Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

Structural Types for Systems of Equations

Type Refinements for Structurally Dynamic First-Class Modular Systems of Equations

John Capper · **Henrik Nilsson**

Received: date / Accepted: date

Abstract Characterising a problem in terms of a system of equations is common to many branches of science and engineering. Due to their size, such systems are often described in a modular fashion by composition of individual equation system fragments. Checking the balance between the number of variables (unknowns) and equations is a common approach to early detection of mistakes that might render such a system unsolvable. However, current approaches to modular balance checking have a number of limitations. This paper investigates a more flexible approach that makes it possible to treat equation system fragments as true first-class entities. Furthermore, the approach handles so-called structurally dynamic systems, systems whose behaviour changes discretely and abruptly over time. The central idea is to record balance information in the type of an equation fragment. This information can then be used to determine if individual fragments are well formed, and if composing fragments preserves this property. The type system presented in this paper is developed in the context of Functional Hybrid Modelling (FHM). However, the key ideas are in no way specific to FHM, but should be applicable to any language featuring a notion of modular systems of equations, including systems with first-class components and structural dynamism.

Keywords Systems of equations · First-class components · Non-causal, structurally dynamic modelling · Structural analysis · Linear constraints · Refinement types.

1 Introduction

Systems of equations, also known as simultaneous equations, are abundant in science and engineering. Applications include modelling, simulation, and optimisation, to name but a few. Describing complex problems mathematically, e.g. modelling the engine of a car, often requires a large number of equations; systems consisting of hundreds of thousands of equations are not uncommon. The systems are usually parametrised, describing not just a specific problem instance, but a set of problems. Moreover, it is more of a rule than an

J. Capper & H. Nilsson
Functional Programming Lab
School of Computer Science
University of Nottingham, UK
E-mail: {jjc,nhn}@cs.nott.ac.uk

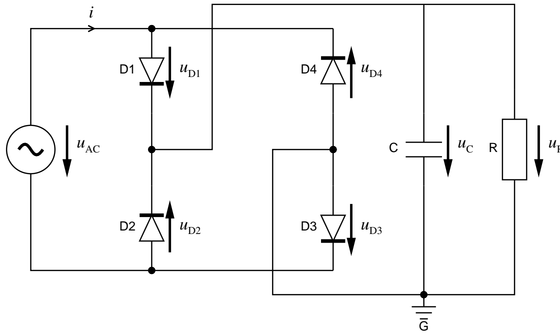

Fig. 1: Full-wave rectifier

exception that no known analytical solution exists, necessitating the use of computers and numerical methods for finding (sufficiently good approximate) solutions. Due to the size of the equation systems, some form of abstraction mechanism that supports a *modular* formulation by composition of individual equation system fragments, typically referred to as system *components*, is often a practical necessity as well.

As an example, consider the full-wave rectifier in Fig. 1. This is not a very big system, consisting of only eight components. Yet, the advantages of being able to derive a system of equations that model this circuit by reusing parametrised models of the individual components should be clear. For instance, a single diode model could be reused four times, for diodes D1–D4. More generally, physically accurate component models can be very involved, making it highly desirable to develop libraries of reusable components.

A number of high-level *equation-based* languages exist across a range of application domains, supporting a modular, parametrised formulation of systems of equations. These languages are supported by an environment that include appropriate approaches for solving the equations given specific values for the parameters, or for solving optimisation or parameter fitting problems. Examples in the area of modelling and simulation of physical systems, languages which could be used to model circuits like the one in Fig. 1, include Modelica [21], VHDL-AMS [16] and Verilog-AMS [1].

In modern, high-level programming languages, *types* play a crucial role. Types aid in creating *safe* programs that conform with their specification. The *power* of a type system can vary greatly, ranging from languages such as C [2], in which types make very few guarantees about correctness, to languages such as Agda [26], where types can be used to specify very precise correctness criteria. In particular, Agda employs dependent types [19] that allow the programmer to not only specify the properties of programs, but also to prove these properties within the language itself.

Equation-based languages are often also typed, with the types playing much the same roles as in conventional programming languages. Additionally, simple invariants related to the structure of the equation systems may be enforced, such as there being equally many equations as variables to solve for, with a view to static detection of structural problems that are likely to render the systems ill-formed and thus unsolvable.

However, there is considerable scope for improving the type systems of current equation-based languages in the latter respect, both in terms of refining the enforced structural invariants, thus allowing more potential problems to be detected early, and to generalise this to

a setting where equation system fragments have *first-class* status and where the systems of equations as such may be *structurally dynamic*: evolving over time. Current, main-stream, equation-based languages have quite limited support for structural dynamism, and making these languages more flexible in that respect is an active research area [25,27,14,32]. This work has thus far focused on constructs and mechanisms for expressing and solving structurally dynamic systems, while little attention has been paid to finding suitable structural invariants for this more general setting.

