \Delta \vdash t_1\, t_2 : \tau_2} \quad (\textsc{T-App})$$

$$\frac{\Gamma, \Delta \triangleright b \vdash t : \tau \qquad b \notin \mathit{free}\,(\Gamma)}{\Gamma, \Delta \vdash \Lambda\, b . t : \forall\, b . \tau} \quad (\textsc{T-TAbs}) \qquad \frac{x : \sigma \in \Gamma}{\Gamma, \Delta \vdash x : \sigma} \quad (\textsc{T-Var})$$

$$\frac{\Gamma, \Delta \vdash_q q : \mathit{Eq}\ e \qquad e \geqslant l}{\Gamma, \Delta \vdash \textbf{sigrel}\ i\ l\ \textbf{where}\ q : \mathit{SR}\ (e - l)} \quad (\textsc{T-SigRel})$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash t_1 : \tau_1 \\ \Gamma \triangleright x, \Delta : \tau_1 \vdash t_2 : \tau_2\end{array}}{\Gamma, \Delta \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : \tau_2} \quad (\textsc{T-Let})$$

$$\boxed{\Gamma, \Delta \vdash_q q : \nu}$$

$$\frac{\Gamma, \Delta \vdash t : \mathit{SR}\ e}{\Gamma, \Delta \vdash_q t \diamond : \mathit{Eq}\ e} \quad (\textsc{T-RelApp})$$

$$\frac{\Gamma, \Delta \vdash_q q_1 : \mathit{Eq}\ e_1 \qquad \Gamma, \Delta \vdash_q q_2 : \mathit{Eq}\ e_2}{\Gamma, \Delta \vdash_q q_1 \wedge q_2 : \mathit{Eq}\ (e_1 + e_2)} \quad (\textsc{T-Pair})$$

$$\frac{}{\Gamma, \Delta \vdash_q \textbf{atomic} : \mathit{Eq}\ \mathit{1}} \quad (\textsc{T-Atomic})$$

$$\boxed{\Gamma, \Delta \vdash_{sw} sw : \nu}$$

$$\frac{\Gamma, \Delta \vdash_q q : \nu}{\Gamma, \Delta \vdash_{sw} \textbf{initially}\ q : \nu} \quad (\textsc{T-Initial})$$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash_{sw} sw : \mathit{Eq}\ e_2 \\ \Gamma, \Delta \vdash_q q : \mathit{Eq}\ e_1 \qquad e_1 \simeq_e e_2\end{array}}{\Gamma, \Delta \vdash_{sw} sw\ \textbf{when}\ q : \mathit{Eq}\ e_1} \quad (\textsc{T-When})$$

Figure 4.6: H_Δ typing rules.

$$\boxed{\Delta \vdash \tau \simeq_\tau \sigma}$$

$$\frac{\Delta \vdash \tau_1 \simeq_\tau \tau_2 \qquad \Delta \vdash \sigma_1 \simeq_\tau \sigma_2}{\Delta \vdash \tau_1 \to \sigma_1 \simeq_\tau \tau_2 \to \sigma_2} \quad (\text{E-}\textsc{Fun})$$

$$\frac{c \notin \Delta \qquad \Delta \triangleright c \vdash [a \mapsto c]\,\tau \simeq_\tau [b \mapsto c]\,\sigma}{\Delta \vdash \forall\, a . \tau \simeq_\tau \forall\, b . \sigma} \quad (\text{E-}\textsc{Forall})$$

$$\frac{\Delta \vdash e_1 \simeq_e e_2}{\Delta \vdash \mathit{SR}\, e_1 \simeq_e \mathit{SR}\, e_2} \quad (\text{E-SR}) \qquad \frac{\Delta \vdash e_1 \simeq_e e_2}{\Delta \vdash \mathit{Eq}\, e_1 \simeq_\nu \mathit{Eq}\, e_2} \quad (\text{E-E}\textsc{q})$$

$$\boxed{\Delta \vdash e_1 \simeq_e e_2}$$

$$\frac{}{\Delta \vdash e_1 + e_2 \simeq_e e_2 + e_1} \quad (\text{E-EA}\textsc{dd}\text{C}\textsc{omm})$$

$$\frac{\Delta \vdash e_1 \simeq_e e_2 \qquad \Delta \vdash e_2 \simeq_e e_3}{\Delta \vdash e_1 \simeq_e e_3} \quad (\text{E-ET}\textsc{rans})$$

$$\frac{}{\Delta \vdash (e_1 + e_2) + e_3 \simeq_e e_1 + (e_2 + e_3)} \quad (\text{E-EA}\textsc{dd}\text{A}\textsc{ssoc})$$

$$\frac{}{\Delta \vdash e + \mathit{0} \simeq_e e} \quad (\text{E-EU}\textsc{nit}) \qquad \frac{}{\Delta \vdash e \simeq_e e} \quad (\text{E-ER}\textsc{efl})$$

$$\frac{\Delta \vdash e_2 \simeq_e e_1}{\Delta \vdash e_1 \simeq_e e_2} \quad (\text{E-ES}\textsc{ym})$$

$$\frac{\Delta \vdash e_1 \simeq_e e_3 \qquad \Delta \vdash e_2 \simeq_e e_4}{\Delta \vdash e_1 + e_2 \simeq_e e_3 + e_4} \quad (\text{E-EA}\textsc{dd}\text{C}\textsc{ong})$$

Figure 4.7: H_Δ type equality.

lemma follows directly from the typing relation. We also make use of a simple substitution lemma: $\Gamma, \Delta \triangleright b + P \vdash t : \tau$ implies $\Gamma, \Delta + P \vdash [b \mapsto e]\ t : [b \mapsto e]\ \tau$ (where $\Delta + P$ is concatenation of contexts). This is simply a translation of the well-known *substitution lemma* to the type-level [Pierce, 2002]. As given below, we also establish unicity of typing.

Lemma 1. (Unicity of Typing) *Given a term t, if $\Gamma, \Delta \vdash t : \tau_1$ and $\Gamma, \Delta \vdash t : \tau_2$ then $\Delta \vdash \tau_1 \simeq_\tau \tau_2$.*

Proof. There is exactly one typing judgement for each syntactic construct. Thus, the proof follows immediately from induction on the structure of t and applications of the induction hypothesis. Note that E-Forall allows us to show the equivalence for the $\Lambda\ b\,.\,t$ case. □

Lemma 2. (Equational Subject Reduction) *Given two equations q and q', if $q \xrightarrow{l} q'$ and $\Gamma, \Delta \vdash_q q : \nu$ then $\Gamma, \Delta \vdash_q q' : \mu$ where $\Delta \vdash \nu \simeq_\nu \mu - l$.*

Proof. The proof, as with many others in this thesis, proceeds by induction on the typing derivation $\Gamma, \Delta \vdash_q q : \nu$. In each case, we can fill in the details about q and v from the derivation. If required, we can then match on the semantic relation $q \xrightarrow{l} q'$ given what we have learnt about the syntactic form of q.

- Case T-Atomic:

  Impossible, no evaluation rules with **atomic** on the left-hand side.

- Case T-RelApp:

  By appealing to the inversion principle we can pick apart the derivation by instantiating q to an equation of the form $t \diamond$ and ν to an equation type $Eq\ e$. We also now know the type of t to be $SR\ e$. Herein, we write an instantiation of y with x as $y := x$.

  By applying these instantiations to the semantic derivation $q \xrightarrow{l} q'$ we end up with the more specific derivation $t \diamond \xrightarrow{l} q'$. There are then two applicable evaluation rules (i.e. rules where an equation of the form $t\ \diamond$ appears on the left-hand side): S-RApp and S-RAppAbs. In each case, we may make further instantiations as we learn more about the syntactic structure of q, q', and l.

For S-RApp, we learn that $l\ :=\ \mathit{0}$ and $q'\ :=\ t'\ \diamond$. We also learn $t\ \longrightarrow\ t'$ from the premise of S-RApp, and by applying the mutual induction hypothesis (theorem 1) to this, we have a proof of $\Gamma, \Delta\ \vdash\ t'\ :\ \mathit{SR}\ e'$ and $\Delta\ \vdash\ \mathit{SR}\ e\ \simeq_\tau\ \mathit{SR}\ e'$. Thus, all that remains to complete this subcase is to show that there exists a μ such that $\Gamma, \Delta\ \vdash_q\ t'\ \diamond\ : \mu$, and that $\Delta\ \vdash\ \mathit{Eq}\ e\ \simeq_\nu\ \mu - \mathit{0}$, which follows immediately if we take $\mu\ :=\ \mathit{Eq}\ e'$.

For S-RAppAbs, given $t\ :=\ \textbf{sigrel}\ i\ l\ q'$, the proof obligation is simply $e - l\ \simeq_e\ e - l$, which is trivially true by reflexivity.

- Case T-Pair:

  Given $\nu\ =\ \mathit{Eq}\ (e_1 + e_2)$ and $q\ =\ q_1 \wedge q_2$, by inversion we learn that $q_1\ :\ \mathit{Eq}\ e_1$ and $q_2\ :\ \mathit{Eq}\ e_2$. There are two applicable rules.

  For S-Pair1, there exists a q_1' such that $\Gamma, \Delta\ \vdash_q\ q_1'\ :\ \mathit{Eq}\ e'$ and $q_1\ \xrightarrow{l}\ q_1'$. We must show $\mathit{Eq}\ (e_1 + e_2)\ \simeq_\nu\ \mathit{Eq}\ ((e' + e_2) - l)$. Using the induction hypothesis, which tells us that $\mathit{Eq}\ e_1\ \simeq_\nu\ \mathit{Eq}\ (e' - l)$, and some equational reasoning on expressions, the proof obligation can be fulfilled.

  For S-Pair2, this case follows the same pattern as above with the induction hypothesis invoked on the second component of the pair.

□

Theorem 1. (Functional Subject Reduction) *Given two terms t and t', if $t\ \longrightarrow\ t'$ and $\Gamma, \Delta\ \vdash\ t\ :\ \tau$ then $\Gamma, \Delta\ \vdash\ t'\ :\ \sigma$ where $\Delta\ \vdash\ \tau\ \simeq_\tau\ \sigma$.*

Proof. Once again, the proof proceeds by induction on the typing derivation. For a number of the cases (e.g. the trivial T-Var case) the proofs are unchanged from that of the simply-typed λ-calculus. Hence, we omit these details here and instead refer the reader to external resources on the topic (see Pierce [2002]).

- Case T-SigRel:

  Given $t = \textbf{sigrel}\ l\ q$ and $\Gamma, \Delta \vdash t : \mathit{SR}\ e$ the only applicable reduction rule is S-SigRel. From S-SigRel we know there exists a q' such that $q \xrightarrow{l'} q'$ and $q' : \mathit{Eq}\ e'$. The goal is then to prove $\mathit{SR}\ (e - l) \simeq_\tau \mathit{SR}\ (e' - (l + l'))$, which follows from the application of lemma 2 to $q \xrightarrow{l'} q'$ and equational reasoning.

- Case T-TApp:

  For this derivation we have $t = t_1\ [e]$ and $\tau = [b \mapsto e]\ \tau_1$. Mirroring term-level abstraction and application, there are two reduction rules that can be applied to this initial term. For S-TApp one learns from the induction hypothesis that given $t' = t_2\ [e]$ then $\Gamma, \Delta \vdash t_2 : \forall\ b' . \sigma$ and also $\forall\ b . \tau_1 \simeq_\tau \forall\ b' . \tau_2$. It is then straightforward to construct a proof of $[b \mapsto e]\ \tau_1 \simeq_\tau [b' \mapsto e]\ \tau_2$.

  The other applicable rule is S-TAppAbs. There are no induction hypotheses to invoke for this rule. However, we can apply the aforementioned substitution lemma, resulting in a trivial proof obligation.

□

Lemma 3. (Equational Progress) *Given a closed, well-typed equation* $\epsilon, \epsilon \vdash_q q$*, either there exists* q' *and* l *such that* $q \xrightarrow{l} q'$ *or else* q *is a value.*

Proof. The proof proceeds by induction on a typing derivation of q (recall that q is required to be well-typed, hence $\epsilon, \epsilon \vdash_q : \nu$). Thus, there are three cases that need our attention.

Case T-Atomic:

Immediate: $q := \textbf{atomic}$, which is a value.

Case T-RelApp:

Given $q := t \diamond$ and $\nu := Eq\ e$, we also know by inversion $t : SR\ e$. Invoking theorem 2 on t, we learn that either t is a value, or else there exists some t' such that $t \longrightarrow t'$. If t is indeed a value, then by canonicity $t := \textbf{sigrel}\ i\ l'\ \textbf{where}\ qv$, and as a result S-RAppAbs applies with $q' := qv$ and $l := l'$. Alternatively, if t can make progress, then S-RApp applies with $q' := t' \diamond$ and $l := 0$.

Case T-Pair:

For T-Pair we have $q := q_1 \wedge q_2$, $\nu := Eq\ (e_1 + e_2)$, and by inversion $\epsilon, \epsilon \vdash_q q_1 : Eq\ e_1$ and $\epsilon, \epsilon \vdash_q q_2 : Eq\ e_2$. Applying the induction hypothesis to the first component of the pair reveals that either q_1 is a value, or else there exists q_1' and l_1 such that $q_1 \xrightarrow{l_1} q_1'$. In the case that q_1 can make progress then the rule S-Pair2 applies. If q_1 is a value then we must appeal to the induction hypothesis on q_2. For the case that both q_1 and q_2 are values then the pairing of the two equations is also a value. Otherwise, the rule S-Pair1 applies.

□

Theorem 2. (Functional Progress) *Given a close, well-typed term $\epsilon, \epsilon \vdash t$, either there exists t' such that $t \longrightarrow t'$ or else t is a value.*

Proof. By taking the usual approach and using induction on the typing derivation of t we find there is only one case of genuine interest.

Case T-Sigrel:

Taking $t := \textbf{sigrel}\ i\ l\ \textbf{where}\ q$ and $\tau := SR\ e$, inversion of T-Sigrel also informs us that $\epsilon, \epsilon \vdash_q q : Eq\ e$. By applying lemma 3 to q we can decide if q is a value, in which case the expression **sigrel** $i\ l$ **where** q is a value as a whole, or if q can make progress, in which case the rule S-Sigrel applies with $t' := \textbf{sigrel}\ i\ (l + l')\ \textbf{where}\ q'$.

□

4.3 A Constraint-based Structural Type System

4.3.1 Key Ideas

So far we have only addressed the property of equation-variable balance. Thus, it is natural to wonder to what extent we can capture the stronger property of structural non-singularity using types. Nilsson [2008] is the earliest known effort in investigating such properties to their full extent by annotating types with equation-variable incidence matrices. The incidence matrices keep track of exactly which variables occur in each equation. Thus, an $n \times m$ matrix tracks the occurrences of n variables over a set of m equations.

However, while initially compelling, attempting to capture such a strong property proved to be quite difficult. In particular, Nilsson does not consider first-class components, but instead requires that all components are concrete, and thus that all incidence matrices are known during type checking. This assumption also suggests that the matrices may be breaking encapsulation, leaking precise structural information to the wider system. Moreover, complex algorithms — both from a human, and computational complexity standpoint — are required for disambiguation purposes, particularly when considering more advanced modelling techniques such as structural dynamism. However, Nilsson's work is significant, and this preliminary work provided much of the inspiration and motivation for the systems developed in this chapter.

The solution in this chapter is to find a middle ground between balance checking and singularity detection. We compromise on type system strength by falling short of full singularity detection (as attempted by Nilsson), but instead prioritise the handling of first-class components and structural dynamism.

The crux of the type system is to enrich the simple approach with sets of *balance constraints* (hereafter simply *constraints*). These constraints restrict the interval of the balance variables occurring in types. Constraints may involve the contributions of several components, and are thus not directly associated with a single signal relation in general. Hence, we adopt a syntax similar to Haskell's type class constraints to express that a set of constraints restricts the intervals of the variables occurring in a type. To illustrate, consider the following type:

$$\mathit{foo} : \forall\, n\ m\,.\,(n \leqslant m, n \geqslant 2) \Rightarrow \mathit{SR}\ n \rightarrow \mathit{SR}\ m$$

Whilst contrived, the above type quantifies over and relates two balance variables, demonstrating how the contributions of the two components are restricted. The meaning of such a type can then be thought of in terms of the set of valid instantiations. A valid instantiation is any valuation of balance variables such that the constraints remain consistent. For example, the following are all valid instantiations for the type of *foo*:

$$\begin{array}{llll} \mathit{bar} : & (3 \leqslant 5, 3 \geqslant 2) & \Rightarrow \mathit{SR}\ 3 & \rightarrow \mathit{SR}\ 5 \\ \mathit{baz} : \forall\, n\,. & (n \leqslant 4, n \geqslant 2) & \Rightarrow \mathit{SR}\ n & \rightarrow \mathit{SR}\ 4 \\ \mathit{qux} : \forall\, o\ m\,. & (n \leqslant (o + 1), n \geqslant 2) & \Rightarrow \mathit{SR}\ n & \rightarrow \mathit{SR}\ (o + 1) \end{array}$$

Should the set of instantiations be empty (i.e. should the constraints be inconsistent) then the type is rejected as ill-formed. It is through this mechanism that we determine if the structure of an equation or its composition is acceptable.

4.3.2 Structural Criteria

With the knowledge that detection of structural singularities is at the very least infeasible and likely also intractable, we turn our attention to devising structural criteria that are both easily expressed as constraints over balance variables, and also eclipse the strength of simple balance checking alone. A number of criteria are introduced in the following section, stemming from the setting of FHM, from which such type constraints can be generated.

It is possible that what constitutes a useful constraint may vary across application domains. We will take care when developing the constraint-based type system to ensure that the constraints chosen are not tied to any specific domain. The only restriction we place on the constraints is that they are linear inequalities. However, even this predicate is only a weak requirement stemming from our choice of constraint solver. Indeed, one could potentially even parameterise our system on the language of constraints and a means with which to solve them, for example, as is the case with Dependent ML (Xi [2007]).

Structurally Well-Formed Signal Relations

In order to formulate structural criteria for well-formedness of signal relations, let us first define a number of terms and quantities pertaining to the different *kinds* of variables and equations. Given a signal relation, the number of interface variables (Sect. 2.3) is denoted by i_Z. The number of local variables, denoted l_Z, is then just the number of variables occurring in the equations minus the number of interface variables. The set of equations in a signal relation can be partitioned into disjoint subsets of interface, local, and mixed equations:

- *interface equation*: only interface variables occur.
- *local equation*: only local variables occur.
- *mixed equation*: both interface and local variable occur.

It is worth noting that the classification of equations into interface, local, and mixed is a coarse approximation of an incidence matrix. In the simple balance type system we disregarded occurrences by effectively assuming that every variable occurred in every equation, thereby defining the coarsest possible approximation of an incidence matrix. By distinguishing between different kinds of equations our approach is a middle ground between simple balance and incidence matrices; we do not retain a completely faithful view of variable occurrences, but we do not assume that, for example, a local variable occurs in an interface equation, as is the case in the simple balance type system. However, what we do assume is that, for example, *all* local variables occur in each local equation (i.e. any local equation can be used to solve for any local variable).

The number of interface, local, and mixed equations is denoted i_Q, l_Q, and m_Q respectively. The total number of equations $a_Q = i_Q + l_Q + m_Q$. A relation is *structurally well-formed* if the following criteria are satisfied:

1. $l_Q + m_Q \geqslant l_Z$: The local variables are not underconstrained.

2. $l_Q \leqslant l_Z$: The local variables are not overconstrained.

3. $i_Q \leqslant i_Z$: The interface variables are not overconstrained.

4. $a_Q - l_Z \leqslant i_Z$: A signal relation must not contribute more equations than there are interface variables (no over-contribution).

5. $l_Q \geqslant 0, m_Q \geqslant 0, \mathit{and}\ i_Q \geqslant 0$: When considering structurally dynamic systems we will permit negative contributions at intermediate stages (e.g. when checking the branches of a switch), but insist that at the top level the contribution of each equation kind must be non-negative.

To illustrate, let us return to the *resistor* example from Sect. 2.3. We have $i_Z = 4$ (recall that each *Pin* contains two variables), $l_Z = 1$, $i_Q = 0$, $l_Q = 0$, $m_Q = 3$ (the application of *twoPin* contributes 2 mixed equations), and thus $a_Q = 3$. The following 7 constraints are generated from the 5 criteria: (1) $0 + 3 \geqslant 1$, (2) $0 \leqslant 1$, (3) $0 \leqslant 4$, (4) $3 - 1 \leqslant 4$, and (5) $0 \geqslant 0$, $3 \geqslant 0$, $0 \geqslant 0$. All constraint criteria are satisfied. Hence, *resistor* is structurally well-formed according to the above criteria.

The question remains as to how the above criteria relate to the two criteria discussed in the previous section. The criteria here are stronger than insisting on balance, as a modular form of variable counting can be derived using criteria (4) and (5) alone. However, the constraints are weaker than insisting on a bijection between equations and variables: the constraints would need to consider the incidence matrices of equations and variables to determine if a bijection exists, as investigated by Nilsson [2008]. However, by taking some account of which variables occur in which equations through the partitioning into interface, local, and mixed equations, we have achieved a better approximation for checking for structural non-singularity than basic balance checking, while retaining a modular formulation that, as we will see in the next section, can be extended to account for structural dynamism.

Structurally Sound Dynamism

Recall that a structurally dynamic system of equations is one where the equations may vary over time. As FHM permits structurally dynamic systems, we need to consider how to generalise the notion of structural well-formedness to work in a structurally dynamic setting. The nature of structural dynamism in FHM means that a very large, possibly even unbounded, number of system configurations are possible. Thus, we cannot hope to enumerate the configurations and check each one. Rather, we need to reconcile the structural properties of the branches of the switch blocks (the variable parts of an FHM system) — without losing too much information — into structural properties that hold at all times for each switch block as a whole, and then use this reconciled information to determine the well-formedness of the entire system.

As a simple example of unbounded dynamism consider a bouncing ball that experiences elastic collision with the floor (see Fig. 4.8). The behaviour that describes the ball falling to the ground can be modelled as a single component. As the ball touches the ground a discrete event is triggered that causes the vertical velocity of the ball to be inverted and a transition occurs from the first continuous mode of operation (m_1) to the next (m_2). At this point we can once again model the ball as a freefalling mass. As we are modelling an elastic collision, the ball will bounce indefinitely, and therefore an infinite number of structural configurations will potentially be generated.

In Hydra, we can model the bouncing ball as given below. For convenience, we use the record accessors $p.x$ and $p.y$ to refer to the x and y position of the body, and the accessors $v.x$ and $v.y$ to refer to their rate of change.

$$\begin{aligned}
&\textbf{type}\ \mathit{Pos} &&= (\mathbb{R}, \mathbb{R}) \\
&\textbf{type}\ \mathit{Vel} &&= (\mathbb{R}, \mathbb{R})
\end{aligned}$$

$$\begin{aligned}
&\mathit{freeFall} \ :\ \mathit{SR}\ (\mathit{Pos}, \mathit{Vel}) \\
&\mathit{freeFall} \ =\ \textbf{sigrel}\ (p, v)\ \textbf{where} \\
&\quad \textbf{der}\ p \quad = \ v
\end{aligned}$$

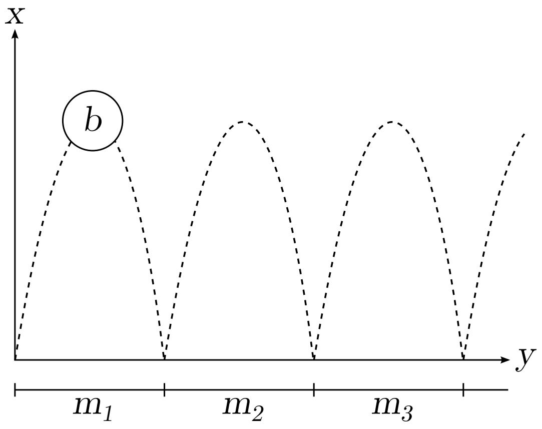

Figure 4.8: An elastic bouncing ball.

```
  der v.x = 0
  der v.y = −g
bouncingBall : SR (Pos, Vel)
bouncingBall = sigrel (p, v) where
  initially ⇒
    freeFall ◇ (p. v)
  when y ⩽ 0 ⇒
    bouncingBall ◇ (p, (v.x, −v.y))
```

The above example serves as a reminder of the type of structural changes we wish to accommodate. We must now consider how the structural constraints on a dynamic model can be formalised such that they are an accurate description of a models structure, regardless of the current mode of operation.

There are a number of ways to compare the structure of different switch branches. One approach might be to insist that each branch have an identical structure: every branch consists of the same number of each kind of equation. Let us call this the *strong* approach for the purpose of this discussion. However, this approach is very restrictive. To understand why, consider a switch with two branches: the first branch consists of an interface equation and a local equation, the second branch consists of two mixed equations. These branches clearly have a very different structure, but are arguably interchangeable: both branches can be used to solve for one interface variable and one local variable.

An obvious alternative is to discard the equation kind information altogether and require only that each branch of a switch block contribute the same number of equations. Let us call this the *weak* approach. Clearly, the previous example now checks under this scheme as both branches contribute 2 equations. However, this approach is arguably too permissive: there are equation systems that contribute the same number of equations but are not structurally compatible. Indeed, this was the very reason to introduce equation kinds in the first place.

Instead, we adopt reconciliation constraints that enforce a stronger notion of structural compatibility than simple equation-variable balance, without requiring the branches of a switch block to be structurally identical. We refer to this as the *fair* approach. The constraints are defined over an n-branch switch block, containing n sets of equations $q_1 \dots q_n$, where q_k consists of l_k local equations, m_k mixed equations, and i_k interface equations. The variables l, m, and i are fresh variables denoting the local, mixed, and interface contribution of the reconciled block as a whole. The constraints are parametrised on k, and the reconciliation constraints for a switch block are obtained by instantiating them for each branch (i.e. for each k in *1* ... n):

6. $l \geqslant l_k \geqslant 0$: The reconciled system contributes at least as many local equations as the systems being reconciled. There cannot be a negative contribution of local equations.

7. $i \geqslant i_k \geqslant 0$: The reconciled system contributes at least as many interface equations as the systems being reconciled. There cannot be a negative contribution of interface equations.

8. $m \leqslant m_k - (l - l_k) - (i - i_k)$: The reconciled system may use mixed equations (from inside or outside the switch block) to compensate for any deficit in the required number of interface or local equations. This may result in m being negative, requiring the enclosing context of the switch

block to contribute additional mixed equations.

9. $l + m + i = l_k + m_k + i_k$: The reconciled system contributes the same number of equations as each branch. Thus, each branch must have the same contribution.

The driving intuition is that we must find and associate some specific, *time-invariant* number of local variables and interface variables with each switch block such that the block, regardless of which branch is active, can provide that many equations to solve for the interface and local variables, respectively. We can then rely on the block to *always* contribute equations to that end, meaning we effectively can view the block as a static equation system fragment with that specific contribution. Of course, reconciling a block need not find a unique contribution and, in-line with the rest of the system, may merely restrict the contribution to an interval, which will later be resolved to a concrete value when used in a complete model. Note that l and i must be at least as high as the maximal number of local equations and interface equations, respectively, over all branches. Otherwise some branches will contribute more local or interface equations than can be used. A subtlety is that the number of *mixed* equations contributed by a switch block is allowed to be negative. This just means that the switch block may need to “borrow” some mixed equations from the enclosing context in order to make up for a deficit of the number of local or interface equations in some branches.

To demonstrate, consider the following contrived example $dynamism_1$:

```
$dynamism_1 : SR\ (\mathbb{R}, \mathbb{R}) \rightarrow SR\ \mathbb{R}$
$dynamism_1\ sr =$ sigrel $x$ where
  local $u$
  initially
    $f\ u = 0$
    $g\ x = 0$
  when $u < 0 \Rightarrow$
    $sr \diamond (x, u)$
```

The relation contains a switch block with two branches: the **initially** branch consists of 1 local equation and 1 interface equation, while the **when** branch consists of n mixed equations, where n is the contribution of the relation sr. The switch block would be rejected under the strong approach, as the structure of the two branches is not identical.

However, under the fair approach, the block is reconcilable. Applying the rules to each branch results in 8 constraints that must be satisfied: $l \geqslant 1 \geqslant 0$, $l \geqslant 0 \geqslant 0$, $i \geqslant 0 \geqslant 0$, $i \geqslant 1 \geqslant 0$, $m \leqslant 0-(l-1)-(i-1)$, $m \leqslant n-(l-0)-(i-0)$, $l+m+i=2$, and $l+m+i=n$. Through simplification, we can verify that they are satisfiable with $l=1$, $m=0$, $i=1$, and $n=2$.

For another example, consider *dynamism*$_2$ below. The switch block provides an interface equation in one branch and a local equation in the other. These branches are thus not immediately reconcilable. However, by considering the mixed equation in the enclosing context, it is possible for the entire relation to be balanced, regardless of which branch is active:

```
dynamism₂ : SR ℝ
dynamism₂ = sigrel x where
  local u
  h x u = 0
  initially
    f x = 0
  when x > 0 ⇒
    g u = 0
```

Applying the fair approach results in the following constraints: $l \geqslant 0 \geqslant 0$, $l \geqslant 1 \geqslant 0$, $i \geqslant 1 \geqslant 0$, $i \geqslant 0 \geqslant 0$, $m \leqslant 0-(l-0)-(i-1)$, $m \leqslant 0-(l-1)-(i-0)$, $l+m+i=1$, $l+m+i=1$. Simplification yields a solution at $l=1$, $i=1$, $m=-1$. Thus, the switch block contributes (or in this case *requires*) -1 mixed equations. The above switch block can be interpreted as being reconcilable provided that it appears in a context containing at least 1 mixed equation.

Finally, consider the example *dynamism*$_3$ where the weak approach is too permissive, but, by contrast, the fair approach correctly rules out the switch block as irreconcilable:

$$
\begin{array}{lcl}
\mathit{dynamism}_3 & : & \mathit{SR}\ \mathbb{R} \\
\mathit{dynamism}_3 & = & \textbf{sigrel}\ x\ \textbf{where} \\
\quad \textbf{local}\ u\ v & & \\
\quad \textbf{initially} & & \\
\qquad u & = & v \\
\qquad f\ u\ v & = & 0 \\
\quad \textbf{when}\ u + v < 0 \Rightarrow & & \\
\qquad g\ x & = & 0 \\
\qquad x & = & u
\end{array}
$$

The **initially** branch consists of 2 local equations, whereas the **when** branch consists of 1 interface equation and 1 mixed equation. Clearly, with only a single mixed equation, it should not be possible to account for the 2 local equations demanded by the reconciled relation. Indeed, running the criteria over the above relation results in the constraints $l \geqslant 2$, $i \geqslant 1$, and $l + m + i = 2$, implying that $m \leqslant -1$. However, there are no additional mixed equations in the enclosing context, and criterion 5 insists that m must be non-negative when checking the body of a signal relation. Hence, *dynamism*$_3$ is rightly rejected.

4.3.3 $H_{\models}$: A Core Language for Structural Types

We return again to the topic of formalisation. However, this time our motivations are slightly different. Our formalisation of the balance type system was born out of the desire to show that certain metatheoretical properties hold. Conversely, the driving motivation now is to provide a reference implementation that accurately describes the constraint-based type system and to provide a specification that can easily be implemented and adapted for existing equation-based languages. Furthermore, rather than strive for metatheoretical properties such as safety, which as it turns out, *does not hold* for this system, we instead aim to present a system that is amenable to other desirable properties, such as total, annotation-free type reconstruction. The lack of safety, and justification for its absence, are discussed later in the section.

Types and Schemes

The main difference between the H_Δ and $H_\models$ categories of types (see Fig. 4.9) — and the first suggestion that we are heading in a more pragmatic direction with the constraint-based type system — is the segregation of types into monotypes (*types*) and polytypes (*schemes*). Consequently, from a theoretical standpoint, the language of types has been greatly restricted. However, from a practical standpoint, the changes make sense: we are trading off theoretical strength for type reconstruction. Of course, higher-rank types have found many applications in other languages (e.g. Schrijvers et al. [2008]), but we are more interested in creating a type system with immediate practical value. Indeed, type reconstruction is more than a mere convenience: it allows us to hide the language of constraints entirely from the user. They need only be revealed when an equation is found to be structurally unsound. Moreover, insisting that the user manually annotate all balance variables and constraints for every abstraction would be both time-consuming and error prone, as illustrated by the earlier examples.

Supplying equation types with three parameters instead of one is another notable difference. Rather than representing the overall balance of an equation the parameters now represent the number of local, mixed, and interface equations that an equation is capable of contributing. Once again, the strategy will be to compute these quantities in a bottom-up fashion.

Terms

The shape of the terms of the constrained language (see Fig. 4.10) is fundamentally the same as those for the simple balance language. Equations in $H_\models$ differ only in that they encode more information about the concrete equations they represent (as opposed to the abstraction used in the core language). In particular, they record the *kind* of an equation or application - information that

$\sigma ::=$	type scheme:
$\forall\, b\,.\,\sigma$	balance abstraction
$c \Rightarrow \tau$	constrained type
$\tau ::=$	type:
$\tau_1 \;\rightarrow\; \tau_2$	function space
$\mathit{SR}\ e$	signal relation
$\mu ::=$	equation:
$c \Rightarrow \nu$	constrained equation
$\nu ::=$	simple equation:
$\mathit{Eq}\ e_1\ e_2\ e_3$	equation
$c ::=$	constraints:
ϵ	empty
$e_1 \leqslant e_2$	constraint
c_1, c_2	constraint conjunction
$e ::=$	balance:
$\mathbb{Z}$	integer
b	variable
$e_1 + e_2$	addition
$-\ e$	negation

Figure 4.9: $H_{\models}$ types.

$t ::=$	term:
x	variable
$t_1\ t_2$	application
$\lambda\, x\,.\, t$	abstraction
$\textbf{let}\ x\ =\ t_1\ \textbf{in}\ t_2$	let binding
$\textbf{sigrel}\ i\ l\ \textbf{where}\ q$	signal relation
$q ::=$	equation:
$\textbf{atomic}\ k$	atomic
$q_1 \wedge q_2$	pairing
$t \diamond k$	application
sw	switch block
$sw ::=$	switch:
$\textbf{initially}\ q$	initial branch
$sv\ \textbf{when}\ q$	event branch
$k ::=$	equation kind:
local	local equation
mixed	mixed equation
interface	interface equation
$i, l ::=$	variable accumulator:
$\mathbb{Z}$	integer

Figure 4.10: $H_{\models}$ terms.

is easy to determine statically, prior to type checking - as this information is necessary to capture the constraint criteria presented in Sect. 4.3.2.

Additionally, we drop the explicit notion of balance variables as we plan to move over to an implicit Hindley-Milner style (Milner [1978]) setting instead.

Values

Values in the constraint-based system also remain very similar to their simple balance system counterparts. Once again, where relevant, equations have been annotated with a *kind*. Explicit balance quantification has also been removed.

A Comparison to Hydra

The syntax of $H_{\models}$ is very similar to that of H_{Δ}. Nevertheless, to reinforce the right intuitions about the meaning of the constructs, it may be helpful to

v ::=	value:
$\lambda\, x \,.\, t$	abstraction
sigrel i l **where** qv	signal relation
qv ::=	equation value:
atomic k	atomic equation
$qv_1 \wedge qv_2$	pairing
sv	switch block
sv ::=	switch value:
initially qv	initial branch
sv **when** q	event branch

Figure 4.11: $H_{\models}$ values.

revisit the par (see Fig. 4.12) example so as to relate the syntax of $H_{\models}$ directly to Hydra. The most important difference is the inclusion of equation kinds, providing more information about the nature of the equations appearing in the body of *par*. Also of note is the absence of balance quantification and type annotations appearing on λ-abstractions. These annotations are left implicit as the type checking algorithm will be capable of inferring this information.

4.3.4 A $H_{\models}$ Type System

Despite the differences between $H_{\triangle}$ and $H_{\models}$, the approaches to formalising both type systems are conceptually the same: a relation is constructed for each major syntactic grammar of the language with one axiom per production. For a term t, $\Gamma \;\vdash\; t \;:\; \sigma$ states that t has the type scheme σ in the context Γ. Similarly, $\Gamma \;\vdash_q\; q \;:\; \mu$ states that q has the constrained equation type μ in the context Γ.

The typing rules for $H_{\models}$ can be seen in Fig. 4.13. For a number of reasons we have chosen to give a formulation with an implicit notion of balance variable quantification, making use of a Hindley-Milner style presentation using generalisation and instantiation rules (see Milner [1978]). One alternative approach is to make the unification constraints explicit in the context of a rule, as seen in the presentation of type reconstruction in Pierce [2002]. However, we believe that this detracts from the main focus of the rules: to capture *balance constraints* in

$\mathit{par}\ \mathit{sr}\ =\ \textbf{sigrel}\ (p, n)\ \textbf{where}$	$\mathit{par}\ =\ \lambda\ \mathit{sr}\,.$
$\quad\textbf{local}\ p_1\ p_2\ n_1\ n_2$	$\quad\textbf{sigrel}\ \mathit{4}\ \mathit{8}\ \textbf{where}$
$\quad\mathit{sr}\ \diamond\ (p_1, n_1)$	$\qquad\mathit{sr}\ \diamond\ \textbf{local}\ \wedge$
$\quad\mathit{sr}\ \diamond\ (p_2, n_2)$	$\qquad\mathit{sr}\ \diamond\ \textbf{local}\ \wedge$
$\quad p.i + p_1.i + p_2.i\ =\ \mathit{0}$	$\qquad\textbf{atomic mixed}\ \wedge$
$\quad n.i + n_1.i + n_2.i\ =\ \mathit{0}$	$\qquad\textbf{atomic mixed}\ \wedge$
$\quad p.v\ \ =\ p_1.v$	$\qquad\textbf{atomic mixed}\ \wedge$
$\quad p_1.v\ =\ p_2.v$	$\qquad\textbf{atomic local}\ \wedge$
$\quad n.v\ \ =\ n_1.v$	$\qquad\textbf{atomic mixed}\ \wedge$
$\quad n_1.v\ =\ n_2.v$	$\qquad\textbf{atomic local}$

Figure 4.12: Comparison of Hydra and $H_{\models}$.

a language of equations. Indeed, we make no contributions to type inference at all as our algorithm uses completely standard techniques for inferring the most general type of an equation system. Thus, we do not present the details of this algorithm here, but refer the interested reader to the accompanying code.

The functional rules are mostly straightforward with only the added burden of accumulating the generated balance constraints. For example, in the rule T-App, we need to combine the constraints generated from both of the subterms. Of course, combining two consistent sets of constraints does not necessarily yield a consistent set of constraints. Thus, we add the implicit assumption that a type is only well-formed if all constraint sets occurring in the type can be satisfied. To determine if a set of constraints is consistent we use the *Fourier-Motzkin Quantifier Elimination* (or *FMQE*) algorithm (see Pugh [1991]). The advantage of this algorithm is that it can find a continuous interval for each balance variable, which is particularly useful for type equality (i.e. two types are equal only if they agree on the interval of each balance available).

Fourier-Motzkin elimination has worst case exponential time complexity in the number of balance variables. However, as shown by Pugh (see Pugh [1991]), the modified variant that searches for integer solutions is capable of solving most common problem sets in low-order polynomial time. Furthermore, systems typically involve only a handful of balance variables, making it feasible to check most cases where complexity is exponential in the number of variables.

Turning our attention to the rules themselves, the reader will notice that the generation of constraints is passed off to a pair of helper functions *con* and con_{sw}. These functions simply generate constraints that correspond to the constraint criteria presented in Sect. 4.3.2. Note that we allow ourselves to use the shorthand $e_1 \leqslant e_2 \leqslant e_3$, which is expanded to $e_1 \leqslant e_2, e_2 \leqslant e_3$, and the shorthand $e_1 = e_2$, which is expanded to $e_1 \leqslant e_2, e_2 \leqslant e_1$.

$$\begin{array}{ll}
con\ (Eq\ i_Q\ m_Q\ l_Q, i_Z, l_Z)\ = & con_{sw}\ (Eq\ l\ m\ i, Eq\ l_k\ m_k\ i_k)\ = \\
\quad i_Q + m_Q + l_Q - l_Z \leqslant i_Z, & \quad 0 \leqslant l_k \leqslant l, \\
\quad i_Q \leqslant i_Z, & \quad 0 \leqslant i_k \leqslant i, \\
\quad l_Q \leqslant l_Z \leqslant l_Q + m_Q, & \quad m \leqslant m_k - (l - l_k) - (i - i_k), \\
\quad 0 \leqslant i_Q, 0 \leqslant m_Q, 0 \leqslant l_Q & \quad l + m + i = l_k + m_k + i_k
\end{array}$$

Forming equation types is slightly more involved in the presence of constraints. Fortunately, we can use two more functions to simplify the presentation. Under the simple balance approach the rules T-Atomic and T-RelApp contributed *1* and *e* equations, respectively. Now that we are differentiating between different equation kinds, we must construct an appropriate equation type that reflects this kind:

$$\begin{array}{lcl}
kind\ (\mathbf{local}, \quad e) & = & Eq\ 0\ 0\ e \\
kind\ (\mathbf{mixed}, \quad e) & = & Eq\ 0\ e\ 0 \\
kind\ (\mathbf{interface}, e) & = & Eq\ e\ 0\ 0
\end{array}$$

The $\oplus$ operator performs a point-wise sum of contributions:

$$(Eq\ c_1\ c_2\ c_3)\ \oplus\ (Eq\ d_1\ d_2\ d_3)\ =\ Eq\ (c_1 + d_1)\ (c_2 + d_2)\ (c_3 + d_3)$$

The symbols *fresh* and *free* denote functions that create a fresh balance variable, and compute the set of free variables occurring in types, respectively.

These functions are left abstract, leaving them as implementation details of a specific algorithm used to implement this type system.

Finally, we define a $\sqsubseteq$ predicate with the rule given below. The rule ensures that no free variables occuring in the monotype become bound by a quantifier, but existing quantifiers may be replaced by new types, including types that introduce new balance variables:

$$\frac{\tau_2 \;=\; [\alpha_i \;\mapsto\; \tau_i]\,\tau_1 \qquad \textit{fresh}\;(\beta_i)}{\forall\,\alpha_1\,...\,\forall\,\alpha_n\,.\,\tau_1 \;\sqsubseteq\; \forall\,\beta_1\,...\,\forall\,\beta_m\,.\,\tau_2}$$

4.3.5 Metatheoretical Properties

The typical notion of type system safety, as explored for H_Δ (see 4.2.5), emerges from the conventional definition of what a type system should be. Pierce defines a type system as follows (see Pierce [2002]):

> "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute."

In particular, Milner [1978] introduced the well-known mantra "well-typed programs do not go wrong", which was subsequently developed into a syntactic techinque by Wright and Felleisen [1994]. This line of thinking has given rise to the idea that a type system should guarantee the absence of a particular class of errors from well-typed programs. Conversely, an ill-typed program should be ruled out as it *may* exhibit undesirable behaviour. Of course, what may be considered as undesirable behaviour is dependent upon the specific type system in question. Certainly, it can be very difficult indeed — and in many cases impossible, for example, if non-termination is considered undesirable behaviour — to design a type system that captures *all* undesirable behaviour. Equivalently, this viewpoint implies that a well-typed program is *definitely not flawed*, for an appropriate value of *flawed*.

$$\boxed{\Gamma \vdash t : \sigma}$$

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \ (\text{T-Var}) \qquad \frac{\begin{array}{c}\Gamma \vdash t_2 : c_2 \Rightarrow \tau_2 \\ \Gamma \vdash t_1 : c_1 \Rightarrow \tau_2 \to \tau_1\end{array}}{\Gamma \vdash t_1\, t_2 : c_1, c_2 \Rightarrow \tau_1} \ (\text{T-App})$$

$$\frac{\Gamma \vdash_q q : c_2 \Rightarrow \mathit{Eq}\; e_1\; e_2\; e_3 \qquad c_1 = \mathit{con}\,(e_1, e_2, e_3, i, l)}{\Gamma \vdash \textbf{sigrel}\; i\; l\; \textbf{where}\; q : c_1, c_2 \Rightarrow \mathit{SR}\,(e_1 + e_2 + e_3 - l)} \ (\text{T-SigRel})$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : c_1 \Rightarrow \tau_1 \\ \Gamma \triangleright x : c_1 \Rightarrow \tau_1 \vdash t_2 : c_2 \Rightarrow \tau_2\end{array}}{\Gamma \vdash \textbf{let}\; x = t_1\; \textbf{in}\; t_2 : c_2 \Rightarrow \tau_2} \ (\text{T-Let})$$

$$\frac{\Gamma \triangleright x : \epsilon \Rightarrow \tau_1 \vdash t : c \Rightarrow \tau_2}{\Gamma \vdash \lambda\, x\,.\, t : c \Rightarrow \tau_1 \to \tau_2} \ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t : \sigma \qquad n \notin \mathit{free}\,(\Gamma)}{\Gamma \vdash t : \forall\, n\,.\,\sigma} \ (\text{T-Gen})$$

$$\frac{\Gamma \vdash x : \sigma_1 \qquad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash x : \sigma_2} \ (\text{T-Inst})$$

$$\boxed{\Gamma \vdash_q q : \mu}$$

$$\frac{\Gamma \vdash t : c \Rightarrow \mathit{SR}\; e}{\Gamma \vdash_q t \diamond k : c \Rightarrow \mathit{kind}\,(k, e)} \ (\text{T-RelApp})$$

$$\frac{}{\Gamma \vdash_q \textbf{atomic}\; k : \epsilon \Rightarrow \mathit{kind}\,(k, \mathit{1})} \ (\text{T-Atomic})$$

$$\frac{\Gamma \vdash_q q_1 : c_1 \Rightarrow \nu_1 \qquad \Gamma \vdash_q q_2 : c_2 \Rightarrow \nu_2}{\Gamma \vdash_q q_1 \wedge q_2 : c_1, c_2 \Rightarrow \nu_1 \oplus \nu_2} \ (\text{T-Pair})$$

$$\boxed{\Gamma \vdash_{sw} sw : \mu}$$

$$\frac{\begin{array}{c}c_2 = \mathit{con}_{sw}\,(\mathit{Eq}\; l\; m\; i, \nu) \\ \Gamma \vdash_q q : c_1 \Rightarrow \nu \qquad \mathit{fresh}\,(l, m, i)\end{array}}{\Gamma \vdash_{sw} \textbf{initially}\; q : c_1, c_2 \Rightarrow \mathit{Eq}\; l\; m\; i} \ (\text{T-Initial})$$

$$\frac{\begin{array}{c}c_3 = \mathit{con}_{sw}\,(\nu_1, \nu_2) \\ \Gamma \vdash_q q : c_2 \Rightarrow \nu_2 \qquad \Gamma \vdash_{sw} sw : c_1 \Rightarrow \nu_1\end{array}}{\Gamma \vdash_{sw} sw\; \textbf{when}\; q : c_1, c_2, c_3 \Rightarrow \nu_1} \ (\text{T-When})$$

Figure 4.13: $H_{\models}$ typing rules.

However, there is an alternative viewpoint: a well-typed program is *not definitely flawed*. By pulling the negation up to the top level we have exchanged our optimism for pessimism: rather than proving the absence of flaws in a well-typed program we instead prove the presence of flaws in ill-typed programs.

Of course, this means that well-typed programs may go wrong, so why take such a viewpoint? One consequence is that the type system can now search for more flaws without necessarily guaranteeing their absence. Arguably, the error-finding power of the type system is increased, particularly in a refined type system where the system can catch strictly more flaws than before. Such a viewpoint is especially effective for modular programming; errors in a library that will definitely lead to undesirable behaviour can be caught before the library is deployed and used. We are not the first to observe this alternative view of type systems, for example, work on Hybrid Type Checking (Flanagan [2006]) and Gradual Typing (Siek and Taha [2006]) could be classified as not-definitely-flawed type systems.

The notion of subject reduction, which makes sense from the conventional standpoint, is no longer meaningful when we accept that well-typed programs may still go wrong. This discussion is sparked by the fact that our constraint-based system does not fall within the conventional definition of a type system. Equation kinds are only an approximation of structural non-singularity. Hence, as we take a step of reduction, new information may be discovered about the kind of an equation. Consequently, a constraint that was satisfiable before reduction may no longer be so in light of new structural information.

Consider the *foo* example below. Checking the constraints reveals that the two branches contained within *foo* are reconcilable: each branch contributes two equations that must be compatible with a contribution of two local equations (i.e. the second branch is compatible as it contributes two mixed equations). However, if one expands the application of *bar* it is revealed that the second branch is not directly compatible as it contributes one local equation and one interface equation. While these two branches are still reconcilable, the constraints are no longer the same. Specifically, a mixed equation is now needed from the enclosing context to make sense of the relation, but there is no such equation. Thus, the resulting program is ill-typed.

```
bar : SR Double 2
bar = sigrel (x, a, b) where
  x = 0
  a = b

foo : SR Double 0
foo = sigrel x where
  local a b
  f x = 0
  initially
    g a = 0
    h b = 0
  when x > 0 ⇒
    bar ◇ (x, a, b)
```

This problem is not just inherent to structurally dynamic systems, but can also occur in programs with no dynamism whatsoever. Clearly, the usual notion of soundness does not hold for the constraint-based type system.

One can imagine alternative notions of type soundness that give us some assurances that the constraint-based system behaves in a reasonable manner. One such notion is that an ill-typed $H_{\models}$ term implies that there exists some structural configuration (i.e. a particular choice of switch branches) such that the term elaborates to a structurally singular system of equations. Subject expansion is another way to specify soundness, which states that a well-typed term may only come from another well-typed term. That is, for all terms t_1 and t_2 such that $t_1 \rightarrow t_2$, if $\Gamma \vdash t_2 : \sigma$ then $\Gamma \vdash t_1 : \sigma$.

A related notion is whether or not the system has a substitution property. In other words, if we substitute a variable with an expression of the same type from the environment, does the program remain well typed and is this type the same as before? We do not formally establish this theorem in this thesis. However, we speculate that it is most likely true as our substitution machinery is entirely standard, and the lack of type preservation arises from the *reduction* of signal relation applications.

We do not attempt to prove these notions of safety outlined in the above paragraphs, but believe that such proofs are an important avenue of research to pursue in the future, not only for the purposes of our type system, but also for its wide applications to other "non-conventional" type systems.

4.4 Evaluation

We have carried out our development in the context of an abstract version of an FHM-like, acausal modelling and simulation language, leaving out those aspects that were not directly relevant to our specific purposes. We did this partly to keep things simple and allow ourselves to focus on the core issues, and partly, as explained earlier, because the ideas underpinning our type system could be useful for any language with a notion of modular systems of equations.

However, this begs the question of how we might evaluate what we have achieved so far, as up to this point we have not been in a position to carry out any large usability studies. In this section, we attempt to address the question in two ways. First, we position our work relative to other work based on exploiting structural properties of systems of equations for which there is independent evidence of usability. Second, we provide a substantial case study that covers all aspects of the language, including structural dynamism.

4.4.1 Structural Properties in the Wild

Based on years of practical experience, a notion of balance checking was considered to be sufficiently useful to be incorporated into version 3.0 of the Modelica standard (see Mod [2012]) in 2007. See Sect. 6.1.1 for a discussion of how

Modelica compares to the work described in this chapter from the perspective of variable and equation balance. Here we just point out that our system checks more fine-grained structural properties than the Modelica system can as we distinguish between different *kinds* of equations. This means our system is capable of catching a strictly larger set of errors, and thus, we argue, is no less useful than the system presently used in Modelica.

Of course, by strengthening the requirements on programs according to the constraint criteria, we inevitably reject some "good" programs that would be accepted by the simple balance approach. The constraint criteria are necessarily an approximation of the true structure of a flat equation system, hence, it may be possible to solve a flat equation system that would be ill formed in its modular state. Despite this, we still firmly believe that the constraint-based approach has distinct advantages over simple equation-variable balance. At the very least, one can experiment with different constraint criteria to find an appropriate balance between error-finding power and usability. Indeed, if in practice the constraint criteria outlined in Sect. 4.3 turned out to be too restrictive for a particular domain, it is easy in our formalisation to weaken the constraints such that fewer "good" programs would be rejected.

Our type-based approach scales to first-class equation fragments and structurally dynamic systems of equations, features that may be commonplace in the next generation of acausal modelling languages (see [Broman, 2010, Zimmer, 2013]). Moreover, our approach is also modular in the sense that different constraints can be added and removed from the type system as desired for a specific domain.

The work by Bunus & Fritzson (see Bunus and Fritzson [2002]), discussed in Sec. 6.1.4, lies at the other end of the spectrum in terms of precision. Because Bunus et al. work on systems of equations after flattening, they are able to perform a global analysis, which is much more detailed than our type system — or Modelica's balance checking — is capable of performing. For example, Bunus & Fritzon show how their approach can identify specific equations as likely being the cause of a problem and even prioritise among a number of ways to address a problem. In essence, the key difference is that Bunus & Fritzson do an analysis at the granularity of individual variable occurrences, while we approximate this by considering occurrences of variables only at the granularity of two different variable kinds: local and interface variables.

While Bunus and Fritzson's approach does not support checking of components in isolation, and is thus not a feasible starting point for a *type system* for modular equations, their approach does demonstrate the practical utility of taking more fine-grained structural properties into account than just the equation-variable balance.

In summary, in terms of "error finding power", the type systems presented in this chapter are somewhere between what currently is used in Modelica and the approach investigated by Bunus and Fritzson, both of which have shown to be empirically useful for finding problems. Yet, our constraint-based type-based approach offers distinct advantages over both.

4.4.2 Case Study: Half-Wave Rectifier

To demonstrate the practical applications of the type system developed in this chapter, we now present a case study. At this point, the reader may want to first review the examples that were presented earlier in this chapter. These demonstrated the constraint-based type system at work, including how it can catch certain mistakes. However, the examples were small and in some cases also artificial. In contrast, this case study concerns a complete model of a half-wave rectifier composed of a number of electrical components including, in particular, a diode: see Fig. 4.14. We are going to model the diode as an ideal component (initially closed), resulting in a structurally dynamic model. The model, borrowed from a paper on FHM (see Nilsson and Giorgidze [2010]) and originally adapted from Cellier's and Kofman's book *Continuous System Simulation* (see [Cellier and Kofman, 2006, pp. 439-443]), raises particular simulation challenges

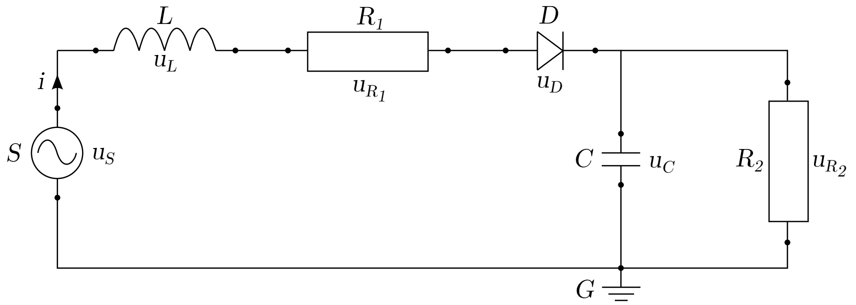

Figure 4.14: Half-wave rectifier with in-line inductor.

as the in-line inductor causes the causality to change when the model switches between the two different structural configurations (i.e. when the ideal diode is open or closed).

Besides the diode, the half-wave rectifier includes a voltage source, an inductor, two resistors, a capacitor, and a ground reference. The implementation of some of these components, such as the resistor, can be found earlier in the paper. However, for convenience the definition of each of these components is given below along with their refined types (with trivially satisfied constraints omitted) and a brief justification for assigning each type.

First of all, recall the definition of *twoPin*, the abstraction that captures the common aspects of electrical components with two pins:

$$\begin{aligned} &\mathit{twoPin} \;:\; () \Rightarrow \mathit{SR}\;(\mathit{Pin}, \mathit{Pin}, \mathit{Voltage})\;2 \\ &\mathit{twoPin} \;=\; \textbf{sigrel}\;(p, n, u)\;\textbf{where} \\ &\quad p.i + n.i \;=\; 0 \\ &\quad p.v - n.v \;=\; u \end{aligned}$$

There are manifestly two (interface) equations and no local variables to solve for, so the net contribution is two equations. All constraints generated are constant inequalities (i.e. they contain no variables) and are trivially satisfiable.

The alternating current voltage source is defined as follows, with the amplitude and frequency given by the parameters v and f, respectively:

$$\mathit{vSourceAC} \;:\; () \Rightarrow \mathit{Voltage} \rightarrow \mathit{Frequency} \rightarrow \mathit{SR}\;(\mathit{Pin}, \mathit{Pin})\;2$$

$$\begin{array}{l}
\mathit{vSourceAC}\ v\ f\ =\ \mathbf{sigrel}\ (p, n)\ \mathbf{where} \\
\quad \mathbf{local}\ u \\
\quad \mathit{twoPin}\ \diamond\ (p, n, u) \\
\quad u\ =\ v * \mathit{sin}\ (\mathit{2} * \pi * f * \mathit{time})
\end{array}$$

Applying the constraint criteria to the voltage source component gives an overall contribution of two equations. This contribution is easily justified. The application of *twoPin* contributes two equations. In this case, they are mixed. The **atomic** equation is local and has to be used to solve for the local variable u, leaving the two mixed equations as the contribution. Applying the typing rules and then simplifying constraints yields the same result.

The resistor, inductor, and capacitor are defined as follows:

$$\begin{array}{l}
\mathit{resistor}\ :\ ()\ \Rightarrow\ \mathit{Resistance}\ \rightarrow\ \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\ \mathit{2} \\
\mathit{resistor}\ r\ =\ \mathbf{sigrel}\ (p, n)\ \mathbf{where} \\
\quad \mathbf{local}\ u \\
\quad \mathit{twoPin}\ \diamond\ (p, n, u) \\
\quad r * p.i\ =\ u
\end{array}$$

$$\begin{array}{l}
\mathit{inductor}\ :\ ()\ \Rightarrow\ \mathit{Inductance}\ \rightarrow\ \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\ \mathit{2} \\
\mathit{inductor}\ i\ =\ \mathbf{sigrel}\ (p, n)\ \mathbf{where} \\
\quad \mathbf{local}\ u \\
\quad \mathit{twoPin}\ \diamond\ (p, n, u) \\
\quad l * \mathbf{der}\ p.i\ =\ u
\end{array}$$

$$\begin{array}{l}
\mathit{capacitor}\ :\ ()\ \Rightarrow\ \mathit{Capacitance}\ \rightarrow\ \mathit{SR}\ (\mathit{Pin}, \mathit{Pin})\ \mathit{2} \\
\mathit{capacitor}\ c\ =\ \mathbf{sigrel}\ (p, n)\ \mathbf{where} \\
\quad \mathbf{local}\ u \\
\quad \mathit{twoPin}\ \diamond\ (p, n, u) \\
\quad c * \mathbf{der}\ u\ =\ p.i
\end{array}$$

Like the voltage source, the relations that result from *resistor*, *capacitor*, and *inductor* (after the application of any functional parameters) each contribute two equations for similar reasons to the above. From the perspective of our type system, the sets of equations that constitute each component are essentially the same: an application of *twoPin* to a set of mixed variables, and an atomic equation. The only difference here is that the atomic equation is mixed.

The *ground* component, unlike previous components, is connected via only a single pin, itself containing two signal variables ($p.v$ and $p.i$):

```
ground : () ⇒ SR Pin 1
ground = sigrel p where
  p.v = 0
```

Its purpose is to set a reference voltage level. Thus, this component is very simple: it contains only a single equation and introduces no new local variables. Hence, our intuition would dictate that the ground component contributes one equation as there are no local variables. This is in agreement with the type assigned by our type system.

The final, and most involved component in the circuit is the diode:

```
icDiode : () ⇒ SR (Pin, Pin) 2
icDiode = sigrel (p, n) where
  local u
  twoPin ⋄ (p, n, u)
  initially; when p.v − n.v > 0 ⇒
    u = 0
  when p.i < 0 ⇒
    p.i = 0
```

The diode is a particularly interesting example as the type of equations contributed is dependent upon the current structural configuration: initially, the switch block defines a local equation, whereas the second branch defines an interface equation. This apparent conflict is resolved thanks to the *fair* policy (Sect. 4.3.2) employed when generating constraints for structurally dynamic code. The two branches of the switch block are reconciled by demanding that a mixed equation is present in the enclosing context. In other words, the switch block contributes 1 interface equation, 1 local equation, and -1 mixed equation. The application of *twoPin* contributes 2 mixed equations, satisfying the demand from the switch block of “borrowing” a mixed equation from the enclosing context. This means we can simply add the contribution from the switch block and from the application of *twoPin*, yielding 1 interface equation, 1 local equation, and 1 mixed equation. The 1 local equation has to be used to solve for the 1 local variable, meaning that the overall contribution from the diode is 2 equations. At this point, it is worth noting that the *strong* approach would be too restrictive: the contributions from the different branches are clearly not identical. The complete half-wave rectifier can now be described as follows:

```
halfWaveRectifier : () ⇒ SR () 0
halfWaveRectifier = sigrel () where
  local lp ln r1p r1n r2p r2n
  local dp dn cp cn acp acn gp
  resistor 1.0          ◇ (r2p, r2n)
  icDiode               ◇ (dp, dn)
  capacitor 0.0         ◇ (cp, cn)
  vSourceAC 1.0 1.0     ◇ (acp, acn)
  ground                ◇ gp
  connect acp lp
  connect ln r1p
  connect r1n dp
  connect dn cp r2p
  connect acn cn r2n gp
```

The rectifier is a non-trivial example, consisting of seven subcomponents, many of which have subcomponents of their own. However, if one follows the typing rules, it is straightforward to construct the appropriate type. There are a total of 26 local variables (recall that each pin contains two variables) and, by no coincidence, the body of the relation contains a total contribution of 26 equations. Note that the **connect** keyword is used as a shorthand for

Kirchhoff's circuit laws, where **connect** p_1 ... p_x desugars to x atomic equations: a sum-to-zero equation and $x - \mathit{1}$ voltage equalities.

The type system does not merely guarantee that the model is balanced, it strengthens the claim by imposing additional constraints that are also satisfied. For example, suppose the programmer made an error in the implementation of diode: instead of applying *twoPin* to a mixed set of variables (i.e. $\mathit{twoPin} \ \diamond \ (p, n, u)$), the application was instead made to a set of interface variables (i.e. $\mathit{twoPin} \ \diamond \ (p, n, \mathit{0})$). In a setting with more than a few interface and local variables it is entirely plausible that such an error might go unnoticed. This mistake would mean that there are no mixed equations to satisfy the -1 mixed equation requirement of the switch block. Interestingly, if one were to only count variables and equations, without any notion of kinds (see related work, Sect. 6.1.1), the aforementioned error would not be detected. Furthermore, in our system this error would be detected early and modularly, while type checking the code for *icDiode* in isolation, and not only once the full model has been assembled, as is the case in the work by Bunus and Fritzson [2002].

Chapter 5

A Semantic Model of FHM

5.1 Preliminaries

5.1.1 Outline

In recent years the popularity of acausal, hybrid modelling languages has grown substantially. This highly modular and declarative approach to modelling has proven to be very useful in practice when designing large and complex systems of equations. Increased interest in these languages has spurred interest in their semantics, with specific approaches tending to focus on either the discrete or continuous aspects of the language (see Broman [2010], Giorgidze [2011], Henzinger [1996], Kågedal [1998], Kågedal and Fritzson [1998], Pepper et al. [2011]). However, previous attempts have been restricted to languages without first-class components or structural dynamism, have produced non-compositional semantics, and/or have not attempted to show any correctness properties.

In this chapter, we attempt to give a semantic model for the *discrete aspects* of an FHM-like language that covers all of the above issues. Our semantic model is parameterised by the *continuous behaviour*, where such aspects merely describe (an approximation of) a solution to the equations describing a particular structural configuration, if such a solution exists. This separation of concerns is deliberate: how prescriptive the continuous aspects of the semantics should be depends on the purpose in hand; consequently, we wish to formalise the discrete part of the semantics in such a way that it fits with any reasonable approach describing the continuous part. For example, one view of our semantics is describing the computation of fragments of a hybrid automata on demand. Thus, work focusing solely on the continuous aspects of the language should be both complementary and orthogonal to our own.

The discrete semantics of FHM can be separated into two aspects:

1. How a modularly composed model is transformed into a flat system of equations that describes the continuous-time dynamic behaviour for a particular structural configuration.

2. How new structural configurations are computed in response to events triggered by the dynamic behaviour of the current configurations.

We formalise a compositional semantics for both of the above aspects in type theory. As our system is parameterised by the continuous aspects, events triggered by the continuous behaviour are left abstract and are injected into the model via an oracle that is postulated in our formalisation.

This chapter makes the following contributions:

1. A compositional semantics for the discrete aspects of an acausal, hybrid, structurally dynamic modelling language expressed in dependent type theory using Normalisation by Evaluation.

2. A novel formalisation of dynamism and the generation of new structural configurations that is declarative, avoiding the traditional imperative bias common to other approaches to dynamism.

3. A semantics that is carefully structured so as to allow the continuous aspects to be described separately, in whatever way is most appropriate for the purpose at hand, while retaining the ability to describe precisely how a system evolves in response to discrete events.

4. A mechanised proofs of type preservation, termination and totality, and *normalisation* for the semantics.

The basis of the work in this chapter was first published in Capper and Nilsson [2012]. The work was then further developed with the proof of normalisation in Sect. 5.3 and the extensions appearing in Sect. 5.5. The prerequisites for this chapter are Chap. 2 and Chap. 3.

5.1.2 Models and Metalanguages

Work on denotational semantics stretches back many years. A particularly notable milestone was set by Scott and Strachey in the early 1970s with their work on denotations for recursively defined programs (see [Scott and Strachey, 1971]). They proposed working with continuous functions between domains (specifically *complete partial orders*) to denote partial and recursive programs. However, of particular importance is the compositionality of the semantics: the denotation of a phrase is given entirely by the denotations of its sub-phrases.

We want to construct a compositional semantics for an FHM-like language. In a similar fashion to H_{Δ} and $H_{\vDash}$, the setting is a simply-typed calculus and we are not concerned with partial or general recursive functions, and by extension Scott-style continuous functions and domains. Instead, we choose a type-theoretic basis, specifically Martin-Löf Type Theory (MLTT) [Martin-Löf, 1984], as realised by the dependently typed programming language and proof assistant Agda [Bove et al., 2009].

The only constraint that we wish to place on our choice of type theory is that it is constructive: the specification of our model in such a theory is simultaneously a semantics and an implementation, intimately relating the two concepts. Of course, we could easily choose another constructive theory of types, or work within a different proof assistant (e.g. Coq). That said, Agda has a number of advantages:

1. Agda has a powerful termination checker.

2. Agda is based on a strong theory of types, which makes it easy to specify and prove theorems about our formulation.

3. Agda has a flexible syntax that permits Unicode symbols and mixfix operators, allowing the syntax of our object language to very closely resemble that of Hydra.

4. Agda is similar to other popular, contemporary functional languages, such as Haskell, hopefully making the code presented in this thesis accessible to a wider audience.

5.1.3 Embedding a Model

The *depth* of a language embedding suggests how closely the syntax and evaluation model of the object language are tied to that of the metalanguage (i.e. Agda). Thus, it is important to consider the implications of the embedding depth when deciding on a specific approach to implementing the semantics.

At one end of the spectrum is the shallow embedding of Higher-Order Abstract Syntax (HOAS) ([Pfenning and Elliot, 1988]), which makes direct use of both the syntax and the reduction machinery of the host language. At the other end of the spectrum are deep embeddings: terms are given as data and reduction is specified as a function on the data.

Using a deep embedding allows one to define and prove properties by induction on the structure of a formula, for example, one could reason directly about the depth of a derivation. Unfortunately, this approach can be intricate and error-prone: the details need to be spelled out carefully and the evaluation mechanisms of the metalanguage are left unexploited. For example, one needs to deal explicitly with the binding and substitution of variables.

Using a shallow embedding allows one to exploit features of the metalanguage, and it is often quite convenient for "importing" results proved in the metalanguage. Unfortunately, this can make shallow embedding, such as HOAS, very permissive, allowing a large, unrestricted set of functions to be defined.

For example, functions can be defined using pattern matching and intermediary data structures. We want a semantics for a much simpler language and do not want our term language to be contaminated with arbitrary metalanguage terms. Additionally, standard definitions of HOAS for the λ-calculus are typically not *strictly positive* (see [Abbott et al., 2005]), which is a requirement in Agda. Simple and clear reasoning about HOAS in type theory that does not violate the positivity requirements of Agda is being explored by [Capretta and Felty, 2009] and may provide an intermediate solution to the problem.

Fortunately, there is a middle ground between these two approaches: we can use metalanguage objects as the semantic domains. This turns out to be ideal for our specific needs. The terms of the language are defined as data, in line with the deep embedding, which are then interpreted as metalanguage objects, allowing us to use Agda's evaluation mechanisms to reduce terms. The middle-ground procedure we use is called *Normalisation by Evaluation*.

Generally speaking, normalisation refers to the process of finding a normal form in a term rewriting system (i.e. an open term for which no rewrite equations apply). In our work, normalisation can be thought of as a reduction-free process that extends evaluation to work on open terms with the rewriting system given by the equational theory of the language (i.e. the theory that specifies which terms are related to one another via the reduction rules of language).

5.1.4 Normalisation by Evaluation

Normalisation by Evaluation (NbE) was first described in the early 1990s for the simply-typed λ-calculus [Berger and Schwichtenberg, 1991]. Among other advances it has been extended to richer theories such as Martin-Löf type theory [Abel et al., 2007]. NbE has shown to be particularly useful in the implementation of proof assistants (e.g. MINLOG [Slaney, 1997]) and dependently-typed languages as a method to normalise proof terms and find normal forms for types, often an essential aspect of type equality. NbE is also typically type directed

— the denotation of a program phrase is governed by the type of the program — and is thus closely related to *type-directed partial evaluation* [Danvy, 1996].

Once the choice to use NbE has been made, suitable approaches to a number of aspects of the work follow almost mechanically. The target model serves as a denotation for our language, and together the normalisation and reification functions serve as both an implementation and as a constructive proof that the model is a faithful representation of the language. This technique produces β-normal, η-long normal forms, and importantly, the model and evaluation are both compositional.

Monoid Expressions

To explain the steps of NbE, and to help the reader become accustomed with our approach and presentation of NbE, we begin with a very simple example: a monoid over elements of a given set A. From this point onward we present our implementation as pseudo-Agda code, as described in Chap. 3.

A monoid expression consists of an identity element (id), an associative binary operator (_∘_), and the elements of the underlying set (var).

```
data Expr (A : Set) : Set where
  id  : Expr A
  _∘_ : (x y : Expr A) → Expr A
  var : (a : A)        → Expr A
```

The first step is to define an appropriate model for our expression language. The objects of the model represent the normal forms, thus, expressions with the same normal form (i.e. expressions that are convertible to one another) should be represented by the same object. In other words, the model identifies expressions up to the equational theory, which in this instance is the three monoid laws: $\mathsf{id} \circ x = x$, $x \circ \mathsf{id} = x$, and $(x \circ y) \circ z = x \circ (y \circ z)$. In this example a normal form can be found via a simple technique: shuffle all the parentheses to the right and eliminate the composed identities; that is, let us take our convertibility relation to be a set of rewriting rules.

The free monoid of lists could be used as a model here. However, this representation requires us to define an inductive list datatype and a recursive concatenation function. Instead, a simpler representation (in the sense that it does not require any new definitions) is to view a monoid as a function space. Composition is used instead of concatenation and the “list” of expressions that results from normalisation is terminated by id rather than the empty list.

```
Model : Set → Set
Model A = Expr A → Expr A
```

The monoid identity is then interpreted as the identity function, and the binary operator is interpreted as function composition. Using this interpretation allows us to exploit the evaluation mechanisms of the metalanguage.

```
⟦_⟧ : Expr A → Model A
⟦ id      ⟧ x = x
⟦ y ∘ z   ⟧ x = ⟦ y ⟧ (⟦ z ⟧ x)
⟦ var a   ⟧ x = var a ∘ x
```

The above interpretation function corresponds to completeness of the model; for every monoid expression there exists a suitable object in the model. Conversely, reification shows that the model is a sound representation; for every object in the model there is a corresponding monoid expression. As the expression language is essentially untyped, reification is straightforward.

```
reify : Model A → Expr A
reify f = f id

nbe : Expr A → Expr A
nbe x = reify ⟦ x ⟧
```

There are two important theorems that capture the correctness of normalisation. We use $\simeq$ to denote convertibility and $\equiv$ to denote propositional equality, the theorems are as follows:

$$\mathsf{nbe}\ t \simeq t \tag{5.1a}$$

$$s \simeq t \Rightarrow [\![\ s\ ]\!] \equiv [\![\ t\ ]\!] \tag{5.1b}$$

Theorem 5.1a states that a term should be convertible to its normal form. In isolation this means that nbe performs enough reductions to make equational convertibility syntactically decidable. This guarantee is strengthened in subsequent sections as we construct nbe such that it is guaranteed to produce normal forms free of redexes. This says something very strong about the behaviour of our normaliser. Namely, that normalisation performs only valid reductions and that it performs reductions until *all* redexes have been eliminated.

Theorem 5.1b states that convertible terms are represented by the same objects in the model. If the equational theory states that two programs are equivalent then they should have the same denotation (i.e. the same meaning). Taken together the two theorems tell us that a term is convertible to its normal form, and that all convertible terms have syntactically equal normal forms.

A number of interesting corollaries can also be derived from the two primary theorems. From theorem 5.1a, one can derive corollaries 5.2a, 5.2c, and the backward direction of 5.2d. Theorem 5.1b gives rise to the remaining corollaries 5.2b and the forward direction of 5.2d.

$$\mathsf{nbe}\ (\mathsf{nbe}\ t) \simeq \mathsf{nbe}\ t \tag{5.2a}$$

$$\mathsf{nbe}\ (\mathsf{nbe}\ t) \equiv \mathsf{nbe}\ t \tag{5.2b}$$

$$s \simeq t \Leftrightarrow \mathsf{nbe}\ s \simeq \mathsf{nbe}\ t \tag{5.2c}$$

$$s \simeq t \Leftrightarrow \mathsf{nbe}\ s \equiv \mathsf{nbe}\ t \tag{5.2d}$$

Normalisation by evaluation provides a *reduction-free* view of evaluation: the semantics is specified by translation into a model rather than by a sequence of reductions. As such, we need not be concerned with proofs of reduction-based theorems, for example, the first *Church-Rosser* theorem (confluence).

Typed Expressions

Before starting work on FHM in earnest we take a last detour to visit an example that highlights important aspects of NbE that were not covered by the previous example. In particular, we consider a *typed* expression language that demonstrates the *type-directed* nature of NbE. Furthermore, the structure of this example bears many similarities to that of our FHM development.

We begin by defining codes for the types in our language. The types are very simple; we provide natural numbers and Boolean values:

```
data Type : Set where
  nat  : Type
  bool : Type
```

The terms of the language are given by the type-indexed family Term. The index ensures that only well-typed expressions are considered as we are not interested in attempting to normalise ill-typed terms:

```
data Term : Type → Set where
  zero : Term nat
  succ : Term nat → Term nat
  add  : Term nat → Term nat → Term nat
  tt   : Term bool
  ff   : Term bool
  if   : Term bool → Term τ → Term τ → Term τ
```

The normal forms of the language are the terms that are redex-free. As we do not have access to any form of subtyping in Agda we express the normal forms as a *new* datatype. Nrm is intended to be a subset of Term and an embedding function embed allows normal forms to be translated back into terms. The definition of embed is straightforward and we omit it here.

```
data Nrm : Type → Set where
  zero : Nrm nat
```

```
succ : Nrm nat → Nrm nat
tt   : Nrm bool
ff   : Nrm bool
```

The definition of the model shows that our approach is type directed. The model is defined by matching on the type of the expression being modelled. For each case an Agda type is used to denote the meaning, for example, nat is denoted by the set of natural numbers ℕ. At this point it is worth noting that we have constructed a simple *universe* [Martin-Löf, 1975], Type are the codes and Model is the decoding function mapping codes to types. We will see this design pattern repeated for FHM, but with a more interesting type of codes.

```
Model : Type → Set
Model nat  = ℕ
Model bool = Bool
```

The interpretation function is straightforward, mapping expressions in the object language to their counterparts in the metalanguage.

```
⟦_⟧ : Term τ → Model τ
⟦ zero      ⟧ = 0
⟦ succ n    ⟧ = 1 + ⟦ n ⟧
⟦ add n m   ⟧ = ⟦ n ⟧ + ⟦ m ⟧
⟦ tt        ⟧ = true
⟦ ff        ⟧ = false
⟦ if b s t  ⟧ = if ⟦ b ⟧ then ⟦ s ⟧ else ⟦ t ⟧
```

We deviate from the previous example by reifying objects to normal forms (i.e. Nrm) rather than terms (i.e. Term). Targeting normal forms requires the model to be complete with respect to the smaller, redex-free set of normal forms. Thus, in turn, the process of interpreting and reifying a term is guaranteed to produce a value. Reification of the model is also more interesting, depending first and foremost on the type of the model.

```
reify : (τ : Type) → Model τ → Nrm τ
reify nat  0         = zero
reify nat  (1 + n)   = succ (reify nat n)
reify bool true      = tt
reify bool false     = ff
```

To complete the round trip to and from the model we must convert normal forms back into terms using embed.

```
nbe : Term τ → Term τ
nbe {τ} t  =  embed (reify τ ⟦ t ⟧)
```

5.2 A Semantic Model of FHM

5.2.1 $H_{[\![\,]\!]}$: A Core Language for a Semantic Model

For the third time in this thesis we are in need of a core language to form the basis of a formalisation. As before, the core language is designed with the specific needs of the formalism in mind. We are interested in showing the correctness of the semantics, and importantly, we need a core language that is representative of FHM and Hydra as a whole. With this in mind, we present $H_{[\![\,]\!]}$ using the proof assistant Agda.

We make a number of changes to earlier core languages that make $H_{[\![\,]\!]}$ more suitable for our specific needs here. We aim to stay honest to the original presentation of FHM by including signal-level expressions for the first time in this thesis. However, even in this setting the interaction between the signal level and the rest of the language is simple. We are not attempting to simplify signal expressions in the hope of finding a solution to the equations; this is the job of the continuous semantics upon which our semantics is parameterised. Indeed, the genuinely interesting interactions in the discrete semantics occur at the boundary between equations and the functional host language, and between different structural configurations.

In contrast to H_{Δ} and $H_{\vDash}$, where the simply-typed λ-calculus was used as the host language, here we opt to use λ_{σ}: the λ-calculus with explicit substitutions [Abadi et al., 1991]. Explicit substitutions are useful for a number of reasons:

1. It is easier to express and reason about the calculus in Agda as the computational rules of the language can be expressed as a binary relation indexed solely by values.

2. Substitutions can be used to give a simple definition of closures, which are used in the formalisation (see Sect. 5.2.1).

3. Substitutions allow us to suspend evaluation, which provide a means to *delay branch normalisation*, an extension found in Sect. 5.5.2.

Explicit substitutions are not strictly necessary to achieve the results presented later in this chapter, nor do they enhance their value. However, the first item in the above list is a particularly compelling reason to use them. Explicit substitutions allow our approach to be syntax directed. The indices of constructors are restricted to data, which is far easier for Agda to unify. In particular, the computational rules of the language can be expressed as a binary relation indexed solely on values. In a recent article, McBride [2014] articulates this viewpoint, in which he argues against the use of "green slime" (i.e. functions used to compute indices from structure, rather than impose it).

The normalisation procedure we present in this chapter for $H_{[\![]\!]}$ produces β-normal, η-long normal forms. Thus, a normal form does not contain any β-redexes, including terms under binders, and has been η-expanded as far as possible without introducing β-redexes. As an example, consider normalising the identity function $\lambda\, f\, .\, f$ at the type $(\alpha\ \rightarrow\ \beta)\ \rightarrow\ (\alpha\ \rightarrow\ \beta)$ to produce the normal form $\lambda\, x\, .\, \lambda\, y\, .\, x\ y$. To simplify the initial development we abide by this aggressive approach when normalising branches of switch statements. However, in Sect. 5.5 we investigate different approaches to allow delayed evaluation of switch branches as seen in the earlier core languages. Furthermore, we reserve treatment of local variables until a dedicated section in Sect. 5.5. Rather than identifying a local variable we merely tag its location instead, once again to simplify the initial presentation.

Types

An inductively defined datatype can also be viewed as a recursively defined grammar. Equally, an indexed family of datatypes can be viewed as a set of deductive rules. For example, the dependently-typed language Epigram provides a two-dimensional syntax to emphasise this point [McBride, 2004]. Consequently, to avoid repetition, we present the grammars and rules of $H_{\llbracket\rrbracket}$ directly as Agda datatypes from here onward.

The $H_{\llbracket\rrbracket}$ language has a simple type system that is stratified into two levels: one level for functional types and one for signal types. This reflects the stratified nature of FHM. By contrast to the previous core languages, signal relation types are now parameterised by a signal-level type, which consists of the unit type (for representing nullary signal relations) and signal-level products. For simplicity, we use an uninterpreted numeric type *Num*, which the reader can consider as a placeholder for a more appropriate type, such as floating point numbers.

```
data SType : Set where
  unit  : SType
  num   : SType
  _×_   : SType → SType → SType

data Type : Set where
  num  : Type
  _→_  : Type   → Type → Type
  sr   : SType → Type
```

Contexts and Substitutions

The meaning of an expression is not given in isolation but instead exists within a (possibly empty) typing context. A context associates each free variable with a type. In this formulation, rather than identifying variables using a token, we opt for a nameless approach using de Bruijn indices [de Bruijn, 1972]. This representation obviates the need for variable freshness conditions and makes α-equivalent terms definitionally equivalent, which is particularly helpful for formulating the system in Agda. Under this approach, contexts are an injection from indices to types. Thus, one implementation is a list of types with the index given by the position in the list.

```
data Ctx : Set where
  ○   : Ctx
  _▷_ : Ctx → Type → Ctx
```

Following Hydra, we allow only one *interface signal variable* to be bound at a given point. With the reintroduction of the signal level and tuples in $H_{\models}$ it once again becomes possible to express a signal relation over multiple interface signals using just one variable and tupling.

```
data SCtx : Set where
  ○ : SCtx
  ▷ : SType → SCtx
```

Working in the λ_σ-calculus dictates that we find a representation for substitutions. As the name suggests, a *substitution* describes a method for replacing the free variables in an expression with terms of an appropriate type. *Renaming* is a very similar operation where one replaces the variables in an expression with other variables. As such, we keep our representation general, allowing variables to be replaced by arbitrary "term-like" objects by defining a general notion of *replacement* ($\Rightarrow$). An object is term-like if it has a type in a given context. For example, terms, variables, and normal forms are all term-like objects.

One way to express the above is to give the relationship between the contexts before and after the replacement is applied. Replacements should also preserve the type of the variable being replaced. We take a conventional approach by providing syntax for both extension and weakening. Replacements also form a category with an identity and composition:

```
TmLike, STmLike : Set₁
TmLike  = Ctx  → Type  → Set
STmLike = SCtx → SType → Set

data _⇒_ {T : TmLike} : Ctx → Ctx → Set where
  id  : Γ ⇒ Γ
  _∘_ : Γ ⇒ Δ → Δ ⇒ E → Γ ⇒ E
  _▷_ : Γ ⇒ Δ → T Δ τ   → (Γ ▷ τ) ⇒ Δ
  wkn : Γ ⇒ (Γ ▷ τ)

data _⇒S_ {S : STmLike} : SCtx → SCtx → Set where
  id  : Φ ⇒S Φ
  _∘_ : Φ ⇒S Ψ → Ψ ⇒S X → Φ ⇒S X
  ▷   : S Φ σ     → (▷ σ) ⇒S Φ
```

The implicit parameters T and S denote the term-like objects that will be used as replacements for variables when extending. These parameters will be omitted when they can be inferred from usage, as is the case in the above definitions. When the instantiation is not obvious a superscript will be used to indicate the intended type. The meaning of each replacement is as follows:

- id: The identity replacement, mapping indices back to themselves.

- $\gamma \circ \delta$: Composition of replacements, realised by first applying the replacement γ, followed by applying the replacement δ.

- wkn: Weakening is the shift replacement, incrementing each index. Weakening is not required for signal-level replacements as they contain at most one index that cannot be incremented.

- $\gamma \triangleright t$: Extension is the replacement of the first variable for the object t, followed by the weakened application of the replacement γ. That is, if Γ can be replaced by Δ, and there exists an object t in Δ, then Γ can be extended to use t as a replacement for the first variable.

There are number of useful auxiliary definitions and properties related to the replacements. Notably, replacements are functorial in the object parameter.

$$\begin{aligned}
&\mathsf{map} : (X\ E\ \tau \to Y\ E\ \tau) \to \Gamma \Rightarrow^{X} \Delta \to \Gamma \Rightarrow^{Y} \Delta\\
&\mathsf{map}\ f\ \mathsf{id} && = \mathsf{id}\\
&\mathsf{map}\ f\ (\gamma \circ \delta) && = \mathsf{map}\ f\ \gamma \circ \mathsf{map}\ f\ \delta\\
&\mathsf{map}\ f\ (\gamma \triangleright t) && = \mathsf{map}\ f\ \gamma \triangleright f\ t\\
&\mathsf{map}\ f\ \mathsf{wkn} && = \mathsf{wkn}
\end{aligned}$$

There also exists a replacement from the empty context into any other context. To help understand this type consider specialising $\Rightarrow$ to renamings, that is, T is instantiated as de Bruijn indices. Applying the empty remaining to a term containing no free variables states that variables may be freely added and renamed without causing name clashes.

$$\begin{aligned}
&\mathsf{empty} : \{X : \mathsf{TmLike}\} \to (\Gamma : \mathsf{Ctx}) \to \circ \Rightarrow^{X} \Gamma\\
&\mathsf{empty}\ \circ && = \mathsf{id}\\
&\mathsf{empty}\ (\Gamma \triangleright \tau) && = \mathsf{empty}\ \Gamma \circ \mathsf{wkn}
\end{aligned}$$

These definitions of contexts and substitutions for a nameless representation in dependent type theory are very similar to the standard approaches taken in much of the existing literature. We choose to follow the approach of Abadi et al. [1991] et al. in our choice of substitution primitives, and our encoding of substitution is very similar to the work of Chapman [2009]. However, our representation is slightly more flexible as it is generalised to any term-like object.

Well-typed Terms

The typical pen-and-paper approach to formalising a language – as exemplified by both H_{Δ} and $H_{\models}$ – is to first define the raw terms, followed by a typing relation that relates terms, contexts, and types. However, an alternative approach in dependent type theory is to define data that is *correct by construction* (for a suitable notion of correct). In this instance, we wish to define the terms that are well-typed and well-scoped. In doing so, we define not only the terms themselves, but also typing derivations. The distinct advantage of this approach is that from here on out we need only consider the well-typed, well-scoped terms when giving definitions. Additionally, it becomes straightforward to state meta-theoretical properties of terms when the type and context of the term appear immediately as an index. This notion of *correct by construction* also sits at the heart of the correctness proof found in Sect. 5.3.

Once again, the stratified terms of FHM are mirrored by the mutually inductive datatypes given below for the functional level (Tm), the signal level (STm), equations (QTm), and switch blocks (Switch).

The bracket notation (e.g. $[\gamma]\ t$) denotes the application of a functional substitution γ to the term t. This operation, inherited from the λ_σ-calculus, expresses explicit substitutions in the term language. In the same manner, a signal-level substitution is denoted by angled brackets (e.g. $\langle\ \phi\ \rangle\ s$).

The shape of a substitution application is given via a Closure. For a given context-indexed type T, Closure states that T is functorial with respect to substitution (i.e. a substitution can be mapped across the structure of T). To convince ourselves that this is the correct specification, take T to be the language of terms. Closure then becomes: $\Delta \Rightarrow \Gamma\ \to\ \mathsf{Tm}\ \Delta\ \tau\ \to\ \mathsf{Tm}\ \Gamma\ \tau$, which corresponds to the usual notion of a term paired with a substitution. For readability we will adopt the notation $\mathsf{Tm}\ \cdot\ \tau$ as shorthand for $\lambda\,\Gamma\ \to\ \mathsf{Tm}\ \Gamma\ \tau$. Note that this notation is not valid Agda syntax and is just used in this thesis to denote partial application.

```
Closure : (Ctx → Set) → (Ctx → Set)
Closure T Γ = Δ ⇒ Γ → T Δ → T Γ

SClosure : (SCtx → Set) → (SCtx → Set)
SClosure S Φ = Ψ ⇒S Φ → S Ψ → S Φ
```

The var, lam, and app constructors represent the well-known method of defining well-typed λ-terms in dependent type theory. The only exception being var, which, due to our formulation of explicit substitutions, represents only the most recently bound variable. Other variables can be constructed through explicit weakening, for example, the third most recently bound variable can be constructed as [wkn ∘ wkn] var. The remaining Tm constructors are lit, which is simply a token to represent numeric literals, and sigrel, which embeds equations.

```
data Tm : TmLike where
  lit    : Tm Γ num
  var    : Tm (Γ ▹ τ) τ
  lam    : Tm (Γ ▹ τ1) τ2     → Tm Γ (τ1 → τ2)
  app    : Tm Γ (τ1 → τ2) → Tm Γ τ1 → Tm Γ τ2
  sigrel : QTm Γ (▹ σ)        → Tm Γ (sr σ)
  [_]_   : Closure (Tm · τ) Γ
```

The goal of this chapter is to present a semantic model of an FHM-like language and at the same time to present an understandable implementation. In keeping with the latter goal, we defer a proper handling of local variables until Sect. 5.5 as they add another layer of complexity and syntactic noise that we believe detracts from the overall message of this section. With that in mind, local variables are represented using the anonymous token lvar.

The fun constructor allows time-invariant functional expressions to be embedded into the signal level. Terms of an arbitrary type cannot be embedded as the functional level and signal level do not have a perfect intersection of types. Of course, one could specify a predicate for the common types and provide marshalling between these types, but little is gained from doing so in this formalisation. We restrict the type of the embedded terms to num.

The remaining signal expressions represent the kind of operations one would typically find in an equation-based modelling language. Pairs allow us to abstract over and build compound signal expressions.

$$
\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{data}\ \mathsf{STm}\ (\Gamma\ :\ \mathsf{Ctx})\ :\ \mathsf{STmLike}\ \mathsf{where}} \\
\quad \mathsf{tt} & : \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{unit} & & & & \\
\quad \mathsf{svar} & : \mathsf{STm}\ \Gamma\ (\triangleright\ \sigma)\ \sigma & & & & \\
\quad \mathsf{lvar} & : \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{num} & & & & \\
\quad \mathsf{fun} & : \mathsf{Tm}\ \ \Gamma\ \ \ \mathsf{num} & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{num} & & \\
\quad \mathsf{binop} & : \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{num} & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{num} & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \mathsf{num} \\
\quad \mathsf{pair} & : \mathsf{STm}\ \Gamma\ \Phi\ \sigma_1 & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \sigma_2 & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ (\sigma_1 \times \sigma_2) \\
\quad \mathsf{fst} & : \mathsf{STm}\ \Gamma\ \Phi\ (\sigma_1 \times \sigma_2) & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \sigma_1 & & \\
\quad \mathsf{snd} & : \mathsf{STm}\ \Gamma\ \Phi\ (\sigma_1 \times \sigma_2) & \rightarrow & \mathsf{STm}\ \Gamma\ \Phi\ \sigma_2 & & \\
\quad [\_]\_ & : \mathsf{Closure}\ \ \ (\mathsf{STm}\ \cdot\ \Phi\ \sigma)\ \Gamma & & & & \\
\quad \langle\_\rangle\_ & : \mathsf{SClosure}\ (\mathsf{STm}\ \Gamma\ \cdot\ \sigma)\ \Phi & & & &
\end{array}
$$

Equations are once again described as a tree using the pairing operation ($\wedge$) with leaves consisting of atomic equations ($=$), signal relation applications ($\diamond$), and switch blocks (switch). The only notable deviation from FHM and the earlier core languages is the design of switching blocks. Rather than specifying an initial branch with other branches that may become active during simulation, we instead describe a finite, fixed-length vector of branches with one currently active branch chosen by an argument of type Fin. Here, the type Fin n represents a finite set containing exactly n elements. There are a number of advantages specific to this implementation:

- The representation is more uniform, no special status is given to the initial branch. Thus, the initial branch can now be reactivated as all branches have switching conditions. Note that in $H_{\models}$, a switching condition is simply a signal s of type num that is "activated" when s crosses zero.

- The number of branches is verifiably fixed throughout evaluation and simulation, the length index enforces this invariant by construction.

- When we discuss the semantics of dynamism in Sect. 5.4 the use of Fin will prevent switching events selecting non-existent branches.

```
data QTm (Γ : Ctx) (Φ : SCtx) : Set where
  empty  : QTm Γ Φ
  _∧_    : QTm Γ Φ        → QTm    Γ Φ     → QTm Γ Φ
  _=_    : STm Γ Φ σ      → STm    Γ Φ σ   → QTm Γ Φ
  _◇_    : Tm  Γ (sr σ)   → STm    Γ Φ σ   → QTm Γ Φ
  switch : Fin n          → Switch Γ Φ n   → QTm Γ Φ
  [_]_   : Closure  (QTm · Φ) Γ
  ⟨_⟩_   : SClosure (QTm Γ ·) Φ
```

```
data Switch (Γ : Ctx) (Φ : SCtx) : ℕ → Set where
  []      :   Switch Γ Φ 0
  branch  :   Switch Γ Φ n → STm    Γ Φ num
            → QTm    Γ Φ   → Switch Γ Φ (1 + n)
  [_]_    :   Closure  (Switch · Φ n) Γ
  ⟨_⟩_    :   SClosure (Switch Γ · n) Φ
```

It is worth noting that, when elaborated to pair of a substitution and a term, our usage of closures is the same approach to substitution application as taken by Danielsson [2006], which in turn is inspired by Abadi et al. [1991].

A Comparison to Hydra

We take a break from definitions to take a quick look at how $H_{[\![]\!]}$ compares to Hydra. A notable extension compared to the previous core languages is a more thorough account of the signal level. Note that we retain infix operator syntax and numeric literals for clarity in the example in Fig. 5.1. Also of note is the anonymity of local variables, which, to reiterate, will be addressed in subsequent sections. Finally, without access to pattern matching, we must explicitly project the components of the local variable.

$$\begin{aligned}
&par\ t\ =\ \mathbf{sigrel}\ (p, n)\ \mathbf{where}\\
&\quad \mathbf{local}\ p_1\ p_2\ n_1\ n_2\\
&\quad t\ \diamond\ (p_1, n_1)\\
&\quad t\ \diamond\ (p_2, n_2)\\
&\quad p.i\ + p_1.i + p_2.i\ =\ 0\\
&\quad n.i\ + n_1.i + n_2.i\ =\ 0\\
&\quad p.v\ =\ p_1.v\\
&\quad n.v\ =\ n_1.v\\
&\quad p.v\ =\ p_2.v\\
&\quad n.v\ =\ n_2.v
\end{aligned}$$

$$\begin{aligned}
&par\ =\ \mathsf{lam}\ (\mathsf{sigrel}\\
&\quad \mathsf{var} \diamond \mathsf{pair\ lvar\ lvar} \wedge\\
&\quad \mathsf{var} \diamond \mathsf{pair\ lvar\ lvar} \wedge\\
&\quad \mathsf{fst}\ \ (\mathsf{fst}\ \ \mathsf{svar}) + \mathsf{fst\ lvar} + \mathsf{fst\ lvar} = 0 \wedge\\
&\quad \mathsf{fst}\ \ (\mathsf{snd\ svar}) + \mathsf{fst\ lvar} + \mathsf{fst\ lvar} = 0 \wedge\\
&\quad \mathsf{snd}\ (\mathsf{fst}\ \ \mathsf{svar}) = \mathsf{snd\ lvar} \wedge\\
&\quad \mathsf{snd}\ (\mathsf{snd\ svar}) = \mathsf{snd\ lvar} \wedge\\
&\quad \mathsf{snd\ lvar} = \mathsf{snd\ lvar} \wedge\\
&\quad \mathsf{snd\ lvar} = \mathsf{snd\ lvar} \wedge)
\end{aligned}$$

Figure 5.1: Comparison of Hydra and $H_{[\![]\!]}$.

5.2.2 A Model of $H_{[\![]\!]}$

Normal Forms

Our next goal is to write a normalisation function that *flattens* a modular system of equations into an initial set of equations to begin simulation. In the context of λ-calculi, normalisation typically manifests itself as a map from open terms to terms that contain no reducible expressions (redexes); that is, $\mathsf{nbe}\ :\ \mathsf{Tm}\ \Gamma\ \tau\ \ \rightarrow\ \ \mathsf{Tm}\ \Gamma\ \tau$. Thus, the obvious codomain for nbe would be to reuse the representation of terms from the previous section (Sect. 5.2.1), with the image of nbe being the subset of Tm that represents *redex-free* terms.

However, it turns out to be far more useful, particularly for the purposes of showing correctness, to create a new set of datatypes that contain exactly the image of nbe and nothing more. That is, the inhabitants of the new datatypes are the well-scoped, well-typed terms that are β-normal and η-long. The typed expressions example from Sect. 5.1.4 shows this method at work for a simple expression language without binding.

The representation of normal forms need not contain closures as any closure can be eliminated by carrying out a substitution during evaluation. In order to exclude substitution application, an explicit representation of non-zero-indexed variables is needed, as variables with an index greater than 0 can no longer be represented via substitutions applied to the 0^{th} variable.

```
data Var : TmLike where
  vz : Var (Γ ▷ τ) τ
  vs : Var Γ τ₁ → Var (Γ ▷ τ₂) τ₁

data SVar : STmLike where
  vz : SVar (▷ σ) σ
```

A functional variable is encoded as a typed de Bruijn index into a typing context. The constructor vz points to the end of the context; that is, the most recently bound variable. The constructor vs weakens a variable by extending the context into which it points. Intuitively, the type Var Γ τ can be thought of as the set of indices pointing to variables of type τ in the context Γ. As there will only ever be a single signal-level variable in scope, a new definition is not strictly necessary, but is included for symmetry.

The following embeddings allow us to convert back to the previous approach of implicitly formulated de Bruijn variables:

```
embVar : Var Γ τ → Tm Γ τ
embVar  vz      = var
embVar (vs v)  = [wkn] (embVar v)

embSVar : SVar Φ σ → STm Γ Φ σ
embSVar vz = svar
```

With the goal of creating a canonical representation of terms in mind, the Base and SBase predicates identify *base types*. Values of base type will be left uninterpreted, such as numeric values. During normalisation, all values will be η-expanded as far as possible (without introducing β-redexes), unless they can be shown to be of base type.

```
data Base : Type → Set where
  num : Base num

data SBase : SType → Set where
  unit  : SBase unit
  num   : SBase num
```

The approach taken to define normal forms is to partition the representation into a pair of mutually defined datatypes that *stratify* the terms such that occurrences of β-redexes are prevented. The two strata are often called *normal terms* and *neutral terms*. Normal terms are those terms whose outermost constructor is known. Conversely, neutral terms are those terms for which reduction has been impeded by the presence of a variable in a key position (e.g. a variable being applied to an argument). Using this approach, terms representing object-level constructors or values are considered normal terms, while variables and object-level destructors are consider to be neutral. It is then easy to compose normal and neutral terms such that destructors are never directly applied to constructors, thus forbidding the construction of redexes.

Given below is the stratification of the functional level. In the typed expression example from Sect. 5.1.4, neutral terms were not required as there were no bound variables that impeded reduction.

```
data Nrm : TmLike where
  lit    : Nrm Γ num
  lam    : Nrm (Γ ▷ τ₁) τ₂  → Nrm Γ (τ₁ → τ₂)
  sigrel : QNrm Γ (▷ σ)     → Nrm Γ (sr σ)
  neu    : Base τ           → Neu Γ τ  → Nrm Γ τ

data Neu : TmLike where
  var  : Var  Γ τ           → Neu  Γ τ
  app  : Neu Γ (τ₁ → τ₂)    → Nrm Γ τ₁  → Neu Γ τ₂
```

There are two key ideas behind this representation. Firstly, the stratification prevents *destructors* being directly applied to *constructors*, for example, preventing the left subtree of app being a lam. This restriction prevents the occurrence of β-redexes. Secondly, the neu constructor only permits neutral terms to appear as normal terms at a base type. This restriction means that values must be either fully η-expanded or else are of an uninterpreted base type.

A similar approach is taken at the signal level. We do not wish to solve or even simplify the equations of a model, and hence, there is little computational behaviour to capture. However, as products were introduced as a way of abstracting over multiple interface variables we will perform the usual β-reductions and η-expansions associated with this type.

```
data SNrm (Γ : Ctx) : STmLike where
  tt    : SNrm Γ Φ unit
  fun   : Nrm  Γ   num → SNrm Γ Φ num
  binop : SNrm Γ Φ num → SNrm Γ Φ num → SNrm Γ Φ num
  pair  : SNrm Γ Φ σ1  → SNrm Γ Φ σ2  → SNrm Γ Φ (σ1 × σ2)
  neu   : SBase σ      → SNeu Γ Φ σ   → SNrm Γ Φ σ
```

```
data SNeu (Γ : Ctx) : STmLike where
  lvar : SNeu Γ Φ num
  svar : SVar   Φ σ             → SNeu Γ Φ σ
  fst  : SNeu Γ Φ (σ1 × σ2) → SNeu Γ Φ σ1
  snd  : SNeu Γ Φ (σ1 × σ2) → SNeu Γ Φ σ2
```

The representations of equations and switches do not need to be stratified as it is not possible to construct a redex with these datatypes alone. Therefore, QNrm and SwNrm follow the same structure as their term counterparts, but with the references to neutral and normal terms carefully chosen to prevent redexes appearing elsewhere. It is also worth noting that the reappearance of a length index on switching blocks will allow us to verify that the number of branches for a given block does not change during normalisation.

```
data QNrm (Γ : Ctx) (Φ : SCtx) : Set where
```

```
  empty  : QNrm Γ Φ
  _∧_    : QNrm Γ Φ        → QNrm   Γ Φ    → QNrm Γ Φ
  _◇_    : Neu   Γ (sr σ) → SNrm   Γ Φ σ  → QNrm Γ Φ
  _=_    : SNrm Γ Φ σ      → SNrm   Γ Φ σ  → QNrm Γ Φ
  switch : Fin n            → SwNrm Γ Φ n  → QNrm Γ Φ

data SwNrm (Γ : Ctx) (Φ : SCtx) : ℕ → Set where
  []     :   SwNrm Γ Φ 0
  branch :   SwNrm Γ Φ n → SNrm   Γ Φ num
           → QNrm   Γ Φ    → SwNrm Γ Φ (1 + n)
```

Just as with variables, it is possible to recover a term from a normal form representation. After all, the normal forms represent the image of the normalisation function, and thus, are necessarily a subset of the terms.

```
embNrm : Nrm Γ τ → Tm Γ τ
embNrm lit         = lit
embNrm (lam t)     = lam (embNrm t)
embNrm (sigrel q)  = sigrel (embQNrm q)
embNrm (neu b t)   = embNeu t
```

A map of the form embFoo exists for all Foo ∈ {Nrm, Neu, SNrm, SNeu, QNrm, SwNrm}. It is interesting to note that the embedding function is a section of the normalisation function (dually, normalisation is a retraction of embedding).

The idea of using a new datatype for normal forms that can be embedded back into the term representation is fairly common in the existing literature, particularly when attempted to formalise normalisation in type theory (Chapman [2009], Danielsson [2006], McBride [2000]). Partitioning the terms to prevent redexes from occurring is a standard technique, though there are novel aspects to our formalisation as we consider a language with multiple levels.

Context Morphisms and Weakening

All terms that are related to one another by the equational theory should be represented by the same object in the model. Therefore, there is a canonical object in the model for each element of the set of terms quotiented by convertibility. The canonical object represents terms that are $\beta\eta$-equivalent to one another by effectively representing the fully reduced and fully η-expanded term to which all other equivalent terms can be converted. Thus, when reifying an object into a normal form, we may need to build a subterm in a context extended by new variables. To see why this is the case consider η-expanding the term $\lambda\, f \,.\, f$ at the type $(\tau_1 \;\rightarrow\; \tau_2) \;\rightarrow\; (\tau_1 \;\rightarrow\; \tau_2)$, or equivalently, the $H_{[\![]\!]}$ term given below:

$$\begin{aligned} &\mathsf{etaFun} \;:\; \mathsf{Tm}\,\Gamma\;((\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)) \\ &\mathsf{etaFun} \;=\; \mathsf{lam\ var} \end{aligned}$$

The term is expanded to $\lambda\, f \,.\, \lambda\; x \,.\, f\;\; x$, which constructs the subterm $f\;\; x$ in a context extended by the new variable x. This behaviour is accounted for in the model using so-called *context morphisms*.

A context morphism is a preorder (a reflexive and transitive binary relation) specified by the type $\Gamma \sqsubseteq \Delta$ to denote a transformation from the context Γ to Δ. Context morphisms (from now on simply morphisms) are a general mechanism for weakening contexts and are closely related to renamings; specifically, a morphism allows variables to be renamed, reordered, or even added to a context. The symbol $\sqsubseteq$ is chosen to emphasise that the context on the right may be *weaker* than the context appearing on the left; that is, any term that is well-scoped in Γ is also well-scoped in Δ by weakening.

$$\begin{aligned} &\_\sqsubseteq\_ \;:\; \mathsf{Ctx} \;\rightarrow\; \mathsf{Ctx} \;\rightarrow\; \mathsf{Set} \\ &\Gamma \sqsubseteq \Delta \;=\; \Gamma \Rightarrow^{\mathsf{Var}} \Delta \end{aligned}$$

Thanks to our parameterised definition of substitutions, context morphisms can simply be represented as a substitution from variables to variables. This presentation is particularly convenient for this work as it integrates smoothly with the choice to use explicit substitutions in $H_{[\![]\!]}$. In particular, much of the machinery already defined for substitutions can be reused. For example, the above definition fulfils the requirements of a preorder due to id and $\circ$.

It is helpful to derive a number of definitions from the specification above. Of particular interest are morphisms that will be useful in later sections of this paper. Variables can be exchanged, implying that variables can be arbitrarily reordered. The relation is monotonic, that is, a morphism can be extended on both sides. There is also the lifting (lift) of renamings to substitutions:

$$\begin{aligned}
\mathsf{exch} &: (\Gamma \triangleright \tau_1 \triangleright \tau_2) \sqsubseteq (\Gamma \triangleright \tau_2 \triangleright \tau_1)\\
\mathsf{exch} &= (\mathsf{wkn} \circ \mathsf{wkn}) \triangleright \mathsf{vz} \triangleright \mathsf{vs}\ \mathsf{vz}
\end{aligned}$$

$$\begin{aligned}
\mathsf{mono} &: \Gamma \sqsubseteq \Delta \to (\Gamma \triangleright \tau) \sqsubseteq (\Delta \triangleright \tau)\\
\mathsf{mono}\ \gamma &= (\gamma \circ \mathsf{wkn}) \triangleright \mathsf{vz}
\end{aligned}$$

$$\begin{aligned}
\mathsf{lift} &: \Gamma \sqsubseteq \Delta \to \Gamma \Rightarrow^{\mathsf{Tm}} \Delta\\
\mathsf{lift} &= \mathsf{map}\ \mathsf{embVar}
\end{aligned}$$

The application of a context morphism to a context-index type is characterised by the type Weaken. It is straightforward to express how a type, such as Tm, can be rephrased into a weaker context. The payoff of representing morphisms as replacements is evident in the implementation of wknTm, which consists of simply lifting and then applying the morphism. Returning to the η-expansion of λ-abstractions, the subterm $f\ x$ now simply exists in the original context weakened by the morphism wkn.

$$\begin{aligned}
\mathsf{Weaken} &: (\mathsf{Ctx} \to \mathsf{Set}) \to \mathsf{Set}\\
\mathsf{Weaken}\ T &= \Gamma \sqsubseteq \Delta \to T\ \Gamma \to T\ \Delta
\end{aligned}$$

$$\begin{aligned}
\mathsf{wknTm} &: \mathsf{Weaken}\ (\mathsf{Tm}\ \cdot\ \tau)\\
\mathsf{wknTm}\ \gamma\ t &= [\mathsf{lift}\ \gamma]\ t
\end{aligned}$$

It would also be useful to express weakening of normal forms. Unfortunately, the explicit substitution constructor is not available to us for the Nrm family of types. This is for good reason: we want to be assured that all redexes have been eliminated from a normal form. However, it is possible to define weakening by recursion, for example, the weakening of variables and neutral terms is given below. We omit weakening definitions (of which there are many) and instead follow the naming scheme that wknFoo denotes the function that weakens the indexing context of the type Foo.

```
wknVar : Weaken (Var · τ)
wknVar id        v        = v
wknVar (γ ∘ δ) v          = wknVar δ (wknVar γ v)
wknVar (γ ▷ t) vz         = t
wknVar (γ ▷ t) (vs v)     = wknVar γ v
wknVar wkn       v        = vs v

wknNeu : Weaken (Neu · τ)
wknNeu γ (var s)     = var  (wknVar γ s)
wknNeu γ (app f x)   = app (wknNeu γ f) (wknNrm γ x)
```

Defining Models by Decoding Types

As promised at the start of this section, we can now present the maps that describe how a $H_{[\![]\!]}$ type, given in context, should be translated to the metalanguage. Given such a map f, a term of type τ in $H_{[\![]\!]}$ can then be interpreted as an inhabitant of the type $f\ \tau$. Viewing $H_{[\![]\!]}$ types as a universe, the following maps are the decoding functions that allow us to write generic definitions over objects of the model, of which interpretation and reification are arguably the most important in this chapter.

Base types, such as unit and num are to be left uninterpreted; that is, there is no computational behaviour associated with these types. Consequently, base types are mapped directly to normal forms as no reductions or expansions will need to be performed, effectively short-circuiting the process of normalisation.

Signal products are mapped to actual products of signal objects, defined recursively by the signal-level map SVal. This definition will enable the reduction of signal-level products that were introduced to allow multiple signal variables to be bound from a single sigrel, in the same fashion as FHM.

$$\begin{aligned}
&\mathsf{SVal} : \mathsf{Ctx} \to \mathsf{STmLike} \\
&\mathsf{SVal}\ \Gamma\ \Phi\ \mathsf{unit} &&= \mathsf{SNrm}\ \Gamma\ \Phi\ \mathsf{unit} \\
&\mathsf{SVal}\ \Gamma\ \Phi\ \mathsf{num} &&= \mathsf{SNrm}\ \Gamma\ \Phi\ \mathsf{num} \\
&\mathsf{SVal}\ \Gamma\ \Phi\ (\sigma_1 \times \sigma_2) &&= \mathsf{SVal}\ \ \Gamma\ \Phi\ \sigma_1 \times \mathsf{SVal}\ \Gamma\ \Phi\ \sigma_2
\end{aligned}$$

At the functional level, $H_{[\![]\!]}$ function spaces are mapped to actual function spaces with an additional context morphism. When reifying the model back into a normal form, the context morphism will be an essential mechanism for weakening contexts, which is needed to perform η-expansion.

To illustrate this point, consider a term f of type $\alpha \to \beta$ in a context Γ. When the object that represents f is reified back to a normal form it is η-expanded at least once to $\lambda\ x\ .\ f\ \ x$. The expanded term exists in the same context as the original term. However, the subterm $f\ x$ exists in the context Γ extended by x. In order to construct the application of f to x we must weaken the context of f. As the extent to which a term must be η-expanded is not known prior to normalisation (e.g. consider $f : (\alpha \to \beta) \to (\alpha \to \beta)$, which requires two steps of expansion, and thus appears in a doubly-weakened context), a context morphism is used to accumulate individual weakenings.

$$\begin{aligned}
&\mathsf{Val} : \mathsf{TmLike} \\
&\mathsf{Val}\ \Gamma\ \mathsf{num} &&= \mathsf{Nrm}\ \Gamma\ \mathsf{num} \\
&\mathsf{Val}\ \Gamma\ (\tau_1 \to \tau_2) &&= \Gamma \sqsubseteq \Delta \to \mathsf{Val}\ \ \ \Delta\ \tau_1 \ \ \ \to \mathsf{Val}\ \ \ \Delta\ \tau_2 \\
&\mathsf{Val}\ \Gamma\ (\mathsf{sr}\ \sigma) &&= \Gamma \sqsubseteq \Delta \to \mathsf{SVal}\ \Delta\ \Phi\ \sigma \to \mathsf{QVal}\ \Delta\ \Phi
\end{aligned}$$

For the remaining syntactic categories – equations and switching blocks – there is no computational behaviour to account for, and thus, they remain effectively uninterpreted. In Chap. 5.5 we will see that there are some nontrivial equivalences between equations that we might want to capture that will be manifest in a revised definition of the model of equations. However, in the interest of keeping the initial presentation as simple and clear as possible, we will ignore these non-essential aspects in this section.

```
QVal : Ctx → SCtx → Set
QVal Γ Φ = QNrm Γ Φ

SwVal : Ctx → SCtx → Set
SwVal Γ Φ = SwNrm Γ Φ
```

The objects of the model, as defined by the Val family of decoding functions, are parameterised on a context so that they can decode the types of *open terms*. When we define the interpreter we will need a way to find the meaning of free variables appearing in an open term. This is achieved by using an *environment*.

In its simplest form, an environment is a injection from variables to model objects. As variables are denoted by de Bruijn indices, and taking inspiration from the definition of contexts, an obvious implementation is a list of objects that correlates with the current context. This correlation can be made precise by indexing an environment on the context, ensuring that the environment is the correct length, and that the objects are of the correct type:

```
data EnvBad : Ctx → Set where
  ○   : EnvBad ○
  _▷_ : EnvBad Γ → Val Γ τ → EnvBad (Γ ▷ τ)
```

Unfortunately, this definition is difficult to use in practice. Each object of the environment exists in a different context, meaning the objects will often need to be explicitly weakened when being used. Furthermore, weakening a context with a morphism would require any environment indexed on this context to be weakened as well, which in general is impossible using the above definition. For example, consider applying the morphism exch.

The solution used in this thesis is to add an additional context parameter that will serve as a context for *all* the objects in the environment. Thus, when passing under a binder the new context parameter can be weakened accordingly, weakening each object in the environment as a consequence. This new context parameter does not interfere with the previous context index, which can continue to track the length and types of the objects independently. A notion of environments is needed at both the functional level and signal level.

```
data Env (Δ : Ctx) : Ctx → Set where
  ○   : Env Δ ○
  _▷_ : Env Δ Γ → Val Δ τ → Env Δ (Γ ▷ τ)

data SEnv (Δ : Ctx) (Ψ : SCtx) : SCtx → Set where
  ▷ : SVal Δ Ψ σ → SEnv Δ Ψ (▷ σ)
```

Finally, we can package up all of the components defined so far to give a model for each syntactic category of $H_{[\![]\!]}$. Note that the functional-level environment is needed at every level of the language due to the fun constructor, which embeds functional terms as time-invariant entities at the signal level.

```
Model : TmLike
Model Γ τ = Env Δ Γ → Val Δ τ

SModel : Ctx → STmLike
SModel Γ Φ σ = Env Δ Γ → SEnv Δ Ψ Φ → SVal Δ Ψ σ

QModel : Ctx → SCtx → Set
QModel Γ Φ = Env Δ Γ → SEnv Δ Ψ Φ → QVal Δ Ψ

SwModel : Ctx → SCtx → Set
SwModel Γ Φ = Env Δ Γ → SEnv Δ Ψ Φ → SwVal Δ Ψ
```

In summary, a model is created for each syntactic category. Each model behaves as an environment machine: it accepts a number of environments and produces a value. The environments that a particular model depend upon are determined by the kinds of variables that may occur bound in an expression for its syntactic category. For example, both functional variables and signal variables may occur in the syntax of a signal expression. As a result, the model of a signal expression requires both environments.

5.2.3 Normalisation

We have yet to define both interpretation and reification, the two main ingredients of Normalisation by Evaluation. Instead, thus far, we have focused on carefully defining the terms and the model. However, much of the hard work has now been done, and defining the two functions is straightforward. This is due, at least in part, to the approach of building data that is correct-by-construction; that is, encoding invariants (where feasible) directly into the definition of a datatype. The invariants put Agda's constraint solving mechanisms to work, allowing many impossible cases to be ruled out automatically; for example, eliminating ill-typed function applications.

We now show how terms can be interpreted into the model and how a normal form can be recovered from the model, thus achieving the first goal of this chapter: to transform a *closed* modular systems of equations into a flat set of equations via a process that is intimately related to the denotational semantics of the language. The requirement that the system of equations be closed means that the top-level definition will have the type Tm ∘ (sr unit), which in turn means that there will be no free functional variables and only one trivial (of type unit) interface signal variable. Consequently, we guarantee that there will be no redexes to impede the reduction of the modular system to a completely flat system of equations.

Interpretation

For each syntactic category an interpreter is defined that translates terms into objects of the appropriate model. Interpretation is implemented as a simple environment machine: when passing under a binder — be it a lam or a sigrel — the value that is bound by a variable is added to the environment. Values in the environment can later be recalled when they are required to be substituted for occurrences of the variables to which they refer.

An explicit substitution must also be interpreted explicitly, which we describe as a function on environments. As the application of a substitution exists in a potentially weaker context than the term being closed over, we need to be able to strengthen the environment again by discarding those variables from the environment that we know are not needed. The substitution encodes the relationship between the enclosed term's environment and the new weaker environment. Thus, the transformation of the environment proceed by examining the substitution to construct a new environment that excludes those variables that are not required. Further details are deferred until later in this section.

The interpreters are defined by induction on the terms. In each case, the meaning of a term is given independently, based solely on the meaning of any subterms. Put simply, the interpretation is compositional.

At the functional level there are a number of interesting cases to consider. Variables are particularly easy to interpret as $H_{[\![]\!]}$ only gives an explicit representation of the first variable, which is simply the value stored at the end of the environment. Application proceeds by first interpreting the object function into an actual function, which is then applied to the identity morphism and interpreted argument. Recall that while the trivial identity morphism is used during interpretation, a non-identity morphism will be required during reification. Similarly, λ-abstractions are interpreted as actual meta-level abstractions, extending the environment with the newly bound variable and weakening the environment with respect to the new morphism. Recall that the body of an abstraction exists in a different context to that of the abstraction itself (i.e. t exist in the same context as lam t, except that it has been extended by a new variable). It is for this reason that, in addition to the environment e, interpretation of an abstraction also requires a morphism γ that can be used to perform the appropriate transformation of the environment, allowing it to be used in the interpretation of t. As meta-level functions were also chosen to represent signal relations their interpretation is also meta-level abstraction, only this time the signal environment is extended instead.

$$\begin{aligned}
&\llbracket\_\rrbracket \;:\; \mathsf{Tm}\,\Gamma\,\tau \;\to\; \mathsf{Model}\,\Gamma\,\tau\\
&\llbracket\, \mathsf{lit} \,\rrbracket\;\; e &&= \mathsf{lit}\\
&\llbracket\, \mathsf{var} \,\rrbracket\;\; (\rho \triangleright v) &&= v\\
&\llbracket\, \mathsf{app}\; f\; x \,\rrbracket\;\; e &&= \llbracket\, f \,\rrbracket\; e\; \mathsf{id}\; (\llbracket\, x \,\rrbracket\; e)\\
&\llbracket\, \mathsf{lam}\; t \,\rrbracket\;\; e &&= \lambda\, \gamma\; v \;\to\; \llbracket\, t \,\rrbracket\;\; (\mathsf{wknEnv}\; \gamma\; e \triangleright v)\\
&\llbracket\, \mathsf{sigrel}\; q \,\rrbracket\;\; e &&= \lambda\, \gamma\; v \;\to\; \llbracket\, q \,\rrbracket_{\mathsf{Q}}\; (\mathsf{wknEnv}\; \gamma\; e)\; (\triangleright\, v)\\
&\llbracket\, [\gamma]\; t \,\rrbracket\;\; e &&= \llbracket\, t \,\rrbracket\; (\llbracket\, \gamma \,\rrbracket_{\Rightarrow}\; e)
\end{aligned}$$

Signal-level interpretation is primarily structural, with the only interesting computation occurring for products. The constructors and destructors for products are interpreted as one would expect, with π_1 and π_2 providing the meta-level projections and $\_,\_$ denoting the meta-level constructor. With only one bound signal variable in scope at any given time, svar is interpreted as the contents of the signal-level environment.

$$\begin{aligned}
&\llbracket\_\rrbracket_{\mathsf{S}} \;:\; \mathsf{STm}\,\Gamma\,\Phi\,\sigma \;\to\; \mathsf{SModel}\,\Gamma\,\Phi\,\sigma\\
&\llbracket\, \mathsf{tt} \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \mathsf{tt}\\
&\llbracket\, \mathsf{svar} \,\rrbracket_{\mathsf{S}}\; e\; (\triangleright\, v) &&= v\\
&\llbracket\, \mathsf{lvar} \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \mathsf{neu}\; \sigma\; \mathsf{lvar}\\
&\llbracket\, \mathsf{fun}\; t \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \mathsf{fun}\; (\llbracket\, t \,\rrbracket\; e)\\
&\llbracket\, \mathsf{binop}\; a\; b \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \mathsf{binop}\; (\llbracket\, a \,\rrbracket_{\mathsf{S}}\; e\; g)\; (\llbracket\, b \,\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\, \mathsf{fst}\; s \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \pi_1\; (\llbracket\, s \,\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\, \mathsf{snd}\; s \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \pi_2\; (\llbracket\, s \,\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\, \mathsf{pair}\; a\; b \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= (\llbracket\, a \,\rrbracket_{\mathsf{S}}\; e\; g), (\llbracket\, b \,\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\, [\, \gamma \,]\; s \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \llbracket\, s \,\rrbracket_{\mathsf{S}}\; (\llbracket\, \gamma \,\rrbracket_{\Rightarrow}\; e)\; g\\
&\llbracket\, \langle\, \phi \,\rangle\; s \,\rrbracket_{\mathsf{S}}\; e\;\; g &&= \llbracket\, s \,\rrbracket_{\mathsf{S}}\; e\; (\llbracket\, \phi \,\rrbracket^{\mathsf{S}}_{\Rightarrow}\; e\; g)
\end{aligned}$$

The application of a signal relation is interpreted in the same way as its functional counterpart, unsurprising given the similarity of their representations in the model. Atomic equations make use of $\mathsf{reify}_{\mathsf{S}}$, which will be defined later in the section, to convert signal values back to normal forms. This forward dependency could have been avoided if equation values were instead redefined using atomic equations of signal *values*. However, in this thesis the benefit of redefining equations is strongly outweighed by the added complexity and noise such a definition would cause. The remaining signal-level constructors along with switching blocks simply preserve the structure of the equation.

$$
\begin{aligned}
&\llbracket\_\rrbracket_{\mathsf{Q}} \;:\; \mathsf{QTm}\;\Gamma\;\Phi \;\to\; \mathsf{QModel}\;\Gamma\;\Phi\\
&\llbracket\; \mathsf{empty} \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \mathsf{empty}\\
&\llbracket\; q_1 \wedge q_2 \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \llbracket\; q_1 \;\rrbracket_{\mathsf{Q}}\; e\; g \wedge \llbracket\; q_2 \;\rrbracket_{\mathsf{Q}}\; e\; g\\
&\llbracket\; t \;\diamond\; s \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \llbracket\; t \;\rrbracket\;\; e\; \mathsf{id}\; (\llbracket\; s \;\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\; s_1 = s_2 \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \mathsf{reify}_{\mathsf{S}}\; \sigma\; (\llbracket\; s_1 \;\rrbracket_{\mathsf{S}}\; e\; g) = \mathsf{reify}_{\mathsf{S}}\; \sigma\; (\llbracket\; s_2 \;\rrbracket_{\mathsf{S}}\; e\; g)\\
&\llbracket\; \mathsf{switch}\; b\; sw \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \mathsf{switch}\; b\; (\llbracket\; sw \;\rrbracket_{\mathsf{Sw}}\; e\; g)\\
&\llbracket\; [\;\gamma\;]\; q \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \llbracket\; q \;\rrbracket_{\mathsf{Q}}\; (\llbracket\; \gamma \;\rrbracket_{\Rightarrow}\; e)\; g\\
&\llbracket\; \langle\; \phi\; \rangle\; q \;\rrbracket_{\mathsf{Q}}\; e\; g \;=\; \llbracket\; q \;\rrbracket_{\mathsf{Q}}\; e\; (\llbracket\; \phi \;\rrbracket^{\mathsf{S}}_{\Rightarrow}\; e\; g)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket\_\rrbracket_{\mathsf{Sw}} \;:\; \mathsf{Switch}\;\Gamma\;\Phi \;\to\; \mathsf{SwModel}\;\Gamma\;\Phi\\
&\llbracket\; [] \;\rrbracket_{\mathsf{Sw}}\; e\; g \;=\; []\\
&\llbracket\; \mathsf{branch}\; sw\; s\; q \;\rrbracket_{\mathsf{Sw}}\; e\; g \;=\; \mathsf{branch}\; (\llbracket\; sw \;\rrbracket_{\mathsf{Sw}}\; e\; g)\; (\llbracket\; s \;\rrbracket_{\mathsf{S}}\; e\; g)\; (\llbracket\; q \;\rrbracket_{\mathsf{Q}}\; e\; g)\\
&\llbracket\; [\;\gamma\;]\; sw \;\rrbracket_{\mathsf{Sw}}\; e\; g \;=\; \llbracket\; sw \;\rrbracket_{\mathsf{Sw}}\; (\llbracket\; \gamma \;\rrbracket_{\Rightarrow}\; e)\; g\\
&\llbracket\; \langle\; \phi\; \rangle\; sw \;\rrbracket_{\mathsf{Sw}}\; e\; g \;=\; \llbracket\; sw \;\rrbracket_{\mathsf{Sw}}\; e\; (\llbracket\; \phi \;\rrbracket^{\mathsf{S}}_{\Rightarrow}\; e\; g)
\end{aligned}
$$

Interpreting explicit substitutions is needed for transforming an environment when normalising a closure. The process is best understood via an example. Given a term t in the context Γ and a substitution ϕ from Γ to Δ, the applied closure $[\phi]\; t$ and corresponding environment ρ exist in the context Δ. If we wish to interpret t, then we must construct an environment in the context Γ. Initially, this seems like it may be problematic as Δ is weaker than Γ. However, as a substitution describes how the two contexts are related, we can use it to reorganise an environment, discarding any values that are not of use.

$$
\begin{aligned}
&\llbracket\_\rrbracket_{\Rightarrow} \;:\; \Delta \Rightarrow E \;\to\; \mathsf{Env}\;\Gamma\;E \;\to\; \mathsf{Env}\;\Gamma\;\Delta\\
&\llbracket\; \mathsf{id} \;\rrbracket_{\Rightarrow}\; e \;=\; e\\
&\llbracket\; \gamma \circ \delta \;\rrbracket_{\Rightarrow}\; e \;=\; \llbracket\; \gamma \;\rrbracket_{\Rightarrow}\; (\llbracket\; \delta \;\rrbracket_{\Rightarrow}\; e)
\end{aligned}
$$

$$\begin{aligned}
[\![\, \gamma \rhd t \,]\!]_{\Rightarrow}\; e &= [\![\, \gamma \,]\!]_{\Rightarrow}\; e \rhd [\![\, t \,]\!]\; e \\
[\![\, \mathsf{wkn} \,]\!]_{\Rightarrow}\; (e \rhd v) &= e
\end{aligned}$$

Note that at the signal level a functional environment is required for the mutually recursive calls in the case for extension.

$$\begin{aligned}
&[\![\_]\!]^{\mathsf{S}}_{\Rightarrow} : \Psi \Rightarrow_{\mathsf{S}} P \to \mathsf{Env}\, \Gamma\, \Delta \to \mathsf{SEnv}\, \Gamma\, \Phi\, P \to \mathsf{SEnv}\, \Delta\, \Phi\, \Psi \\
&[\![\, \mathsf{id} \,]\!]^{\mathsf{S}}_{\Rightarrow}\; e\; g = g \\
&[\![\, \phi \circ \psi \,]\!]^{\mathsf{S}}_{\Rightarrow}\; e\; g = [\![\, \phi \,]\!]^{\mathsf{S}}_{\Rightarrow}\; e\; ([\![\, \psi \,]\!]^{\mathsf{S}}_{\Rightarrow}\; e\; g) \\
&[\![\, \rhd \phi \,]\!]^{\mathsf{S}}_{\Rightarrow}\; e\; g = \rhd ([\![\, \phi \,]\!]_{\mathsf{S}}\; e\; g)
\end{aligned}$$

Given the above definitions of interpretation, it may not be clear why explicit substitutions are at all useful or why context morphisms are included in the model. In particular, it would have been much simpler to define interpretation with an implicit notion of substitution. However, as we will discover in the following sections, the benefits of explicit substitutions are reaped when formalising the equations of convertibility and when proving correctness properties. While context morphisms serve no obvious purpose during interpretation, the reader is reminded that they will be essential during reification.

Our interpreter is similar to existing approaches in earlier literature. With the exception of context morphisms, our interpreter takes roughly the same approach as the earliest work on Normalisation by Evaluation by Berger and Schwichtenberg [1991]. The morphisms are needed in our work as part of the framework for dealing with fresh variable generation and to avoid variable name clashes. Berger et al. assume the existence of a function “gensym” for generating fresh names, whereas we use de Bruijn indices, well-typed and well-scoped contexts, and context morphisms to handle weakening.

The approach taken by Dybjer and Filinski [2000] solves the problem of name generation in a very similar way to our own approach, though their formalisation is not implemented in type theory and thus they leave implicit a number of issues that we must deal with explicitly (e.g. weakening). They define evaluation and reification with respect to a family of terms indexed on the set of variable names.

This is the same idea as indexing terms on the context as in our formalisation.

Reification

If the model represents the denotation of $H_{[\![]\!]}$ then the interpreter defined in the previous subsection provides half of the correspondence between the two; the model is sound as all well-typed programs can be represented in the model. Thus, all that remains is to give the remaining half of the correspondence; the model is complete as each object of the model is associated with a well-typed program (specifically, a normal form). This mapping is often called *reification* and is manifested by a function taking objects of the model back to terms, or in our case, normal forms. The connotations of the terminology are intentional: the model is a very abstract representation of the language and reification aims to make this representation more concrete.

Reification is type directed and compositional. Reification is simply the identity at base types as they are mapped to normal forms in the model. As reification is responsible for η-expansion, it is not surprising that no operations are performed on types that are intentionally left uninterpreted.

We also define the reflect functions, which are the conceptual *dual* of reification. Reflection takes neutral terms to values, and as we shall see shortly, they will be used to construct values from variable indices of arbitrary types.

$$
\begin{array}{lll}
\mathsf{reify_S} : (\sigma : \mathsf{SType}) \to \mathsf{SVal}\,\Gamma\,\Phi\,\sigma \to \mathsf{SNrm}\,\Gamma\,\Phi\,\sigma & & \\
\mathsf{reify_S}\ \mathsf{unit}\quad n & = & n \\
\mathsf{reify_S}\ \mathsf{num}\quad n & = & n \\
\mathsf{reify_S}\ (\sigma_1 \times \sigma_2)\ (a, b) & = & \mathsf{pair}\ (\mathsf{reify_S}\ \sigma_1\ a)\ (\mathsf{reify_S}\ \sigma_2\ b)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{reflect_S} : (\sigma : \mathsf{SType}) \to \mathsf{SNeu}\,\Gamma\,\Phi\,\sigma \to \mathsf{SVal}\,\Gamma\,\Phi\,\sigma & & \\
\mathsf{reflect_S}\ \mathsf{unit}\quad n & = & \mathsf{neu}\ \mathsf{unit}\ n \\
\mathsf{reflect_S}\ \mathsf{num}\quad n & = & \mathsf{neu}\ \mathsf{num}\ n \\
\mathsf{reflect_S}\ (\sigma_1 \times \sigma_2)\ n & = & (\mathsf{reflect_S}\ \sigma_1\ (\mathsf{fst}\ n)), (\mathsf{reflect_S}\ \sigma_2\ (\mathsf{snd}\ n))
\end{array}
$$

Signal-level products are the only interesting case at the signal level with the usual η-expansion rule for products being used to expand values. We follow the same discipline at the functional level, using the η-rules for function spaces and signal relations to build normal forms. Notice that our context morphisms are finally of use, allowing the application of function values to occur in a weakened context when η-expanding.

$$\begin{aligned}
&\mathsf{reify} \;:\; (\tau \;:\; \mathsf{Type}) \;\rightarrow\; \mathsf{Val}\,\Gamma\,\tau \;\rightarrow\; \mathsf{Nrm}\,\Gamma\,\tau\\
&\mathsf{reify}\;\mathsf{num} \qquad\quad n \;=\; n\\
&\mathsf{reify}\;(\tau_1 \rightarrow \tau_2)\;n \;=\; \mathsf{lam}\;(\mathsf{reify}\;\tau_2\;(n\;\mathsf{wkn}\;(\mathsf{reflect}\;\tau_1\;(\mathsf{var}\;\mathsf{vz}))))\\
&\mathsf{reify}\;(\mathsf{sr}\;\sigma) \qquad n \;=\; \mathsf{sigrel}\;(n\;\mathsf{id}\;(\mathsf{reflect}_{\mathsf{S}}\;\sigma\;(\mathsf{var}\;\mathsf{vz})))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{reflect} \;:\; (\tau \;:\; \mathsf{Type}) \;\rightarrow\; \mathsf{Neu}\,\Gamma\,\tau \;\rightarrow\; \mathsf{Val}\,\Gamma\,\tau\\
&\mathsf{reflect}\;\mathsf{num} \qquad\quad n \;=\; \mathsf{neu}\;\mathsf{num}\;n\\
&\mathsf{reflect}\;(\tau_1 \rightarrow \tau_2)\;n \;=\; \lambda\,\gamma\,x \;\rightarrow\; \mathsf{reflect}\;\tau_2\;(\mathsf{app}\;(\mathsf{wknNeu}\;\gamma\;n)\;(\mathsf{reify}\;\tau_1\;x))\\
&\mathsf{reflect}\;(\mathsf{sr}\;\sigma) \qquad n \;=\; \lambda\,\gamma\,x \;\rightarrow\; \mathsf{wknNeu}\;\gamma\;n \diamond \mathsf{reify}_{\mathsf{S}}\;\sigma\;x
\end{aligned}$$

All that remains is to put together the pieces to produce the full normalisation procedure. The symbol idEnv denotes the identity environment, which maps variables back to themselves.

$$\begin{aligned}
&\mathsf{nbe} \;:\; \mathsf{Tm}\,\Gamma\,\tau \;\rightarrow\; \mathsf{Tm}\,\Gamma\,\tau\\
&\mathsf{nbe}\;t \;=\; \mathsf{embNrm}\;(\mathsf{reify}\;([\![\, t \,]\!]\;\mathsf{idEnv}))
\end{aligned}$$

5.3 Correctness and Other Properties

Metatheoretical results are often a driving force in programming language formalisation. Demonstrating that a language and its semantics are well-behaved allows the user to safely reason about programs. Type systems are a prolific approach to specifying what it means for a program to be correct, and in conjunction with a formal semantics, provide a tractable method for *proving* the absence of certain program behaviours, and show that these proofs are consistent with evaluation.

In this section a number of desirable properties inherent to the formalisation of $H_{[\![]\!]}$ are discussed. A large portion of the section, and a major contribution of this chapter, is dedicated to demonstrating how the types used in the formalisation can be refined to help prove *normalisation*, a key correctness theorem.

5.3.1 Correct by Construction

An important property of the formalisation is *termination* and *totality*. It is guaranteed that normalisation will terminate for all input terms. The iteration of normalisation steps is also guaranteed to be productive. In both instances, these properties are provided by Agda's support for automatic termination and productivity checking. Taken together, one can state that while a program may be simulated indefinitely (which may well be the intended behaviour, e.g. modelling Brownian motion with events occurring at particle collisions to compute new trajectories), the process never gets *stuck* attempting to compute a *new* generation of equations during simulation.

The normalisation procedure is guaranteed to produce canonical normal forms, which arises by construction as it is impossible to construct a normal form that violates these properties. Combined with termination, the result is the strong property that normalisation always finds a normal form for all input terms. Later in this section we will strengthen this claim by showing the correctness of normalisation: we prove that a normal form is reached only via a sequence of zero or more applications of rules from the equational theory.

The type of normalisation ($\mathsf{Tm}\ \Gamma\ \tau\ \rightarrow\ \mathsf{Tm}\ \Gamma\ \tau$) witnesses the fact that the normalisation procedure is type preserving. Only well-typed terms are accepted for normalisation (indeed, only well-typed terms may even be constructed), with the type of each step of normalisation (e.g. reification) enforcing preservation.

Finally, compared to a more direct approach of a shallow embedding, our approach is relatively efficient. Furthermore, through standard optimisation techniques, NbE can be made to rival the efficiency of the underlying metalanguage [Boespflug, 2009]. While the metalanguage currently used is not particularly efficient, one could extract or transcribe the procedure into a more efficient functional language, such as Haskell.

5.3.2 Convertibility

In Sect. 5.1.4 we discussed what it meant for Normalisation by Evaluation to be correct. An important component of correctness that is particularly relevant to our existing metatheoretical properties is theorem 5.3.

$$\mathsf{nbe}\ t \simeq t \tag{5.3}$$

Working toward a proof of theorem 5.3 for $H_{[\![]\!]}$ we are first obliged to specify the convertibility relations ($\simeq$), often called the *equational theory*, that will be used to characterise the behaviour of the normalisation function. Rather than providing the reader with every axiom of all the relations involved – which would be very tedious and repetitive – we outline only the different types of rules that are present for each equivalence, giving concrete examples taken from the relation for functional terms.

Equivalence

All the relations are *equivalences*: they are reflexive, symmetric, and transitive (Fig. 5.2). These axioms are not strictly required as they could be derived from the definition of each relation directly (see Danielsson [2006]). However, such derivations can be involved and time consuming, and given that we do not intend to match on a proof of convertibility, it serves just as well to postulate these additional rules. This is the same approach as taken by Chapman [2009].

$$\frac{}{t \simeq t}\ (\textsc{Refl}) \qquad \frac{t_1 \simeq t_2}{t_2 \simeq t_1}\ (\textsc{Symm}) \qquad \frac{t_1 \simeq t_2 \qquad t_2 \simeq t_3}{t_1 \simeq t_3}\ (\textsc{Tran})$$

Figure 5.2: Equivalence rules.

Congruence

The relations are also the obvious *congruences*. Figure 5.3 gives the congruence rules for functional terms.

$$\frac{f_1 \simeq f_2 \qquad x_1 \simeq x_2}{\mathsf{app}\ f_1\ x_1 \simeq \mathsf{app}\ f_2\ x_2}\ (\textsc{Cong}_{app}) \qquad \frac{t_1 \simeq t_2}{\mathsf{lam}\ t_1 \simeq \mathsf{lam}\ t_2}\ (\textsc{Cong}_{lam})$$

$$\frac{q_1 \simeq q_2}{\mathsf{sigrel}\ q_1 \simeq \mathsf{sigrel}\ q_2}\ (\textsc{Cong}_{sr}) \qquad \frac{\gamma_1 \simeq \gamma_2 \qquad t_1 \simeq t_2}{[\gamma_1]\ t_1 \simeq [\gamma_2]\ t_2}\ (\textsc{Cong}_{[]})$$

Figure 5.3: Congruence rules for functional terms.

Substitution Application

Due to the decision to use a calculus of explicit substitutions, the interaction between terms and substitutions must be given by the equational theory. The interaction axioms (Fig. 5.4) are inspired by Chapman [2009], and while not minimal, the axioms turn out to be very convenient for our purposes.

The Iden, Comp, and Proj rules describe how to eliminate an explicit substitution. The sub family of rules allow a substitution to move under a constructor. The rule $\textsc{Sub}_{lam}$ is noteworthy as the substitution is both weakened and extended. The extension by var means that the new variable is substituted for itself, thus rendering the substitution inert for this newly bound variable.

Similar rules also exist at the signal level for signal-level terms and substitutions. By comparison, in a language of implicit substitutions, the behaviour of applying a substitution would instead be given by a function.

$$\frac{}{[\mathsf{id}]\ t \simeq t}\ (\textsc{Iden}) \qquad \frac{}{\mathsf{lam}\ ([\gamma \circ \mathsf{wkn} \rhd \mathsf{var}]\ t) \simeq [\gamma]\ (\mathsf{lam}\ t)}\ (\textsc{Sub}_{lam})$$

$$\frac{}{[\gamma_1 \circ \gamma_2]\ t \simeq [\gamma_2]\ [\gamma_1]\ t}\ (\textsc{Comp}) \qquad \frac{}{\mathsf{sigrel}\ ([\gamma]\ q) \simeq [\gamma]\ (\mathsf{sigrel}\ q)}\ (\textsc{Sub}_{sr})$$

$$\frac{}{t \simeq [\gamma \rhd t]\ \mathsf{var}}\ (\textsc{Proj}) \qquad \frac{}{\mathsf{app}\ ([\gamma]\ f)\ ([\gamma]\ x) \simeq [\gamma]\ (\mathsf{app}\ f\ x)}\ (\textsc{Sub}_{app})$$

Figure 5.4: Substitution interaction rules for functional terms.

Computation

Arguably the most interesting aspect of the equational theory is the set of computational rules describing the valid β and η conversions (see Fig. 5.5). Only a handful of the rules express computational behaviour, the remainder of the rules, of which there are many, are relatively mundane by comparison.

$$\frac{}{\langle \rhd s \rangle\ q \simeq \mathsf{sigrel}\ q \diamond s}\ (\beta\text{-SR}) \qquad \frac{}{[\mathsf{id} \rhd t_1]\ t_2 \simeq \mathsf{app}\ (\mathsf{lam}\ t_2)\ t_1}\ (\beta\text{-}\textsc{Lam})$$

$$\frac{}{\mathsf{sigrel}\ (t \diamond \mathsf{svar}) \simeq t}\ (\eta\text{-SR}) \qquad \frac{}{\mathsf{lam}\ (\mathsf{app}\ ([\mathsf{wkn}]\ t)\ \mathsf{var}) \simeq t}\ (\eta\text{-}\textsc{Lam})$$

$$\frac{}{s_1 \simeq \mathsf{fst}\ (\mathsf{pair}\ s_1\ s_2)}\ (\beta\text{-}\textsc{Fst}) \qquad \frac{}{s_2 \simeq \mathsf{snd}\ (\mathsf{pair}\ s_1\ s_2)}\ (\beta\text{-}\textsc{Snd})$$

$$\frac{}{\mathsf{pair}\ (\mathsf{fst}\ s)\ (\mathsf{snd}\ s) \simeq s}\ (\eta\text{-}\textsc{Prod})$$

Figure 5.5: Computation rules.

Substitution Equivalence

Finally, a number of equations are given to describe the equivalence of substitutions (Fig. 5.6). The Assoc, IdL, and IdR rules are the normal categorical laws of associativity, left identity, and right identity, respectively. Wkn, Shift, and ExtId are a number of convenient rules for manipulating substitutions. Shift is particularly interesting as it allows us to commute extension ($\rhd$) across composition ($\circ$). As before, a similar set of rules exist for the signal level.

Agda is currently much more successful at unifying indices if they are in a

$$\frac{}{(\gamma_1 \triangleright t) \circ \gamma_2 \simeq \gamma_1 \circ \gamma_2 \triangleright [\gamma_2]\ t}\ (\textsc{Shift}) \qquad \frac{}{\mathsf{wkn} \circ (\gamma \triangleright t) \simeq \gamma}\ (\textsc{Wkn})$$

$$\frac{}{\gamma_1 \circ (\gamma_2 \circ \gamma_3) \simeq (\gamma_1 \circ \gamma_2) \circ \gamma_3}\ (\textsc{Assoc}) \qquad \frac{}{\mathsf{id} \circ \gamma \simeq \gamma}\ (\textsc{IdL})$$

$$\frac{}{\mathsf{id} \simeq (\mathsf{id} \circ \mathsf{wkn}) \triangleright \mathsf{var}}\ (\textsc{ExtId}) \qquad \frac{}{\gamma \circ \mathsf{id} \simeq \gamma}\ (\textsc{IdR})$$

Figure 5.6: Substitution rules.

constructor-headed form (i.e. syntax directed). Thanks to the use of explicit substitutions, all of the rules in the equational theory are syntax directed making them much easier to work with in Agda.

5.3.3 Indexing and Reindexing

The principal idea behind the normalisation proof is to use a technique due to Danielsson [2006] that refines the types of normal forms and values. The technique involves adding a *term index* to said types with the invariant that the types be "equationally related" to the index. A *reindexing* operation allows the index term of a value or normal form to be changed provided that the new term index is convertible to the previous. A proof of convertibility is stored as a witness to this requirement. Thus, as normalisation proceeds it is required to reindex values and normal forms each time a rule of the equational theory is applied. In effect, the normalisation of each term is associated with a proof tree that demonstrates how the input term can be converted to the output term.

The indexed variants of Nrm and Neu are given below to demonstrate the technique. The indexing of normal forms is very regular and predictable. At this point, the reader can refer back to Sect. 5.2.2 for a comparison with the original definitions. The indexing is mostly structural; the cases for lit, lam, neu, sigrel, and app simply wrap up their argument indices with the appropriate constructor. Variables (var) use embVar to embed the given variable into a term.

```
data Nrm : Tm Γ τ → Set where
```

```
  lit    : Nrm lit
  lam    : Nrm t    → Nrm (lam t)
  sigrel : QNrm q   → Nrm (sigrel q)
  neu    : Base τ   → Neu t   → Nrm t
  _::_   : t₁ ≃ t₂  → Nrm t₁  → Nrm t₂

data Neu : Tm Γ τ → Set where
  var  : (v : Var Γ τ) → Neu (embVar v)
  app  : Neu t₁         → Nrm t₂ → Neu (app t₁ t₂)
  _::_ : t₁ ≃ t₂        → Neu t₁ → Neu t₂
```

The new constructor :: has also been added to both datatypes, allowing the index of a normal form to be *cast* to another term, provided that the new index and old index are convertible. Another perspective on the new index is to treat normal forms as a *view* of terms constructed using nbe. Thus, a normal form is a canonical view of a term, which may be cast to represent the view of any other convertible term.

With the indexed variant of normal forms in hand, the modification to values (Val) to include a term index is possible. For base types, a value is a canonical view of the term. For function spaces and signal relations, the context morphism is used to weaken the indexing term before it is applied. Values for signals, equations, and events are also modified in a similar manner.

```
Val : Tm Γ τ → Set
Val {num}     t = Nrm t
Val {sr σ}    t = (γ : Γ ⊑ Δ) → SVal s → QVal (wknTm γ t ⋄ s)
Val {σ → τ}  f = (γ : Γ ⊑ Δ) → Val  x → Val (app (wknTm γ f) x)
```

Just as a cast constructor (::) is required to reindex a normal form, a cast function can be derived for values. The cast function is defined using the same approach as values by discriminating against the type of the indexing term.

The next step is to define an indexed model. A crucial component of the model are the environments. The functional and signal environments are redefined with substitution and signal-level substitution indices, respectively. The choice of a substitution index is motivated by the need to keep track of the relationship between the two indexing contexts.

```
data Env : Δ ⇒ Γ → Set where
  ○    : Env empty
  _▷_  : Env γ     → Val t   → Env (γ ▷ t)
  _::_ : γ₁ ≃ γ₂  → Env γ₁  → Env γ₂
```

The refined model is given below. The family of functions swknFoo weaken the *signal context* of the type Foo, where Foo $\in$ {QTm, STm, Switch}. For example, the signature defined by SModel should be read: given an environment indexed on a substitution γ and a signal environment indexed on a signal substitution ϕ, construct a signal value indexed on the input signal term s, but first weaken s by γ and then ϕ. The choice of weakening order is arbitrary, it would have worked equally well to first weaken by the signal substitution.

```
Model : Tm Γ τ → Set
Model t = Env γ → Val (wknTm γ t)

QModel : QTm Γ Φ → Set
QModel q = Env γ → SEnv ϕ
  → QNrm (swknQTm ϕ (wknQTm γ q))

SModel : STm Γ Φ σ → Set
SModel s = Env γ → SEnv ϕ
  → SVal (swknSTm ϕ (wknSTm γ s))

SwModel : Switch Γ Φ → Set
SwModel sw = Env γ → SEnv ϕ
  → SwNrm (swknSwTm ϕ (wknSwTm γ sw))
```

It is worth recognising that the representations of normal forms, values, and the model have not fundamentally changed. While it is the case that the representations are no longer isomorphic up to propositional equality, they are still equivalent in the sense that one can freely discard all cast constructors.

It is also worth mentioning that both wknFoo and swknFoo trivially unfold to constructor headed terms. This is another instance where the choice to use explicit substitutions has paid off by creating constructor-headed indices.

As a number of type definitions have been refined, it remains to refine the functions that depend on these types. The primary concerns are, of course, reification and interpretation. All that needs to be done is to insert casts at appropriate locations (e.g. where reduction is being performed). Only a few equations are included to demonstrate the idea of inserting casts, for a comprehensive list of all changes, consult the accompanying material.

$$
\begin{array}{l}
[\![\_]\!] \;:\; (t \;:\; \mathsf{Tm}\,\Gamma\,\tau) \;\rightarrow\; \mathsf{Model}\;t \\
[\![\;\mathsf{lam}\;t\;]\!]\;e \;=\; \lambda\,\gamma\;v \;\rightarrow\; \ldots\;\beta\mathsf{lam}\;\ldots\;::\;[\![\;t\;]\!]\;(\mathsf{wknEnv}\;\gamma\;e \triangleright v) \\
[\![\;\ldots\qquad]\!]\;e \;=\; \ldots
\end{array}
$$

$$
\begin{array}{l}
\mathsf{reify}_{\mathsf{S}} \;:\; (\sigma \;:\; \mathsf{SType}) \;\rightarrow\; \mathsf{SVal}\;t \;\rightarrow\; \mathsf{SNrm}\;t \\
\mathsf{reify}_{\mathsf{S}}\;(\sigma_1 \times \sigma_2)\;(s_1, s_2) \;=\; \eta\mathsf{prod}\;::\;\mathsf{reify}_{\mathsf{S}}\;s_1, \mathsf{reify}_{\mathsf{S}}\;s_2 \\
\mathsf{reify}_{\mathsf{S}}\;\ldots\qquad\quad n \qquad\quad\;\;=\; \ldots
\end{array}
$$

Modulo the cast constructors, the implementation of the indexed version is no different from the original implementation. The ... stands for uninteresting code. For example, the case for interpretation of lam states that there is a proof of convertibility involving the βlam rule. For signal reification, the case for products is converted simply by using the ηprod rule.

5.3.4 Embeddings

Our attention is now turned to the task of using the indices to derive the normalisation proof. This turns out to be quite straightforward. While one might argue that the modifications to include complex indices in the previous section complicate the presentation somewhat, the power they give to concisely present a proof of normalisation becomes clear in the following section. A more detailed comparison of the of merits of verification versus correctness by construction are discussed at the end of this section.

The most direct route to a proof is to show that the family of embedding functions (embFoo) from normal forms to terms respects convertibility. As a working example, the embedding from Nrm to Tm is given below.

```
embNrm : Nrm t → Tm Γ τ
embNrm lit          = lit
embNrm (lam t)      = lam (embNrm t)
embNrm (sigrel q)   = sigrel (embQNrm q)
embNrm (neu b t)    = embNeu t
embNrm (p :: n)     = embNrm n
```

The proof below states that the embedding of a normal form into a term is convertible to the index of the normal form. At base types, the proof is reflexivity. For other constructors, the appropriate congruence rules and recursive calls are used. The cast constructor requires the use of transitivity. This pattern is the same for all of the embedding proofs.

```
embNrmResp : {t : Tm Γ τ} → (n : Nrm t) → embNrm n ≃ t
embNrmResp lit          = refl
embNrmResp (lam t)      = congLam (embNrmResp t)
embNrmResp (neu b n)    = embNeuResp n
embNrmResp (sigrel q)   = congSr (embQNrmResp q)
embNrmResp (p :: n)     = trans (embNrmResp n) p
```

The above proof is true due to the invariant we have established for Nrm. Specifically, an element of Nrm t represents a normal form that is convertible to the term t. As our convertibility relation (≃) acts on terms, the proof embNrmResp simply states that embNrm respects convertibility.

5.3.5 Proof of Normalisation

Finally, the main normalisation theorem can be shown. The proof is constructive and does not depend on any postulates. Thus, if so inclined, one can execute the proof to compute the witness that a term is convertible to its normal form.

```
normProof : (t : Tm Γ τ) → nbe t ≃ t
normProof t = trans (embNrm (nbe t)) idL
```

As an example, consider the definitions of the S and K combinators given below, v2 and v3 define abbreviations for the second and third most recently bound variables, respectively.

```
v2 : Tm (Γ ▷ τ₁ ▷ τ₂) τ₁
v2 = [wkn] var
v3 : Tm (Γ ▷ τ₁ ▷ τ₂ ▷ τ₃) τ₁
v3 = [wkn ∘ wkn] var

K : Tm Γ (τ₁ → τ₂ → τ₁)
K = lam (lam v2)
S : Tm Γ ((τ₁ → τ₂ → τ₃) → (τ₁ → τ₂) → τ₁ → τ₃)
S = lam (lam (lam (app (app v3 var) (app v2 var))))
```

The identity combinator can be derived from S and K as demonstrated below. However, more importantly, using normProof we can also compute the proof tree that explains how app (app S K) K ≃ lam var. Inspecting the tree also reveals the computational rules that justify each step of reduction.

```
> nbe (app (app S K) K)
lam var
> normProof (app (app S K) K)
trans (trans (congLam ...) idL
```

In Sect. 5.1.4, a number of corollaries of the main theorems are discussed. It is easy to express these corollaries in our formalisation. In particular, ≃-idem-nbe states that nbe is idempotent, which follows immediately from normProof. Normalisation is congruent with respect to convertibility, which is demonstrated using simple equational reasoning by ≃-cong-nbe. The inverse can also be shown that nbe is injective with respect to convertibility, shown by $\simeq$-inj-nbe, once again via equational reasoning. Finally, by ascertaining that $\equiv \;\Rightarrow\; \simeq$, it is trivial to derive $\equiv$-inj-nbe from $\simeq$-inj-nbe.

$$\begin{aligned}
\simeq\text{-idem-nbe} &: \forall\; t \;\to\; \mathsf{nbe}\;(\mathsf{nbe}\; t) \simeq \mathsf{nbe}\; t \\
\simeq\text{-cong-nbe} &: \forall\; t\; u \;\to\; u \simeq t \;\to\; \mathsf{nbe}\; u \simeq \mathsf{nbe}\; t \\
\simeq\text{-inj-nbe} &: \forall\; t\; u \;\to\; \mathsf{nbe}\; u \simeq \mathsf{nbe}\; t \;\to\; u \simeq t \\
\equiv\text{-inj-nbe} &: \forall\; t\; u \;\to\; \mathsf{nbe}\; u \equiv \mathsf{nbe}\; t \;\to\; u \simeq t
\end{aligned}$$

5.3.6 Approaches to Mechanised Theorem Proving

We have shown a proof of normalisation by encoding the necessary invariants directly into the structure of terms. Of course, the proof method used in this section is not the only possible approach. A more conventional approach would be to prove correctness “directly” from the unindexed representation. This approach separates the representation from the properties and involves *verifying* the desired properties after the data has been constructed. In many situations, this separation can cause difficulties as the representation and the properties are often needed together anyway. Moreover, to show that a property holds for a subterm one typically needs to decompose the property to prove further lemmas about the term. Of course, the proof method we have chosen is not without its disadvantages. Notably, the representations become less reusable, and the additional indices make the formulation more complicated. However, as Danielsson has demonstrated, the chosen method is effective and can scale up to very expressive type theories (i.e. dependent types). Furthermore, the types of our functions also become somewhat more accurate: interpretation takes a term t and constructs a model of (indexed on) t.

A correct-by-construction approach may appear to have a big impact on the way something is formalised, to extend that aspects pertaining to the proof obscure the data. However, we argue that for our proof the fundamental idea is quite simple: we “remember” the term we started with by explicitly recording the equational laws used in each step. It is then very easy to extract this log of steps to produce a proof. Thus, while the additional indices may sometimes obscure the data, their introduction is quite straightforward and is a mostly mechanical process. One should also remember that verification is not without its complexities; for example, verifying lemmas relating to substitution and binding can be an intricate task.

There are advantages and disadvantages to both approaches. Verification is notionally simpler from first principles as only a simple representation of terms is needed. Properties and invariants can then be established incrementally on top of this foundation. However, as these properties are not inherent to the structure of a term they often need to be passed around explicitly as premises.

Finally, there is no need to assume that these two approaches must be mutually exclusive. A simple representation can usually be recovered from an indexed variant using an erasure function. In many cases, there is also a mapping from a simple datatype to an indexed variant that constructs indices from the verification function (i.e. $(a\ :\ \mathsf{Simple})\ \rightarrow\ \mathsf{NotSoSimple}\ (\mathsf{verify}\ a)$). Moreover, recent work on aspect-oriented programming and ornaments [Mcbride, 2011] may provide a means to further integrate these two different approaches.

5.4 A Model of Dynamism

Structural dynamism — the temporal composition of components — has shown to be very useful in practice. The generality and flexibility of the approach to dynamism taken in Hydra sets it apart from the current industry state-of-the-art. Hydra makes it easy to express structural changes, allowing a very large, or possibly even unbounded set of structural configurations to be specified.

The dynamism in $H_{[\![]\!]}$ is restricted slightly in that the number of configurations is always bounded. This is due to the lack of recursion in the functional host language. We decided to disallow recursively defined functions as their inclusion significantly complicates the formalisation. The focus of the semantics, and this thesis, should be on the aspects that are new and relatively unexplored. Nevertheless, we believe that recursion is compatible with our formalisation and would make for interesting work worth pursuing in the future.

A core tenet of the semantics described thus far has been *compositionality*; the meaning of a term is given entirely by the meaning of its subterms. We continue to abide by this tenet in this section, phrasing dynamism in a manner that is compatible with compositionality. Of course, our approach is not the only possible approach. Indeed, the syntax does not completely fix the possible interpretations of a term, and thus, our semantics is not the only possible meaning for a structurally dynamic term. We will discuss each design decision on an individual basis when presenting the implementation.

5.4.1 Shapes and Deformations

The dynamism semantics are based on two principles: *shapes* and *deformations*. A flattened system of equations describes a rose tree structure: the data at the nodes and leaves are equation system fragments, and the tree branching corresponds to the branches of a switch block. It is the structure of these trees that is captured by a *shape*. In other words, a shape describes the tree of possible structural configurations, in a manner not all that dissimilar to the graph of configurations described by a hybrid automata.

In our formalisation we define shapes inductively, resulting in a finite tree. In a setting with recursively defined configurations, shapes would be described coinductively, allowing branches of infinite depth, though remaining finitely branching. The shape tree is derived directly from the syntax of a flattened system of equations, leaving few design choices to be made at this point.

```
data QShape : Set where
  end    : QShape
  _∧_    : QShape → QShape     → QShape
  switch : Fin n  → SwShape n → QShape
```

The node and leaf data, which represents equation system fragments, is given by QShape. This shape retains the inherent tree structure of equations due to the ∧ constructor. Thus, we are in fact defining *a pair* of mutually defined trees. The end constructor denotes leaf equations that cannot result in structural reconfiguration, such as an empty or atomic equation. The switch constructor describes the shape of switching blocks. Even in a coinductive setting the number of branches is finite, allowing the currently active branch to be identified by the argument of type Fin.

```
data SwShape : ℕ → Set where
  []     : SwShape 0
  branch : QShape → SwShape n → SwShape (1 + n)
```

The shape of a switch block, like the term counterpart, is indexed on a natural number that fixes the number of branches. Notice that the recurrence of the ℕ index in each representation of switching blocks allows us to once again verify that the number of branches remains unchanged during a transformation.

The constructors of SwShape reflect those of Switch, defining a finite vector of equation shapes with one shape for each branch of the switch.

Shapes are useful as they provide an *abstract* representation of the structure of an equation. A change in structure can now be represented as a manipulation of an equation's shape. Of course, not all manipulations are valid. Indeed, the validity of a manipulation depends on the desired semantics of switching. We classify the *valid* manipulations using a datatype of *deformations*, thereby fixing the semantics of switching. An alternative semantics (e.g. allowing more than one branch to be active at a time) could be achieved by considering a different set of deformations. While we do not investigate alternatives in this thesis, we discuss the choice of deformations and how they impact upon the semantics.

The deformation of an equation or switch is expressed as a binary relation on shapes. An equation with shape h_1 can be structurally reconfigured to an equation with shape h_2 if there exists a deformation from h_1 to h_2. Thus, deformations are a sufficient condition for switching. However, they are not a necessary condition as there exist manipulations that are not meaningful (e.g. eliminating the second component of a ∧). Indeed, it is exactly the manipulations without meaning that we wish to exclude. Hence, a deformation is a model of switching and its interpretation provides a semantics and implementation for structural dynamism in $H_{[\![\,]\!]}$.

```
data QDeform : QShape → QShape → Set where
  end    : QDeform end end
  new    : (x y : Fin n)   → x ≢ y
                           → QDeform  (switch x w) (switch y w)
  _∧_    : QDeform h₁ h₃   → QDeform  h₂ h₄
                           → QDeform  (h₁ ∧ h₂) (h₃ ∧ h₄)
  switch : (x : Fin n)     → SwDeform (toℕ x) h₁ h₂
                           → QDeform  (switch x w₁) (switch x w₂)
```

The QDeform datatype provides four ways to build an equation deformation. The switch and ∧ constructors allow us to move through an equation without disturbing the structure. They effectively serve as congruence rules, allowing a deformation to be applied deep within a shape. The switch constructor also retains knowledge of the currently active branch, and passes this information onto SwDeform in the form of an additional index x. This is to ensure that only the active branch can be manipulated.

The end deformation insists that a leaf shape must be mapped back to another leaf shape. As no other constructors target end, we can be sure that leaf equations are not manipulated accidentally.

If a deformation is to have any effect it must eventually change the active branch of a switch. Switching from the x^{th} branch to the y^{th} branch is indicated by the constructor new x y p, where p is a proof that x and y are not equal. The proof is not strictly required as it will not be used to interpret the deformation. However, the proof prevents new from “switching” from a branch back to itself, an operation that does not make sense from a discrete event standpoint.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{data}\ \mathsf{SwDeform}\ (x\ :\ \mathbb{N})\ :\ \mathsf{SwShape}\ n\ \rightarrow\ \mathsf{SwShape}\ n\ \rightarrow\ \mathsf{Set}\ \mathsf{where}} \\
\quad [\,] & :\ \mathsf{SwDeform}\ x\ [\,]\ \ [\,] & \\
\quad \mathsf{skip} & :\ \mathsf{SwDeform}\ x\ w_1\ w_2 & \rightarrow\ \mathsf{SwDeform}\ x\ (\mathsf{branch}\ h\ w_1)\ (\mathsf{branch}\ h\ w_2) \\
\quad \mathsf{do} & :\ x \equiv n & \rightarrow\ \mathsf{QDeform}\ h_1\ h_2 \\
 & & \rightarrow\ \mathsf{SwDeform}\ x\ (\mathsf{branch}\ h_1\ w)\ (\mathsf{branch}\ h_2\ w)
\end{array}
$$

The SwDeform datatype dictates how the branches of a switching block can be deformed. The main purpose of the datatype is to ensure that only the active branch can be manipulated. This is achieved through the do and skip constructors, which allow a branch to be either deformed or left untouched, respectively. Any branch may be skipped, including the active branch, allowing a switch deformation to have no effect. Only the active branch can be deformed thanks to the equality proof in do.

Notice that do does not require an inductive deformation argument on the remaining branches of the shape w as there should only be one active branch at a time, and thus, deformations of any other branches are disallowed.

Furthermore, newly activated branches cannot themselves be deformed. A branch that is inactive when an event is raised cannot simultaneously raise an event as none of its switching conditions would be “live” at this point.

Before moving on to the meaning of a deformation, we take a brief detour to spell out the details of computing a shape from a flattened equation, as realised by the maps $\mathsf{shape}_{\mathsf{Q}}$ and $\mathsf{shape}_{\mathsf{Sw}}$. These maps perform the expected operations, extracting the abstract structure of an equation as characterised by a shape.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{shape}_{\mathsf{Q}}\ :\ \mathsf{QNrm} \circ \Phi\ \rightarrow\ \mathsf{QShape}} \\
\mathsf{shape}_{\mathsf{Q}}\ \mathsf{empty} & = & \mathsf{end} \\
\mathsf{shape}_{\mathsf{Q}}\ (s_1 = s_2) & = & \mathsf{end} \\
\mathsf{shape}_{\mathsf{Q}}\ (q_1 \wedge q_2) & = & \mathsf{shape}_{\mathsf{Q}}\ q_1 \wedge \mathsf{shape}_{\mathsf{Q}}\ q_2 \\
\mathsf{shape}_{\mathsf{Q}}\ (\mathsf{switch}\ x\ q) & = & \mathsf{switch}\ x\ (\mathsf{shape}_{\mathsf{Sw}}\ q)
\end{array}
$$

Recall that a flat system of equations necessarily contains no functional variables, hence the empty context ($\circ$) can be used for the arguments to $\mathsf{shape}_{\mathsf{Q}}$

and $\mathsf{shape}_{\mathsf{Sw}}$. Working on functionally-closed terms means that we can safely ignore the case for $\diamond$ as all signal relation applications have been eliminated.

$$\begin{array}{lcl}
\mathsf{shape}_{\mathsf{Sw}}\ :\ \mathsf{SwNrm} \circ \Phi\ n & \rightarrow & \mathsf{SwShape}\ n \\
\mathsf{shape}_{\mathsf{Sw}}\ [] & = & [] \\
\mathsf{shape}_{\mathsf{Sw}}\ (\mathsf{branch}\ sw\ s\ q) & = & \mathsf{branch}\ (\mathsf{shape}_{\mathsf{Q}}\ q)\ (\mathsf{shape}_{\mathsf{Sw}}\ sw)
\end{array}$$

5.4.2 Oracles and Interpretation

As we have opted to parameterise our system on the continuous behaviour it is not immediately clear how the concepts of flat equations, simulation runtime events, and deformations are related. Our solution is to introduce an abstract oracle that can compute a deformation from a flat system of equations:

$$\begin{array}{l}
\mathsf{postulate}\ \mathsf{oracle}\ :\ (q\ :\ \mathsf{QNrm} \circ (\mathsf{sr}\ \mathsf{unit})) \\
\quad \rightarrow\ \exists\,(h\ :\ \mathsf{QShape})\ (\mathsf{QDeform}\ (\mathsf{shape}_{\mathsf{Q}}\ q)\ h)
\end{array}$$

Reading the type of oracle we see that it accepts a flat system of equations q with the type of a top-level signal relation definition. It claims that there exists a shape h into which the shape of q can be deformed. In more concrete terms, oracle represents our simulation or “solving” function. Simulation runtime events are translated into deformations, for example, triggering the switching condition of an inactive branch corresponds to the new deformation. It is then a simple matter to construct a full deformation tree in a bottom-up manner that represents the discrete event. Thus, oracle is an abstraction over the continuous semantics in our formalisation, allowing us to explicitly state the potentially infinite iteration of flattening and runtime events.

Before presenting the interpreter for deformations it is useful to define QNrmSh and $\mathsf{SwNrmSh}$ that represent *shaped equations*: flat equations indexed by their shape. These datatypes represent a new view of shapes: a shape h is viewed as the set of equations qs where $\forall\ q\ :\ qs\,.\,\mathsf{shape}_{\mathsf{Q}}\ q\ =\ h$. Once again it is safe to omit the $\diamond$ constructor as all usages have been eliminated.

```
data QNrmSh (Φ : SCtx) : QShape → Set where
  empty  : QNrmSh Φ end
  _∧_    : QNrmSh Φ h₁  → QNrmSh  Φ h₂   → QNrmSh Φ (h₁ ∧ h₂)
  _=_    : SNrm ∘  Φ σ  → SNrm ∘  Φ σ    → QNrmSh Φ end
  switch : (x : Fin n)  → SwNrmSh Φ n h  → QNrmSh Φ (switch x h)

data SwNrmSh (Φ : SCtx) : (n : ℕ) → SwShape n → Set where
  []     : SwNrmSh  Φ 0 []
  branch : SwNrmSh  Φ n h₂ → SNrm ∘   Φ num
         → QNrmSh Φ h₁     → SwNrmSh Φ (1 + n) (branch h₁ h₂)
```

Finally, the meaning of a deformation is given by a function on shaped equations. The benefit of using shaped equations is evident in the type of ⟦_⟧QD. The interpreter is defined by matching first on the deformation and then on the shaped equation. Leaf equations should not been manipulated, hence, the case for end is just the identity. The interpretation of ∧ and switch is structural, simply applying the interpretation recursively. In the case of new, the argument indicating the active branch is updated.

```
⟦_⟧QD : QDeform h₁ h₂ → (QNrmSh Φ h₁ → QNrmSh Φ h₂)
⟦ end         ⟧QD q              = q
⟦ d₁ ∧ d₂     ⟧QD (q₁ ∧ q₂)      = ⟦ d₁ ⟧QD q₁ ∧ ⟦ d₂ ⟧QD h₂
⟦ switch x wd ⟧QD (switch .x sw) = switch x (⟦ wd ⟧SwD sw)
⟦ new x y _   ⟧QD (switch .y sw) = switch y sw
```

A switch deformation is interpreted by stepping through the branches of a switch in search of the active branch. If skip is encountered then we move on to the next branch. If [] is encountered, then all branches (including the active branch) have been skipped, implying that the switch deformation has no effect. If do is encountered then we have reached the active branch and the stored deformation can be interpreted using the equation at this branch.

```
⟦_⟧SwD : SwDeform x h₁ h₂ → (SwNrmSh Φ n h₁ → SwNrmSh Φ n h₂)
⟦ []   ⟧SwD []                  = []
```

$$\begin{aligned}
\llbracket\ \mathsf{skip}\ wd\ \rrbracket_{\mathsf{SwD}}\ (\mathsf{branch}\ sw\ s\ eq) &= \mathsf{branch}\ (\llbracket\ wd\ \rrbracket_{\mathsf{SwD}}\ sw)\ s\ eq \\
\llbracket\ \mathsf{do}\ \_\ d\ \ \rrbracket_{\mathsf{SwD}}\ (\mathsf{branch}\ sw\ s\ eq) &= \mathsf{branch}\ sw\ s\ (\llbracket\ d\ \rrbracket_{\mathsf{QD}}\ eq)
\end{aligned}$$

For the interpreters to be of any use we must be able to convert freely between shaped and unshaped equations, that is, between QNrm and QNrmSh. The maps Q+Shape and Sw+Shape provide a means to embellish equations and switches with their shape information. Equally, the maps Q-Shape and Sw-Shape let us erase all shape information.

$$\begin{aligned}
&\mathsf{Q{+}Shape} &&: (q\ \ : \mathsf{QNrm}\ \circ \Phi) &&\to \mathsf{QNrmSh}\ \ \Phi\ \ (\mathsf{shape}_{\mathsf{Q}}\ q) \\
&\mathsf{Sw{+}Shape} &&: (sw : \mathsf{SwNrm} \circ \Phi\ n) &&\to \mathsf{SwNrmSh}\ \Phi\ n\ (\mathsf{shape}_{\mathsf{Sw}}\ sw)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Q\text{-}Shape} &&: \mathsf{QNrmSh}\ \ \Phi\ \ \ h &&\to \mathsf{QNrm}\ \ \circ \Phi \\
&\mathsf{Sw\text{-}Shape} &&: \mathsf{SwNrmSh}\ \Phi\ n\ h &&\to \mathsf{SwNrm} \circ \Phi\ n
\end{aligned}$$

As a coherence condition we show that QNrm is a retract of QNrmSh, and similarly that SwNrm is a retract of SwNrmSh. The proofs retract$_{\mathsf{Q}}$ and retract$_{\mathsf{Sw}}$, which can be shown by induction on the input term, provide evidence that the aforementioned conversion functions are well-behaved, and that the round-trip does not alter the structure of the equation or switch.

$$\begin{aligned}
&\mathsf{retract}_{\mathsf{Q}} &&: (q\ \ : \mathsf{QNrm}\ \ \circ \Phi) &&\to \mathsf{Q\text{-}Shape}\ \ \ (\mathsf{Q{+}Shape}\ q) &&\equiv q \\
&\mathsf{retract}_{\mathsf{Sw}} &&: (sw : \mathsf{SwNrm} \circ \Phi\ n) &&\to \mathsf{Sw\text{-}Shape}\ (\mathsf{Sw{+}Shape}\ sw) &&\equiv sw
\end{aligned}$$

5.4.3 Metatheoretical Properties

In general, we cannot prove that an arbitrary equation system has a solution. Thus, given an initial set of equations, it is not possible to decide which switching condition, if any, will become active first. Even with a concrete oracle, we would not necessarily be able to relate the input equations (and switching conditions) to the output of our interpretation function. That is, we cannot show that the flat system of equations to be simulated will fire an event that corresponds to the deformation generated by oracle, and that ultimately, the resulting new generation of equations is related to the previous generation.

However, even though our formulation is parameterised on the continuous semantics , there are still some desirable properties of the interpreter that can be shown. The proofs $\mathsf{identity}_{\mathsf{QD}}$ and $\mathsf{identity}_{\mathsf{SwD}}$ show that interpreting an identity deformation produces the identity function. A similar theorem for composition cannot be shown as deformations are necessarily non-compositional. If a deformation could be composed then invariants of the datatype would not hold. For example, a new branch of a switch could be activated and then be immediately deformed.

$$\begin{array}{lll}
\mathsf{identity}_{\mathsf{QD}} & : (q : \mathsf{QNrmSh}\ \Phi\ h) & \rightarrow (d : \mathsf{QDeform}\ h\ h) \\
 & & \rightarrow [\![\ d\ ]\!]_{\mathsf{QD}}\ q \equiv q \\
\mathsf{identity}_{\mathsf{SwD}} & : (sw : \mathsf{SwNrmSh}\ \Phi\ n\ h) & \rightarrow (wd : \mathsf{SwDeform}\ b\ h\ h) \\
 & & \rightarrow [\![\ wd\ ]\!]_{\mathsf{SwD}}\ sw \equiv sw
\end{array}$$

5.5 Extensions

In this section we discuss a number of extensions to the $H_{[\![]\!]}$ core language. There are many ways that the core language could be extended, but we choose to focus on those that we believe are of particular importance: *local signal variables* and *delayed evaluation* of inactive switch branches. The former is more of a necessity than extension, which we have deferred until now due to the added complexity local variables bring to the implementation and semantics.

Each extension is implemented on top of the base core language as seen at the end of Sect. 5.2.1, keeping the presentation of each extension relatively simple. However, for the most part, the extensions are not in conflict with one another and could be implemented together.

Chapter 7 discusses avenues of future work, including further extensions to the $H_{[\![]\!]}$ core language. Notable extensions include recursively-defined signal relations, modelling connections à la. Broman and Nilsson [2012], and initialisation of dynamic models. We hope that our implementation of $H_{[\![]\!]}$ will provide a suitable framework to continue to explore the denotations and design space for cutting-edge equation-based modelling languages.

5.5.1 Local Signal Variables

Local signal variables had a profound impact on the small-step semantics of the balance type system in Sect. 4. It should come as no surprise then that local signal variables also present an interesting problem for our denotation model.

The key issue is that local variables do not behave in a conventional manner. Rather than serving as a token that is abstracted over and later substituted away, local variables instead represent the degrees of freedom in a set of equations. As reduction proceeds, local variables are accumulated in a bottom-up fashion. Thus, a fully-evaluated top-level signal relation will declare the local variables (via **local**) for the entire system of equations.

If we wish to verify that local variables are only ever used in a well-typed and well-scoped manner, then the process of accumulation becomes problematic. For λ-bound and interface signal variables, a context tracks the type of each variable. A variable can then only be used if it points to a valid position in the context. If we wish to apply this same technique to local variables, then the local variable context would need to be weakened as new local variables are discovered *during* reduction. The process of weakening is not in itself a problem: after all we have already designed a flexible notion of weakening using substitutions. The problems arise when determining *what* the new weakened context should be. The type of a signal relation (i.e. $\mathsf{sr}\ \sigma$) gives no clues as to the set of local variables contained within. The set of local variables for a flattened equation cannot simply be determined prior to evaluation.

At this point one might be tempted to suggest that information about local variables could be stored in the type of a signal relation (in the same manner as the interface variables). However, that would be an abstraction leak: when writing functions over equation fragments one should not be concerned with the local variables; an operation on equations should work for any fragment with a compatible interface, regardless of its internal details.

A solution to our problem is hinted at by the small-step semantics for the balance type system (see Fig. 4.5). To derive an algorithm from the semantics one might create a function taking an equation as input and producing a pair of a set of local variables and a flattened equation. By the same logic, our model of equations becomes a dependent product of a local variable context and a flat equation in that context.

Contexts, Terms, and Normal Forms

The structure of the local variable context is very similar to that of the functional context: a list of (signal) types. We generalise the existing notion of a context to allow both local and functional contexts to be derived from the same definition.

```
data GenCtx (A : Set) : Set where
  ○   : GenCtx A
  _▷_ : GenCtx A → A → GenCtx A

Ctx, LCtx : Set
Ctx   = GenCtx Type
LCtx  = GenCtx SType
```

Many of the original definitions involving context, such as variable indices, can also be generalised in the same way. Additionally, we define concatenation of context, which will be useful when describing the accumulation of local variables.

```
_++_ : GenCtx A → GenCtx A → GenCtx A
○         ++ D  =  D
(C ▷ a) ++ D  =  C ++ D ▷ a
```

For the most part, the definition of the language remains unchanged, with the largest alterations appearing in the model, and by extension, the interpreter. The notable exception being the representation of local variables (lvar), which in conjunction with an additional LCtx index gives an honest representation of a local variable by replacing the anonymous tokens used in the base core language.

```
data STm (Γ : Ctx) (Φ : SCtx) : LCtx → SType → Set where
  svar : STm Γ (▷ σ) Λ σ
  lvar : Var Λ σ → STm Γ Φ Λ σ
```

Just as in H_Δ, the local variables of an equation are bound by the enclosing signal relation by an additional parameter Λ to sigrel.

```
data Tm : Ctx → Type → Set where
  sigrel : (Λ : LCtx) → QTm Γ (▷ σ) Λ → Tm Γ (sr σ)
```

By the same reasoning that requires QTm and Switch to carry an index Φ, they are also required to carry the local variable context LCtx as an index. Similar alterations are also made to the normal forms, which are omitted here. For a full code listing consult the accompanying resources.

```
data QTm    (Γ : Ctx) (Φ : SCtx) (Λ : LCtx) : Set
data Switch (Γ : Ctx) (Φ : SCtx) (Λ : LCtx) : ℕ → Set
```

Objects of the Model

There are many ways to express the weakening of a variable. In the base core language we specialised a notion of replacements as this worked well with other aspects of the language, such as closures. This same notion of weakening could be reused for local variables, though it would require replacements to be further generalised away from Type. Moreover, we will never need to substitute a local variable, and thus the extension replacement (▷) is redundant.

A simpler option that suffices for our purposes is to define local context weakening as a function on variables, as given by ⊆.

```
_⊆_ : LCtx → LCtx → Set
Λ ⊆ K = Var Λ τ → Var K τ
```

A function space provides us with the usual identity ($\mathsf{\subseteq\text{-}refl}$) and composition rules ($\mathsf{\subseteq\text{-}trans}$), which give rise to the useful implication $\equiv \;\Rightarrow\; \subseteq$. Using concatenation, a more general version of wkn can also be derived ($\mathsf{\subseteq\text{-}wkn}$).

$$\begin{aligned}
&\mathsf{\subseteq\text{-}refl} &&: \Lambda \subseteq \Lambda \\
&\mathsf{\subseteq\text{-}trans} &&: \Lambda \subseteq K \;\to\; K \subseteq J \;\to\; \Lambda \subseteq J \\
&\mathsf{\subseteq\text{-}wkn} &&: \Lambda \subseteq (\Lambda +\!\!+ K) \\
&\mathsf{\equiv\text{-}implies\text{-}\subseteq} &&: \Lambda \equiv K \;\to\; \Lambda \subseteq K
\end{aligned}$$

Just as before with the functional-level weakening, we can characterise what it means to weaken the local variable context of a LCtx-indexed set. The family of functions $\mathsf{wknFoo}_{\Lambda}$ descend through the structure of a term applying the weakening to each local variable.

$$\begin{aligned}
&\mathsf{Weaken}_{\Lambda} \;:\; (\mathsf{LCtx} \;\to\; \mathsf{Set}) \;\to\; \mathsf{Set} \\
&\mathsf{Weaken}_{\Lambda}\; T \;=\; \Lambda \subseteq K \;\to\; T\,\Lambda \;\to\; T\,K
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wknQNrm}_{\Lambda} &&: \mathsf{Weaken}_{\Lambda}\; (\mathsf{QNrm}\; \Gamma\; \Delta\; \cdot) \\
&\mathsf{wknSNrm}_{\Lambda} &&: \mathsf{Weaken}_{\Lambda}\; (\mathsf{SNrm}\; \Gamma\; \Delta\; \cdot\; \sigma) \\
&\mathsf{wknSVal}_{\Lambda} &&: \mathsf{Weaken}_{\Lambda}\; (\mathsf{SVal}\; \Gamma\; \Delta\; \cdot\; \sigma)
\end{aligned}$$

We have now reached what is arguably the most important aspect of the extension: modifying Val and Model. These definitions *are* the semantic model of the core language and so we must be mindful when modifying them as even a small change might have a profound impact on the rest of the formalisation.

With this in mind, the changes we make to the Val family of definitions are modest. The changes reflect the comments made at the start of the section: the meaning of a value in isolation (i.e. not yet in context with a corresponding environment) is a dependent product (or existential). The product states that there exists a new local variable context that is used to weaken the “underlying” value. For equations and switches the underlying values are simply normal forms. At the signal level, the underlying value is given by the original definition from the base core language.

```
QVal : Ctx → SCtx → LCtx → Set
QVal  Γ Φ Λ    = ∃ (K : LCtx) (QNrm Γ Φ (Λ ++ K))

SwVal : Ctx → SCtx → LCtx → ℕ → Set
SwVal Γ Φ Λ n  = ∃ (K : LCtx) (SwNrm Γ Φ (Λ ++ K) n)

SValΛ : Ctx → SCtx → LCtx → SType → Set
SValΛ Γ Φ Λ σ  = ∃ (K : LCtx) (SVal Γ Φ (Λ ++ K) σ)

Val : Ctx → Type → Set
Val Γ  num        = Nrm Γ num
Val Γ  (τ1 → τ2)  = Γ ⊑ Δ → Val  Δ τ1      → Val  Δ τ2
Val Γ  (sr σ)     = Γ ⊑ Δ → SVal Δ Φ Λ σ → QVal Δ Φ Λ
```

We can further reason as to why this definition makes sense by looking at the example terms t_1 and t_2 given below. The model of t_1 is a function $\Gamma \sqsubseteq \Delta \;\rightarrow\; \mathsf{SVal}\;\Delta\;\Phi\;\Lambda\;\sigma \;\rightarrow\; \mathsf{QVal}\;\Delta\;\Phi\;\Lambda$, where Λ needs to be universally quantified as the precise context cannot be deduced from the type of t_1. Since the equation q_1 may make reference to local variables in Λ_1, the return type $\mathsf{QVal}\;\Delta\;\Phi\;\Lambda$ effectively only claims Λ as a lower bound for the context of local variables in the resulting equation. As the signal expression s in t_2 is restricted to the set of local variables Λ_2, the context Λ_2 is the lower bound in the application $t_1 \diamond s$, which is subsequently extended to the context $\Lambda_2 +\!\!+ \Lambda_1$ due to the existential in QVal.

```
t1 = sigrel Λ1 q1
t2 = sigrel Λ2 (t1 ⋄ s)
```

For cohesion with accumulation the definition of SEnv is also updated to use the new definition of signal values (SVal_Λ as opposed to SVal). This is necessary for the sigrel case of the interpreter where signal values will be added to the environment in a potentially extended local context.

```
data SEnv (Γ : Ctx) : SCtx → SCtx → LCtx → Set where
```

$$\triangleright \;:\; \mathsf{SVal}_{\Lambda}\;\Gamma\;\Phi\;\Lambda\;\tau \;\rightarrow\; \mathsf{SEnv}\;\Gamma\;\Phi\;(\triangleright\;\tau)\;\Lambda$$

Only minor changes are required to the Model family of definitions to carry through the recently-included local context indices.

$$\begin{aligned}
&\mathsf{Model} \;:\; \mathsf{Ctx} \;\rightarrow\; \mathsf{Type} \;\rightarrow\; \mathsf{Set}\\
&\mathsf{Model}\;\Gamma\;\tau \;=\; \mathsf{Env}\;\Delta\;\Gamma \;\rightarrow\; \mathsf{Val}\;\Delta\;\tau\\[1em]
&\mathsf{QModel} \;:\; \mathsf{Ctx} \;\rightarrow\; \mathsf{SCtx} \;\rightarrow\; \mathsf{LCtx} \;\rightarrow\; \mathsf{Set}\\
&\mathsf{QModel}\;\Gamma\;\Phi\;\Lambda \;=\; \mathsf{Env}\;\Delta\;\Gamma \;\rightarrow\; \mathsf{SEnv}\;\Delta\;\Psi\;\Phi\;\Lambda \;\rightarrow\; \mathsf{QVal}\;\Delta\;\Psi\;\Lambda\\[1em]
&\mathsf{SModel} \;:\; \mathsf{Ctx} \;\rightarrow\; \mathsf{SCtx} \;\rightarrow\; \mathsf{LCtx} \;\rightarrow\; \mathsf{SType} \;\rightarrow\; \mathsf{Set}\\
&\mathsf{SModel}\;\Gamma\;\Phi\;\Lambda\;\sigma \;=\; \mathsf{Env}\;\Delta\;\Gamma \;\rightarrow\; \mathsf{SEnv}\;\Delta\;\Psi\;\Phi\;\Lambda \;\rightarrow\; \mathsf{SVal}_{\Lambda}\;\Delta\;\Psi\;\Lambda\;\sigma\\[1em]
&\mathsf{SwModel} \;:\; \mathsf{Ctx} \;\rightarrow\; \mathsf{SCtx} \;\rightarrow\; \mathsf{LCtx} \;\rightarrow\; \mathbb{N} \;\rightarrow\; \mathsf{Set}\\
&\mathsf{SwModel}\;\Gamma\;\Phi\;\Lambda\;n \;=\; \mathsf{Env}\;\Delta\;\Gamma \;\rightarrow\; \mathsf{SEnv}\;\Delta\;\Psi\;\Phi\;\Lambda \;\rightarrow\; \mathsf{SwVal}\;\Delta\;\Psi\;\Lambda\;n
\end{aligned}$$

Interpretation

All that remains is to update the interpreter in light of the new model. In most cases this is a straightforward exercise in propagating local variable contexts and weakening the returned objects appropriately. We discuss the interesting cases, particularly those that highlight important aspects of the design.

The interpretation of interface signal variables demonstrates quite succinctly why SVal_{Λ} is needed in SModel. With the exception of the svar case, the interpretation of a signal expression would not need to expand on the initial local context. However, if we are to use the corresponding value in the signal environment — the only sensible option — then we must accept that new local variables may be introduced as a result. Indeed, looking back to our earlier example involving t_1 and t_2, the signal expression s must be weakened to account for the incoming local context Λ_1 during the reduction of $t_1 \diamond s$.

$$[\![\;\mathsf{svar}\;]\!]_{\mathsf{S}}\;e\;(\triangleright\;v) \;=\; v$$

The use of $\circ$ below indicates that the interpretation of lvar does not introduce new local variables. However, we must still make use of $\mathsf{\subseteq\text{-}wkn}$ as Λ is not definitionally equal to $\Lambda +\!\!+ \circ$.

$$[\![\ \mathsf{lvar}\ n\ ]\!]_{\mathsf{S}}\ e\ g\ =\ \circ, \mathsf{reflect}_{\mathsf{S}}\ (\mathsf{lvar}\ (\mathsf{\subseteq\text{-}wkn}\ n))$$

To demonstrate the propagation of local contexts, consider the pair example below. As both subterms may independently introduce a local context, the resulting local context is an aggregate of the two, and each sub-value must be weakened to encompass the local context of the other.

$$\begin{aligned}
&[\![\ \mathsf{pair}\ s_1\ s_2\ ]\!]_{\mathsf{S}}\ e\ g\ = \\
&\quad \mathsf{let}\ v_1\ =\ [\![\ s_1\ ]\!]_{\mathsf{S}}\ e\ g \\
&\quad\qquad v_2\ =\ [\![\ s_2\ ]\!]_{\mathsf{S}}\ e\ g \\
&\quad \mathsf{in}\ (\pi_1\ v_1 +\!\!+ \pi_1\ v_2), (\mathsf{wknSVal}_{\Lambda}\ ...\ (\pi_2\ v_1), \mathsf{wknSVal}_{\Lambda}\ ...\ (\pi_2\ v_2))
\end{aligned}$$

The story for $\diamond$ is much the same as the previous case: interpreting each subterm individually and then aggregating any new local contexts before returning an appropriately weakened composite value.

$$\begin{aligned}
&[\![\ t \diamond s\ ]\!]_{\mathsf{Q}}\ e\ g\ = \\
&\quad \mathsf{let}\ sv\ =\ [\![\ s\ ]\!]_{\mathsf{S}}\ e\ g \\
&\quad\qquad tv\ =\ [\![\ t\ ]\!]\ e\ \mathsf{id}\ (\pi_2\ sv) \\
&\quad \mathsf{in}\ (\pi_1\ sv +\!\!+ \pi_1\ tv), \mathsf{wknQNrm}_{\Lambda}\ ...\ (\pi_2\ tv)
\end{aligned}$$

Finally, saving the most interesting and subtle case to last, we look at the interpretation of sigrel. Modulo the signal context, the intermediate definition qv looks very similar to the interpretation of sigrel in the base core language. Using Val as a guide we know that interpreting q should result in an object of type $\mathsf{QVal}\ \Delta\ \Phi\ \Lambda$; that is, Λ is the “base” local context. As the supplied signal value v comes in a context K, we must weaken v and package it up with K before placing it in the interface signal environment. Once again referring back to the earlier example, v corresponds to the interpreted signal expression s, which must be weakened with respect to Λ_1 before it can be used.

Recall that before returning qv, we must also explicitly specify the new local variables introduced as a result of interpreting the subterms. In this case, the context Λ is revealed along with any new local variables introduced as a result of interpreting q, hence $\Lambda +\!\!+\ \pi_1\ qv$ are accumulated alongside K.

$$
\begin{array}{l}
[\![\, \mathsf{sigrel}\ \Lambda\ q \,]\!]\ e \ = \ \lambda\ \{K\}\ \phi\ v\ \rightarrow \\
\quad \mathsf{let}\ qv\ :\ \ \mathsf{QVal}\ \Delta\ \Phi\ \Lambda \\
\qquad qv\ =\ [\![\, q \,]\!]_{\mathsf{Q}}\ (\mathsf{wknEnv}\ \phi\ e)\ (\triangleright\ (K, \mathsf{wknSVal}_{\Lambda}\ ...\ v)) \\
\quad \mathsf{in}\ (\Lambda +\!\!+\ \pi_1\ qv), \mathsf{wknQNrm}_{\Lambda}\ ...\ (\pi_2\ qv)
\end{array}
$$

Successfully implementing the interpreter gives us some assurances. Specifically, local variables remain well-typed and importantly, well-scoped during normalisation, giving us some confidence that our approach is correct.

5.5.2 Delayed Branch Normalisation

There are a number of situations where it is desirable to defer the normalisation of a switching branch until that branch becomes active. Rather than eagerly evaluating every branch of a switch, instead, only the *active path* is computed to produce the set of equations that are required for the current generation of simulation. Reasons for deferring normalisation include (but are not limited to):

- Operationally, premature normalisation of branches is in general wasteful as a branch may never be activated. Thus, leaving inactive branches unnormalised reflects what an efficient implementation *should* do.

- To address the issue of simulation state transfer across switches, it is necessary to allow normalisation to depend on state information *at the time of a switching event*, by allowing references to signal values just prior to a switch event, or by events carrying a payload. Once such functionality is added to the language it will not in general be possible to normalise a branch before it is activated.

- From many real-world applications, it is also desirable, or even essential, to allow equations to be computed by tail-calling a signal relation, guarding the recursive calls by a branch (for example, consider modelling a state machine). Postponing normalisation of branches permits a form of guarded corecursion, allowing such calls to be productive.

In the base core language we were not concerned with the above points; efficiency is a low priority, transfer of simulation state is not considered, and recursive relations are omitted to keep the initial presentation tenable. Of course, this is not to say that the above points are without value, and each one could make for interesting work in the future. Therefore, it is essential to investigate delayed branch normalisation as a prerequisite to these extensions.

The key issue is that we wish to delay only a fragment of a larger program. While a branch may be left unevaluated, the surrounding context will still be fully reduced. This leads to a situation where a variable bound outside the fragment is substituted for a value. This effectively removes the variable from the context as all instances of it will be replaced by the new value, including those inside the inactive branches of a switch.

In an earlier article (see [Capper and Nilsson, 2012]), we solved the problem by defining a family of substitution functions that traverse the structure of a term and substitute the "external" variables without performing any further reduction. This was a reasonable solution that allowed us to adjust the contexts of an inactive branch to keep it compatible with the enclosing program.

However, the above solution was devised for a core language without an explicit notion of substitutions. In $H_{[\![]\!]}$ we have a much simpler way to adjust the context of a term called *explicit substitution application* that is denoted $[\phi]\ t$. By converting environments to substitutions we demonstrate a simple technique for delaying the evaluation of a subterm in a setting with explicit substitutions.

Equation Modes

The base core language does not distinguish between active and inactive branches as every branch is fully normalised, and thus is represented as a normal form (i.e. QNrm). If we wish to delay the normalisation of an inactive branch we must represent the branch as an unnormalised term (i.e. QTm). Therefore, the representation of an equation at a given branch depends upon whether or not that branch is active. Thus, the first step is to make this dependency explicit by refining the representation of normal forms and by using *equation modes*:

```
data QMode (Γ : Ctx) (Φ : SCtx) (m : ℕ) : ℕ → Set where
  here  : QNrm Γ Φ → QMode Γ Φ m m
  there : m ≢ n → QTm Γ Φ → QMode Γ Φ m n
```

An equation mode (herein simply mode) is intended to replace the representation of an equation at a branch. A mode is indexed on two natural numbers m and n. The first number denotes the index of the active branch and the second number denotes the index of the branch containing the mode. Only when the two numbers are equal (here) (i.e. the current branch is the active branch) should we represent the branch equation as a normal form. In all other circumstances the current branch must be inactive and an unevaluated term should be used to represent the equation instead.

To use a mode we can thread the information about the active branch through from the switch to the QMode. This is simply a case of adding an additional parameter to SwNrm to record the index of the active branch.

```
data SwNrm (Γ : Ctx) (Φ : SCtx) (m : ℕ) : ℕ → Set where
  []       :    SwNrm Γ Φ m 0
  branch   :    SwNrm Γ Φ m n → SNrm   Γ Φ num
              → QMode Γ Φ m n → SwNrm Γ Φ m (1 + n)

data QNrm (Γ : Ctx) (Φ : SCtx) : Set where
  switch : (x : Fin n) → SwNrm Γ Φ (toℕ x) n → QNrm Γ Φ
```

Environments as Substitutions

Environments bear a number of similarities to substitutions. An environment is indexed on two contexts: the first represents the context of the values contained within, and the second is used to constrain the length of the environment and the types of its values. It is implicit in the definition of $\mathsf{Env}\ \Gamma\ \Delta$ that Γ subsumes Δ, and it is this subsumption that gives rise to a substitution.

A environment is, in some sense, a more general notion than a substitution. An environment might be empty, for which there is a corresponding empty substitution. An environment can also be extended, just like a substitution. It is this correspondence that allows us to map an environment to a substitution provided we have a means to convert a value to a term. However, there is no obvious way to express the composition of environments, and thus we cannot express the inverse map from substitutions to environments.

$$\begin{aligned}
&\mathsf{envToSub} : \mathsf{Env}\ \Gamma\ \Delta \ \rightarrow\ \Gamma \Rightarrow \Delta\\
&\mathsf{envToSub}\ \circ \quad\quad = \mathsf{empty}\\
&\mathsf{envToSub}\ (e \triangleright v) \ = \mathsf{envToSub}\ e \triangleright \mathsf{embNrm}\ (\mathsf{reify}\ v)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sEnvToSub} : \mathsf{SEnv}\ \Gamma\ \Phi\ \Psi \ \rightarrow\ \Phi \Rightarrow_{\mathsf{S}} \Psi\\
&\mathsf{sEnvToSub}\ (\triangleright\ v) \ = \triangleright\ \mathsf{embSNrm}\ (\mathsf{reify}_{\mathsf{S}}\ v)
\end{aligned}$$

The interpretation of the equations at a branch is now handled by an interpretation function for modes ($[\![\ q\ ]\!]_{\mathsf{QM}}\ e\ g$) that proceeds by determining if the current branch is active. We make use of the decidable predicate $m \overset{?}{=} n$, which returns yes if the numbers are equal along with a witness to this fact or else it returns no with a proof to the contrary. If the decision procedure returns yes then the branch is active and we should interpret the contained equation. If it returns no then the branch is inactive and we must explicitly weaken the equations by the converted environments.

$$\begin{aligned}
&[\![\_]\!]_{\mathsf{QM}} : \mathsf{QTm}\ \Gamma\ \Phi \ \rightarrow\ \mathsf{Env}\ \Delta\ \Gamma \ \rightarrow\ \mathsf{SEnv}\ \Delta\ \Psi\ \Phi \ \rightarrow\ \mathsf{QMode}\ \Delta\ \Psi\ m\ n\\
&[\![\ q\ ]\!]_{\mathsf{QM}}\ e\ g\ \mathsf{with}\ m \overset{?}{=} n
\end{aligned}$$

```
... | yes refl = here    (⟦ q ⟧Q e g)
... | no p     = there p (⟨ sEnvToSub g ⟩ [envToSub e] q)

⟦_⟧Sw : Switch Γ Φ m → SwModel Γ Φ m n
⟦ []            ⟧Sw e g = []
⟦ branch sw s q ⟧Sw e g = branch (⟦ sw ⟧Sw e g) (⟦ s ⟧S e g) (⟦ q ⟧QM e g)
```

This procedure allows us to avoid premature normalisation, or even inspection of the equations at an inactive branch. Only when the branch is activated will the explicit substitutions be eliminated and the equations be normalised.

To date, we have not investigated the impact of this extension on the semantics of dynamism (see Sect. 5.4). Integrating this extension with dynamism is not trivial as the current implementation relies on knowing the complete structure of a flattened equation prior to simulation runtime. We speculate that a formulation involving potentially “unknown” shapes might provide a solution, and look forward to investigating these problems in the future.

Chapter 6

Related Work

6.1 Structural Types

6.1.1 Modelica

Modelica is an industrial-strength, equation-based language for acausal modelling of hybrid systems. The language design draws heavily from object-oriented languages with notions like classes and inheritance used to structure the models. As per the Modelica specification [Mod, 2012, pp. 43–48] models are required to be *locally balanced*. A model is locally balanced if it locally declares or inherits the same number of variables as equations. *Global balance* is then defined as equation-variable balance for a complete, composite model, which follows immediately if all subcomponents are locally balanced.

The language specification only requires checking of the local balance once specific values of parameters are known. The number of variables and equations may depend on the constants through conditional selection among blocks of equations and array sizes. Checking that a model is locally balanced for *all* possible values of the parameters is left as a "quality-of-implementation" issue.

Compared to our approach, Modelica is quite restrictive: there are good reasons for why certain components need to be locally *un*balanced, and then used as building blocks of larger systems that ultimately will be balanced. For this reason, Modelica allows components to be marked as *partial*, thereby disabling balance checking (in isolation) for those components. Furthermore, Modelica does not classify equations depending on which variables occur in them (i.e. distinguishing between local, mixed, and interface equations). As such, the class of structural properties checked by Modelica is similar to the simple balance type system (see Sect. 4.2), except without consideration for locally unbalanced models. Therefore, Modelica checks for a much smaller class of structural properties than the constrained type system (see Sect. 4.3).

Finally, Modelica lacks a notion of first-class models: there are methods for parametrising models on other models, but these do not approach the generality of FHM (see [Giorgidze, 2012]). However, this does mean that checking balances late, once parameters are fully known, suffices in the case of Modelica.

6.1.2 Broman, Nyström, and Fritzson

Broman et al. [2006] have developed a more flexible approach to modular balance checking than the approach described by the current Modelica specification. Notably, models are not required to be locally balanced provided that the fully assembled system is balanced. The type system, called *Structural Constraint Delta* (C_Δ), is developed for a subset of Modelica called *Featherweight Modelica*.

The idea behind C_Δ is to refine the notion of type equality such that two models are equal only if they are equal under the Modelica interpretation (see [Mod, 2012]) and have the same equation-variable balance. This refinement is motivated by the principle of safe substitution: replacing one class by another is safe only if the replacement preserves the global balance of the system.

The refined notion of type equality is realised by annotating the type of a class with the difference (C_Δ) between the total number of defined equations and variables. The annotation is a concrete value as Featherweight Modelica classes are not first-class entities: the information required to compute the annotation is always manifest in the structure of the object being analysed. Hence, the C_Δ may always be computed in a bottom-up fashion.

By contrast, the type system discussed in this thesis lifts a number of restrictions inherent to C_Δ. Our approach permits first-class models. Hence, we do not rely on manifest type information as the structure of a model may be partially or even completely unknown. Furthermore, parameterised models are parametric in their balance; a model may be instantiated with different values for its parameters, resulting in distinct balances for each usage of the model within the same context. As with Modelica, the approach taken by Broman et al. is strictly balance oriented. Thus, once again the class of structural properties checked by C_Δ is smaller than that of our type system $H_\models$.

To our knowledge, the idea of incorporating balance checking into the type system of a non-causal modelling language was suggested independently by Nilsson et al. [2003] and Broman et al. [2006], with the latter giving the first detailed account of such an approach.

6.1.3 Nilsson

As a precursor to the work presented in this thesis, Nilsson [2008] conducted a preliminary investigation into a type system for checking stronger properties relating to the structure of equations and variables beyond that of simple balance. Nilsson's *structural types* are designed to rule out systems with structural singularities that would otherwise be accepted under a simple balance checking approach. Furthermore, Nilsson also developed his approach for the FHM framework, and thus, his motivations were much the same as our own.

Nilsson's approach is centred around the notion of incidence matrices. The incidence matrix of a system of equations represents the occurrences of variables in equations. By approximating incidence matrices in the types of signal relations and equations, Nilsson approaches the capabilities of the techniques described by Bunus and Fritzson [2002], while retaining the capability of checking fragments in isolation. Partitioning equations into classes depending on whether the occurring variables are local, interface, or both is central to Nilsson's approach and led to the notion of equation kinds in this thesis.

However, Nilsson's work is only a preliminary investigation into structural types. Notably, the approach does not consider first-class models; that is, Nilsson assumes that the concrete structure of an equation required to compute an instance matrix is known statically. While this is sufficient for a language such as Modelica, it is not clear that it would be possible to generalise the method to a first-class setting while retaining the precision of the types.

Nilsson formalises the fundamental concepts of computing incidence matrices from concrete signal relations, including equation composition and handling situations that may give rise to ambiguity. Nilsson forgoes a presentation of the type system in its entirety (i.e. expressing the type system as a deductive set of rules and relating this to a semantics) due to the preliminary nature of the work. Instead, Nilsson presents an algorithm for determining the (approximate) best possible structural type for an abstract system of equations.

The time complexity of the algorithm for computing structural types is also a concern as it relies on partitioning the set of mixed equations in all possible ways. Moreover, a disadvantage of the precision of the types is that they may be hard to understand and cumbersome to use in practice. Suitable methods to communicate type errors to the programmer would also have to be investigated, although the paper does suggest that the work by Bunus & Fritzson could provide a good starting point. By contrast, the type system presented here does handle first-class models, but is not able to detect as many structural problems. Furthermore, this thesis also considers structural dynamic systems.

6.1.4 Bunus and Fritzson

Bunus and Fritzson [2002] describe a static analysis technique for pinpointing problems with modular systems of equations developed in equation-based languages such as Modelica. The primary motivation for their work is to develop effective debugging techniques for equation systems.

They are concerned with the same structural properties as we are, but by permitting systems to be flattened before analysis allows them to perform a much more fine-grained localisation of problems. In essence, viewing the flattened system as a bipartite graph (the nodes being the equations on the one hand and the occurring variables on the other), they attempt to put the equations in a one-to-one correspondence with variables occurring in them by performing a Dulmage-Mendelsohn canonical decomposition. This technique partitions the system into a *well-constrained* part (a one to one correspondence is possible), an *over-constrained* part (too many equations), and an *under-constrained* part (too many variables). If the latter two parts are empty, the system as a whole is structurally well-constrained.

The main contribution of the work is the localisation and reporting of program errors in a method consistent with the programmers perception of the system. An efficient technique for annotating equations for future analysis is also outlined. The methods discussed are robust, even in the face of program optimisations that may change the intermediate structure of the modular system of equations. Bunus and Fritzson implemented a prototype of their tool, attached to the MathModelica simulation environment, and evaluated the usability of their system in that setting. A case study is presented in their paper.

The methods outlined by Bunus and Fritzson are applicable only to a modularly constructed system once it has been flattened. Thus, the methods are in many ways complimentary to the work presented in this thesis. The methods could even be performed during simulation, making them potentially very useful for analysis of iteratively-staged, structurally-dynamic systems.

The work by Bunus and Fritzson illustrates the benefits of going beyond basic balance checking when finding problems with systems of equations. Some of those benefits are also realised by our type systems thanks to the classification of equations into different kinds depending on the variables that occur in them (i.e. by approximating individual variable occurrences).

6.1.5 Furic

Furic [2009] proposes a novel approach to model composition for Modelica. The approach is centred around a notion of equation-variable balance, which provides improved guarantees of compositionally. Once again, no classification of equations is made depending on whether occurring variables are local or otherwise. Furic's balance checking algorithm works on a *physical connection graph* describing the structure of an assembled system. Thus, its present formulation is not modular. However, Furic suggests that the additional syntactic information that the proposed approach makes available could form a basis for a type system for enhanced static checking and separate compilation. Interestingly, Furic's approach supports a much more flexible notion of structural dynamism than Modelica does at present, although this hinges on either pre-enumerating all configuration for checking purposes, or running the checking algorithm at each structural change during simulation.

Despite being quite different from our type-based approach, Furic's work underscores the practical importance of enforcing constraints on the equation-variable balance for a modularly constructed system of equations. Moreover, his approach to composition offers a number of advantages over Modelica's, such as the protection of intellectual property when building models from proprietary libraries, and it would be interesting to see if the approach can be recast into a type-based approach and adapted to the FHM setting.

6.1.6 Modelyze

Broman and Siek [2012] have developed a framework, called Modelyze, in which domain-specific modelling languages can be embedded. The goal of their work is to provide a host language that can be extended, as required, to accommodate the needs of a specific domain. The language is based on gradual typing, which allows fine-grain control over a mix of both static and dynamic types. This is used to provide types for symbolic expressions that, due to symbolic lifting analysis, are integrated seamlessly into the language. This integration means that symbolic errors arising from domain-specific features remain meaningful and easy to understand to the domain expert. Moreover, the separation of domain-specific features means that the core language is simple and straightforward to formalise compared to the current industrial state-of-the-art (e.g. Modelica).

The key features of Modelyze are first-class functions for structuring models and symbolic types for detecting symbolic errors. Like Hydra, the functional host language provides mechanisms for abstracting over and composing models. However, compared with Hydra, the type of symbolic errors that can be detected statically are much weaker. In particular, no degree-of-freedom analysis (such as equation-variable balance) is checked. Instead, Modelyze only verifies static information that can be checked in isolation, such as uninitialise signals.

As such, we believe that the work on Modelyze could be mutually beneficial to our own. Our work on static analysis in the presence of unknown model structure would be beneficial to Modelyze, and the symbolic integration it supports could be used in Hydra to provide a better experience for the end user.

6.2 Semantics

In this section we look at the work that is closely related to our own work on dynamic semantics. NbE is a key component of our method, and the field of NbE is an area of active research [Aehlig et al., 2012, Altenkirch et al., 2001, Fiore, 2002, Vestergaard, 2000]. This existing literature makes contributions to the field of NbE by studying the theory of the semantic technique directly. By contrast, our work makes contributions to the field of equation-based languages, and only uses NbE as a tool, in a similar manner to that of Danielsson [2006] or Fridlender and Pagano [2013]. Thus, in this section we focus on the work that is related to our contributions: work that attempts, by whatever means, to formalise the semantics of equation-based languages.

6.2.1 Kågedal

To our knowledge, the earliest attempt to *formally* specify the semantics of a noncausal modelling language is Kågedal's natural semantics for a subset of Modelica [Kågedal, 1998, Kågedal and Fritzson, 1998]. The approach is similar to our own: the semantics provide a translation from a Modelica model into a specialised language of flat equations called *Flat Modelica*.

The main focus of the work is to accurately capture the meaning of Modelica's object-oriented constructs (e.g. classes, inheritance, etc.) as well as Modelica specifics such as connect-equations. The formal specification was written in response to difficulties arising from the under-specification and ambiguity in early versions of the Modelica language. Furthermore, the targeted language of flat equations needed to be compatible with existing equation solvers, and the output of a translation was expected to be human readable.

An additional goal of the work is the automatic generation of a Modelica implementation from the semantics, an aspect that is reminiscent in our work as we attempt to capture both semantics and implementation using Normalisation by Evaluation. However, this is where the similarities between our two works end. The semantics is expressed in RML, a restricted form of natural semantics that can be compiled into fairly efficient code.

Kågedal's work provides an important first step, though it remains "incomplete" in a couple of aspects. As there is no notion in Modelica of generating new structural configurations during simulation, Kågedal's semantics is (as mentioned earlier) entirely static: it encompasses conventional static semantics (i.e. Modelica's type system etc.) and a translation into flat equations.

Modelica does have limited support for hybrid systems: the "if" and "when" language constructs provide a restricted form of dynamism, and the more general notion of algorithm blocks provide some means to capture hybrid behaviour. However, no attempt is made in Kågedal's work to capture the dynamic semantics of such constructs. Instead, after static checking, such constructs are transliterated into the output format. Thus, Flat Modelica is a rich language, including not just ordinary mathematical equations, but also an algorithmic sub-language and facilities for handling events. This is in contrast to our work: the meaning of a structural configuration in our setting is given by a simple set of conventional mathematical equations, and our semantics does explain how the system evolves in response to events. Furthermore, no attempt is made to prove any formal properties of Kågedal's semantics, neither manually nor through the use of a mechanised theorem prover.

6.2.2 Henzinger

Henzinger [1996] proposed the *hybrid automaton* as a formal model for a hybrid system. A hybrid automaton is given by a graph along with a finite set of time-varying variables. The vertices of the graph represent a possible system configuration and contain equations that describe the dynamic behaviour of the system in that state. The edges of the graph represent the switching conditions that dictate when transitions between states can and must occur. For that reason, a hybrid automaton provides a model of both the discrete and ideal continuous semantics for a system of equations.

A hybrid automaton effectively represents a set of flattened equation systems that describe the continuous behaviour along with a set of predicates that describe the discrete behaviour. It is not in itself concerned with the process of reaching a flat system from a modular structure (whatever an abstract representation of the modular structure might be). It is, however, a suitable semantic model into which noncausal, hybrid languages can be translated. For example, Beek et al. [2006] use this approach to give semantics to the modelling and simulation language χ (Chi).

Our semantics can be understood as describing a *tree* of structural configurations, whereby the continuous behaviour is given by the equations at the active node and its direct ancestors. If we realise *unbounded* structural dynamism then our semantics could be viewed as describing the construction of the reachable parts of a possibly infinite hybrid automata *on demand* from a high-level, declarative, system model. Taken together, this thesis and hybrid automata would provide a complete semantics of FHM, and thus Henzinger's work is very much complementary to our own.

6.2.3 Giorgidze

Giorgidze [2012] was the first to attempt a semantics for FHM, and to our knowledge, the first to try and capture both continuous and discrete aspects of a noncausal language that supports unbounded structural dynamism.

Giorgidze's approach is to translate the concrete syntax of Hydra (itself an embedded domain specific language of Haskell) into a Haskell expression augmented with second order propositional logic that describes the continuous behaviour of the system. The semantics is "ideal," with the meaning of an equation fragment being the solution to the equations, if a solution exists. Giorgidze also works in a non-constructive setting, as the decidability of the semantics depend on evaluating arbitrary functional values and deciding signal equality.

The approach described by Giorgidze is quite different to our own. We make no attempt to describe the continuous aspects in this thesis, but instead seek to separate these concerns, allowing the continuous aspects to be described in whatever way is most appropriate for the application. As a result, we do not couple continuous concepts to the discrete semantics (e.g. using a double precision floating point number as an approximation of continuous time).

Our semantics is *interpreted* into a model, whereas Giorgidze's semantics is more closely related to compilation: it takes a high-level DSL into a more primitive expression language, with the meaning of a program ultimately relying on the semantics of Haskell. As we were able to design our model from the ground up, we have more control over the meaning of an equation fragment, allowing us to make fundamental aspects of FHM explicit that are left implicit in Giorgidze's formulation (e.g. local signal variable propagation).

Finally, our approach is constructive: the semantics and implementation are one and the same. Conversely, in Giorgidze's semantics it is not clear, for example, what the meaning of non-termination or convergent events (the occurrence of infinite events in finite time) may be. Moreover, no properties of Giorgidze's system are demonstrated, mechanically or otherwise.

6.2.4 Pepper et al.

Taking a new approach, Pepper et al. [2011] give a semantics to a Modelica-style language with variable structure by giving meaning directly to a modular system of components. They describe the continuous semantics of a component in situ without first flattening the system of equations. The individual semantic blocks can then be composed to provide meaning for a complete model.

Pepper et al. consider variable-structure systems that are slightly more permissive than those currently accepted by the Modelica standard. However, as with our semantics, their approach is still restricted to *bounded* dynamism, and requires that all possible system configurations be computed statically.

They consider both an ideal semantics and an approach based on *simulation semantics* (i.e. "real" semantics that account for approximations and uncertainty problems) for continuous aspects of the language. In contrast to our own work, they are not interested in flattening a modular system, and thus, their work is primarily focused on the continuous rather than the discrete. Hence, their work is in some ways complementary to our own.

While the idea of prescribing a semantics before flattening is both novel and interesting, it is unclear how to relate the semantics to an implementation: the former does not naturally give rise to the latter, as is the case in our thesis. The approach we have taken strongly resembles a real-world implementation (i.e. our interpreter uses methods very similar to those found in the Hydra framework). Furthermore, Pepper's work is centred around Modelica, a language that lacks a true notion of first-class models. Therefore, it would be interesting to see how Pepper's work could be modified to work in the more general framework of FHM, in a setting where the structure of a model may be completely unknown.

6.2.5 Broman

Broman [2010] (also [Broman and Fritzson, 2008]) has developed Modelling Kernel Language (MKL), a meta-modelling language intended as a core language for noncausal modelling languages such as Modelica. MKL is a functional language with a notion of first-class models, making it quite similar to FHM. How a model should be used, along with other useful meta-operations, can be specified as part of a model definition, allowing useful properties and constraints to be directly encoded in a library of components.

MKL utilises a hybrid type system, with the core of the language given by a statically-typed, effectful λ-calculus. Conversely, models in MKL are dynamically-typed and the static and dynamic aspects of the type system are unified using Gradual Typing [Siek and Taha, 2006].

Broman provides a small-step operational semantics for MKL, and thus, his approach is similar to earlier work of our own [Capper and Nilsson, 2010] and to the semantics we present in Chap. 4.2.3. Furthermore, Broman uses his semantics as a basis to prove the conventional notion of type safety (progress and preservation) for his type system.

To date, the work on MKL has not yet considered systems of variable structure, though Broman has stated (through personal communication) that structural dynamism is an area of active research for MKL. Hence, we hope that the work presented in this thesis will be useful, not only for MKL, but for any semantic investigation into structurally-dynamic modelling languages.

6.2.6 Wan and Hudak

Wan and Hudak [2000] have given semantics to both the continuous and discrete parts of a simple FRP language. They present both an ideal semantics and also an operational semantics that makes use of discrete sampling. The discrete semantics is shown to converge to the ideal continuous semantics in the limit as the time steps approach zero.

It would be interesting to see how the techniques employed by Wan and Hudak could be adapted to work for FHM. In particular, generalising their approach from signal functions to signal relations would be an important first step, assuming such a generalisation is possible using their methods. They also make some strong assumptions, that while appropriate for their applications, may not be admissible when working in an acausal setting. For example, their approach forbids instantaneous predicates events and they assume *uniform convergence*.

6.2.7 Acumen

Acumen [Zhu et al., 2010] is a language for modelling and simulation of structurally-dynamic hybrid systems. The continuous aspects of the language are specified via DAEs, like Hydra, along with partial differential equations (PDEs). Unlike Hydra, however, the discrete aspects are modelled via an event-oriented paradigm of FRP. Acumen has recently been extended to support unbounded structural dynamism [Taha et al., 2012].

To date, the work on Acumen has been focused on automatic methods for mapping analytical models to executable code. While no formal verification has been attempted so far, the authors earlier work on verifying multi-staged programs [Inoue and Taha, 2012] suggests that our work may also be a useful starting point for formalising Acumen.

6.2.8 Sol

Zimmer [2013] has developed Sol (see also [Zimmer, 2007]), a language similar to

Modelica that focuses on *efficiently* modelling systems of variable structure. The language supports unbounded structural dynamism and makes use of symbolic methods to try and minimise the number of equations that need to be modified or added as the result of a discrete event. As Zimmer is primarily interested in language design and advanced implementation techniques only an informal account of the semantics is provided.

6.2.9 Danielsson

The proof technique that allows us to give a proof of normalisation (see Sect. 5.3) is due to Danielsson [2006]. In his article, Danielsson is interested in showing how a language with inductive-recursive families can be used to formalise a dependently-typed λ-calculus. While the article is not directly focused on giving a semantic model, it does provide the first formal account of Normalisation by Evaluation for a dependently-typed language.

Chapter 7

Summary and Future Work

7.1 Summary

In this thesis, we have made contributions to the static and dynamic semantics of equation-based languages, and specifically to languages for modelling and simulation. These contributions came in the form of a type system designed to enforce desirable structural invariants, and from the construction and partial verification of a semantic model of an FHM-like language.

Chapter 1 introduced equation-based languages, and specifically, languages for physical modelling. Two different areas of interest were identified: the use of static techniques to detect structural anomalies in modular equation systems, and the formalisation of a discrete semantics for equation-based languages.

Chapter 2 introduced the fundamental concepts common to many equation-based modelling languages, such as modularity, acausality, hybrid behaviour, and structural dynamism. Equation-based modelling was illustrated through the object-oriented approach of Modelica, and through the functional approach of FHM. Due to the importance of structural dynamism in this thesis, the flexibility and expressiveness of FHM's approach was highlighted and compared to the relatively restrictive approach of Modelica.

Chapter 3 provided a brief introduction to Agda and the specialist notation used throughout this thesis. An overview of the aspects that have been mechanically formalised in this thesis was also given.

Chapter 4 considered structural properties and type systems to aid in the early detection of structural anomalies for equation-based languages. This chapter constituted the first half of two major technical contributions of this thesis. The chapter began by discussing the structural properties that are typically considered to be highly desirable when modelling physical systems. The first of these properties was then expressed in a language called H_{Δ} that lent itself to metatheoretical study. Specifically, the type system of the language was refined to include annotations on the types of signal relations that indicate the number of equations the relation is capable of *contributing*.

A more refined approach was then taken by considering a much richer set of structural properties to produce $H_{\vDash}$: a language with type-level constraints. The constraints allowed us to better approximate structural singularity detection without violating model abstractions by considering the *kinds* of equations and unknowns. The formalisation of this language was more pragmatic, and a type inference algorithm was developed to demonstrate that the constraint-based type system could have genuine practical applications. To reinforce this, the chapter concluded with a real-world case study of a half-wave rectifier model.

Chapter 5 investigated the denotational semantics of an FHM-like modelling language called $H_{[\![]\!]}$ by constructing a model and implementing an interpreter using Normalisation by Evaluation. This chapter constitutes the second half of our technical contributions. The chapter began by introducing the semantic model and normalisation procedure. The chapter then investigated various metatheoretical properties and proceeded to show a proof of normalisation, a major theorem of the semantics. The chapter then turned its focus toward investigating the semantics of dynamism. The chapter ended by considering two extensions: local variables semantics and delayed branch normalisation.

Chapter 6 discussed the work most closely related to our own.

7.2 Future Work

There are several examples in this thesis of the refined type systems being of practical use in FHM. However, it would be interesting to see to what extent these techniques can be adapted to the object-oriented approach, and then in particular to Modelica-like languages.

Modelica lacks first-class models, and the ability to track simple balance in such a setting has already been explored by Broman et al. [2006]. Therefore, the first step should be to investigate how our more sophisticated constraint-based approach can be realised in a language such as Modelica. The benefits of tracking additional structural properties in Modelica programs have already been demonstrated by Bunus and Fritzson [2002], and thus we would expect that modular detection of some of these additional properties at compile time would be both a useful and popular addition to the current Modelica standard.

Zimmer [2007] has developed Sol, a Modelica derivative with first-class models and variable structure. Our work on refined type systems was developed with language features such as these specifically in mind. Thus, it would also be interesting to see how, initially, the simple balance approach, and then subsequently, the constraint-based approach could each be applied to Zimmer's framework.

Another important consideration is the usability of the refined type systems. From the perspective of translating a model into a program, full type inference means the modeller need not be concerned with annotating (or even understanding) the constraints at work in the background. However, it is then unclear how best to communicate type errors resulting from unsatisfiable constraints to the modeller. While simple examples might result in obvious structural invariants being violated, desugaring of higher-level syntactic features may cause equation systems to become unrecognisable to the modeller. In such instances, the work by Bunus and Fritzson [2002] may prove useful in tracking the surface-level meaning of programs through syntactic transformations, allowing errors to be communicated in a more meaningful way.

Further investigation into additional or alternative structural constraint criteria may also be worthwhile. For example, a piecewise-continuous equation system may need to re-initialise the unknowns of the system at each discrete event in order to preserve continuity assumptions (i.e. the value of a signal may not be allowed to jump discontinuously). Hence, in addition to the standard equations that describe the time-varying behaviour of the system, the system also uses *instantaneous equations* to describe initialisation constraints. In such circumstances it would be incorrect to include these initialisation equations as part of a signal relation's contribution, and these equations would be considered a new *kind* of equation. Moreover, there might then be new structural constraints specific to initialisation equations, such as checking that every local unknown is re-initialised in each structural configuration.

An important and interesting avenue of future work would be to investigate how our type system can be extended to operate on objects of higher dimensions. At present, all local variables are assumed to be dimensionless, and thus effectively account for a single unknown. By introducing higher-dimension objects, such as vectors and matrices (i.e. with dimensions of 1 and 2, respectively), one would need to reason about the size of these objects during type checking. This is precisely the kind of problem that has been solved by dependent types. In particular, Dependent ML allow the sizes of objects to be computed and checked entirely automatically Xi [2007]. Thus, it would be valuable to see how this work on dependent types could be translated into our setting.

There are numerous avenues of future work for the semantic model. A particularly interesting avenue might be to explore a semantic core language that supports recursively-defined signal relations. Such a language would allow models exhibiting *unbounded* structural dynamism to be defined. For example, a perpetually bouncing ball can be modelled as a pair of mutually recursive signal relations: one relation is used for each direction of motion and discrete events switch between them when the ball touches the floor or reaches its apex. Work by Capretta [2005] (see also [Bove and Capretta, 2003]) and Danielsson [2012] could provide a useful starting point for the investigation. Depending on how one chooses to model recursive signal relations, we envisage that the semantics of dynamism could be restated to allow an infinite number of configurations to be described using a cotree. This may then resolve the incompatibility between the dynamism semantics and delayed branch evaluation.

Another avenue is to show further metatheoretical results for the semantic core language. Obvious candidates for study would be to show that the semantics objects are the canonical representation for their equivalence class, and to demonstrate some overall correctness properties for the local variables extension. Unfortunately, as discussed by Danielsson [2006], the technique used to prove normalisation may complicate a proof of object canonicity, and thus an alternative approach might need to be considered. In either case, proving this property is likely to be a large, albeit important, undertaking.

Finally, a long term goal is that our semantics lead (or at least contribute) to a verified implementation of FHM. One option would be to translate the semantic formalisation into a proof assistant such as Coq [Bertot and Castéran, 2004] that is capable of erasing proof objects and extracting a relatively efficient OCaml implementation [Leroy et al., 2013].

Bibliography

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *TCS: Theoretical Computer Science*, 342, 2005.

Andreas Abel. Miniagda: Integrating sized and dependent types. *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010*, pages 14–28, 2010.

Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, 2007.

Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program*, 22(1):9–30, 2012.

Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS-01)*, pages 303–310, Los Alamitos, CA, June 16–19 2001. IEEE Computer Society.

Edward J. Barbeau. *Pell's Equation, Problem Books in Mathematics.* Springer-Verlag, 2003.

D. A. Van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and consistent equation semantics of Hybrid Chi. *The Journal of Logic and Algebraic Programming*, 68:129–210, June 2006.

Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 203–211, 1991.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

Mathieu Boespflug. Efficient normalization by evaluation. In *The 2009 Workshop on Normalization by Evaluation*. HAL - CCSD, 2009.

Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. Submitted to Mathematical Structures in Computer Science, 2003.

Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

David Broman. *Meta-Languages and Semantics for Equation-Based Modeling and Simulation*. PhD thesis, Department of Computer and Information Science, Linkping University, Sweden, 2010.

David Broman and Peter Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 59–69, Paphos, Cyprus, 2008.

David Broman and Henrik Nilsson. Node-based connection semantics for equation-based object-oriented modeling languages. In Claudio V. Russo and

Neng-Fa Zhou, editors, *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, volume 7149 of *Lecture Notes in Computer Science*, pages 258–272. Springer, 2012.

David Broman and Jeremy G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical report, EECS Department, University of California, Berkeley, Jun 2012.

David Broman, Kaj Nyström, and Peter Fritzson. Determining over- and under-constrained systems of equations using structural constraint delta. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 151–160. ACM, 2006.

Peter Bunus and Peter Fritzson. A debugging scheme for declarative equation based modeling languages. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002)*, volume 2257 of *Lecture Notes in Computer Science*, pages 280–298, OR, USA, January 2002. Springer-Verlag.

Rod M. Burstall. Christopher strachey - understanding programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):51–55, 2000.

John 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, USA, May 2010.

John Capper and Henrik Nilsson. Towards a formal semantics of structurally dynamic non-causal modelling language. In *Proceedings of the 7th Workshop On Types in Language Design and Implementation*, Philadelphia, USA, 2012.

John Capper and Henrik Nilsson. Structural types for systems of equations: Type refinements for structurally dynamic first-class modular systems of equations. In *Higher-order and Symbolic Computation*, 2013.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

Venanzio Capretta and Amy Felty. Higher-order abstract syntax in type theory. In S. Barry Cooper, Herman Geuvers, Anand Pillay, and Jouko Väänänen, editors, *Logic Colloquium 2006*, volume 32 of *Lecture Notes in Logic*, pages 65–90. Cambridge University Press, 2009.

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, Mierlo, The Netherlands*, pages 53–64, 1996.

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

James Maitland Chapman. *Type checking and normalisation.* PhD thesis, Department of Computer Science, Nottingham University, Nottingham, UK, July 2009.

Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006.

Nils Anders Danielsson. Operational semantics using the partiality monad. *In Proceedings of the 17th ACM SIGPLAN international conference on Functional programming (ICFP 2012)*, 47(9):127–138, September 2012.

Olivier Danvy. Type-directed partial evaluation. In *In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 242–257, St. Petersburg, Florida, January 1996. ACM Press.

Nicolaas G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 137–192. Springer, 2000.

Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, 1978.

Marcelo Fiore. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-02)*, pages 26–37, New York, October 2002. ACM Press.

Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 06)*, volume 41, 1 of *ACM SIGPLAN Notices*, pages 245–256, New York, January 2006. ACM Press.

Daniel Fridlender and Miguel Pagano. A type-checking algorithm for martin-Löf type theory with subtyping based on normalisation by evaluation. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume 7941 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2013.

Sébastien Furic. Enforcing model composability in Modelica. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009*, volume 43 of *Linköping Electronic Conference Proceedings*, pages 868–879. Linköping University Electronic Press, 2009.

George Giorgidze. *First-class Models: On a Noncausal Language for Higher-order and Structurally Dynamic Modelling and Simulation.* PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2011.

George Giorgidze. *First-class models: on a noncausal language for higher-order and structurally dynamic modelling and simulation.* Nonpeerreviewed, Department of Computer Science, Nottingham, July 2012.

George Giorgidze and Henrik Nilsson. Higher-order non-causal modelling and simulation of structurally dynamic systems. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009*, volume 43 of *Linköping Electronic Conference Proceedings*, pages 208–218. Linköping University Electronic Press, 2009.

George Giorgidze and Henrik Nilsson. Mixed-level embedding and JIT compilation for an iteratively staged DSL. In Julio Mariño, editor, *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, volume 6559, pages 48–65. Springer-Verlag, 2011.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logics in Computer Science (LICS 1996)*, pages 278–292, 1996.

William A. Howard. The formulae-as-type notion of construction, 1969. *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490, 1980.

Paul Hudak. Functional reactive programming. *Lecture Notes in Computer Science*, 1576, 1999.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, UK, 2007.

Jun Inoue and Walid Taha. Reasoning about multi-stage programs. In *Programming Languages and Systems – 21st ESOP'12 (Part of 15th ETAPS'12)*, volume 7211 of *Lecture Notes in Computer Science (LNCS)*, pages 357–376. Springer-Verlag (New York), Tallinn, Estonia, March 2012.

Brian W. Kernigham and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

Oleg Kiselyov. Functional style in C++: closures, late binding, and Lambda abstractions. *ACM SIGPLAN Notices*, 34(1):337–337, January 1999.

David Kågedal. A natural semantics specification for the equation-based modeling language Modelica. Master's thesis, PELAB, Department of Computer and Information Science, Linköping University, October 1998.

David Kågedal and Peter Fritzson. Generating a Modelica compiler from natural semantics specifications. In *The 1998 Summer Computer Simulation Conference (SCSC'98)*, Reno, Nevada, U.S.A, July 1998.

Edward A. Lee. Cyber physical systems: Design challenges. In *ISORC*, pages 363–369. IEEE Computer Society, 2008.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.01: Documentation and user's manual, September 2013.

Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In Gabriele Keller, editor, *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*, pages 73–82. ACM, 2007.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

Yuri V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, Cambridge, Massachusetts, 1993.

Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, LFCS, University of Edinburgh, Edinburgh, Scotland, 2000.

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.

Conor Mcbride. Ornamental algebras, algebraic ornaments. In *Journal of Functional Programming*, December 2011.

Conor McBride. Agda-curious?: an exploration of programming with dependent types. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 1–2. ACM, 2012.

Conor Thomas McBride. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 297–309, New York, NY, USA, 2014. ACM.

Robin Milner. A theory of type polymorphism in programming. *JCSS: Journal of Computer and System Sciences*, 17, 1978.

Modelica — A Unified Object-Oriented Language for Systems Modelling; Language Specification Version 3.3. Modelica Association, May 2012.

Pieter J. Mosterman and Gautam Biswas. Formal specifications for hybrid dynamical systems. In *IJCAI*, pages 568–577. Morgan Kaufmann, 1997.

Henrik Nilsson. Functional automatic differentiation with dirac impulses. *In Proceedings of the Eight ACM SIGPLAN International Conference on Functional Programming*, 38(9):153–164, 2003.

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*, Linköping Electronic Conference Proceedings, pages 71–81, Paphos, Cyprus, July 2008. Linköping University Electronic Press.

Henrik Nilsson and George Giorgidze. Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*, Prague, Czech Republic, September 2010. Czech Technical University Publishing House.

Henrik Nilsson, John Peterson, and Paul Hudak. Functional hybrid modeling. In *Proceedings of PADL'03: 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.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2007.

Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

Christoph Nytsch-Geusen, Thilo Ernst, Andre Nordwig, Peter Schwarz, Peter Schneider, Matthias Vetter, Christof Wittwer, Thierry Nouidui, Andreas Holm, Jurgen 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*, Hamburg, Germany, 2005.

Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, March 1988.

Peter Pepper, Alexandra Mehlhase, Christoph Höger, and Lena Scholz. A compositional semantics for modelica-style variable-structure modeling. In *4th International Conference on Equation-based Object-oriented Modelling Languages and Tools*, ETH Zürich, Switzerland, 2011.

Simon Peyton Jones. Special issue: Haskell 98 language and libraries. *Journal of Functional Programming*, 13, January 2003.

Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 199–208, June 1988.

Benjamin Pierce. *Types and Programming Languages*. The MIT Press, 2002.

William Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing 91*, 1991.

Erik Saaman and Grant Malcolm. Well-founded recursion in type theory. Computing science notes, Department of Mathematics and Computer Science, University of Groningen, 1987.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *ACM SIGPLAN Notices*, 43(9):51–62, September 2008.

Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proc. Symp. Computers and Automata*. Polytechnic Inst. of Brooklyn Press, 1971. Also Technical Monograph PRG-6, Programming Research Group, Oxford University.

Michael L. Scott. *Programming Language Pragmatics (3. ed.)*. Academic Press, 2009. ISBN 978-0-12-374514-9.

Neil Sculthorpe. *Towards safe and efficient functional reactive programming.* PhD thesis, Department of Computer Science, Nottingham University, Nottingham, UK, July 2011.

Raymond Serway. *Physics for scientists and engineers.* Thomson-Brooks/Cole, Belmont, CA, 2004.

Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings, Scheme and Functional Programming Workshop 2006*, pages 81–92. University of Chicago TR-2006-06, 2006.

Simulink. *SIMULINK - User's Guide.* MathWorks, Inc., Cochinate Place, 24 Prime Park Way, Natick, MA, 1992.

John Slaney. MINLOG: A minimal logic theorem prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 268–271, Berlin, July 1997. Springer.

Walid Taha, Paul Brauner, Yingfu Zeng, Robert Cartwright, Veronica Gaspes, Aaron Ames, and Alexandre Chapoutot. A core language for executable models of cyber-physical systems (preliminary report), June 2012.

Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, July 1996.

Rene Vestergaard. Normalisation by evaluation for system F using staged outermost reduction, June 2000.

Philip Wadler. *Monads for Functional Programming.* Prentice Hall, 1993.

Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, pages 242–252, June 2000.

Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. *ACM SIGPLAN Notices*, 36(10):146–156, October 2001.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. URL `citeseer.nj.nec.com/wright92syntactic.html`.

Hongwei Xi. Dependent ML: An Approach to Practical Programming with Depedent Types. *Journal of Functional Programming*, 2(17):215–286, 2007.

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

Dirk Zimmer. Enhancing modelica towards variable structure systems. In Peter Fritzson, François E. Cellier, and Christoph Nytsch-Geusen, editors, *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, EOOLT 2007, Berlin, Germany, July 30, 2007*, volume 24 of *Linköping Electronic Conference Proceedings*, pages 61–70. Linköping University Electronic Press, 2007.

Dirk Zimmer. Towards improved class parameterization and class generation in modelica. In Peter Fritzson, Edward A. Lee, François E. Cellier, and David Broman, editors, *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2010, Oslo, Norway, October 3, 2010*, volume 47 of *Linköping Electronic Conference Proceedings*, pages 33–42. Linköping University Electronic Press, 2010.

Dirk Zimmer. A new framework for the simulation of equation-based models with variable structure. *Simulation*, 89(8):935–963, 2013.