In the following, we seek to address the above points. We develop a type system for modular systems of equations that enforces a refined set of structural invariants while supporting first-class components and structural dynamism. We treat equations in the abstract in our formal development, not assuming any specific application domain. However, we do need a concrete language for expressing modular, structurally dynamic systems of equations. To that end, we develop a core equation-based language that captures the essence of Functional Hybrid Modelling (FHM) [25,14]. FHM is a framework for modelling physical systems that can be described by differential equations, such as the full-wave rectifier above. For this reason, most of our examples will also be drawn from physical systems modelling, with the unknowns in the equations representing time-varying entities (functions of time). FHM was chosen as it features component-based modelling, first-class models, and structural dynamism in a relatively minimalistic way. However, we reiterate that the core ideas put forward in this paper are not at all limited to FHM. Our specific contributions are:

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.

We have also implemented a type checker for our system capable of inferring types [6]. It is implemented in the dependently typed language Agda [26] primarily to ensure totality and termination of the inference algorithm, but also with a view to facilitate formal verification of aspects of the type system at a later stage.

The remainder of this article is structured as follows. Sections 2 and 3, respectively, introduces modular systems of equations and FHM. Section 4 investigates structural properties of equation systems, setting the stage for Sect. 5 that presents the main technical contribution of this article: the core language, its semantics, the type system, and implementation notes. Section 6 evaluates what has been achieved, in part through a complete worked example. Related work is considered in Sect. 7, while avenues for future work are discussed in Sect. 8. Finally, Section 9 concludes.

2 Modular Systems of Equations

This section gives a detailed introduction to modular systems of equations, covering basic theory, modularity, abstraction over systems, causality, and structural dynamism.

2.1 Preliminaries

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. The domain of the variables and signatures for equations is mostly orthogonal to the work presented in this article. However, for reasons of presentation, we will use the domains of reals or time-varying reals unless stated otherwise. The following is an example of a system consisting of two equations and two unknowns:

$$x^2 + y = 0 \tag{1}$$
$$3x = 10 \tag{2}$$

This system can be solved by using (2) to solve for x, substituting the value of x into (1), thus enabling the latter to be used to solve for y. However, consider the following parametrised version of the system instead. The solvability of the system now depends on the value of the coefficient c: when $c = 0$, no solution exists.

$$x^2 + y = 0 \tag{3}$$
$$cx = 10 \tag{4}$$

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.

Yet, when building systems of equations modularly, it is desirable to catch problems that may ultimately lead to unsolvable systems of equations early; i.e., already at the level of individual equation system fragments or partial compositions of fragments. One property that can be checked modularly is whether the number of equations and unknowns agree. While an equal number of equations and unknowns in itself is neither a necessary nor sufficient condition for solvability, an unequal number is in practice often indicative of problems. Consequently, rules pertaining to the balance between the number of equations and unknowns are adopted in industrial-strength, equation-based languages such as Modelica [21].

We will discuss structural properties in more depth in Sect. 4. In particular, we will identify invariants that are more refined than basic equation and variable balance, thus allowing checking for a larger class of structural problems, while still admitting modular checking in a setting with first-class equation system fragments and structural dynamism. In Sect. 5 we will then proceed to show how these refined invariants can be integrated into a type system.

2.2 Abstraction over Systems of Equations

The equation systems needed to describe real-world problems are usually large and complex. On the other hand, there tends to be a lot of repetitive structure [10], making it beneficial to describe the systems in terms of reusable equation system fragments. For example, consider an electrical circuit comprising resistors, capacitors, and inductors. 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.

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:

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

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.

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. We express this as *relation application*, denoted by $\diamond$:

$$\begin{aligned} u+v+w &= 10 \\ rel &\diamond (u,v) \\ rel &\diamond (v,w+7) \end{aligned}$$

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

$$\begin{aligned} u+v+w &= 10 \\ u+v+z_1 &= 0 \\ u-z_1 &= 1 \\ v+(w+7)+z_2 &= 0 \\ v-z_2 &= 1 \end{aligned}$$

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 *rel*.

2.3 Causality

A *causal* system is one in which the cause-and-effect relationship between variables is explicit. In other words, the equations are *directed*: the equations are solved in a given order, with known (at that point) variables on one side of the equal sign, and a single unknown on the other (which then becomes a known in subsequent equations). One example is a set of ordinary differential equations (ODEs) in explicit form where the system state at a point in time is considered known, enabling the state derivatives at that point in time to be computed (from which the system state at the “next” point in time can then be approximated). Conversely, an *acausal* system is *undirected* with the equations simply expressing a relation on the variables, without any á priori given inputs and outputs, and thus also without any á priori given strategy for solving the equations. Differential Algebraic Equations (DAEs) [10] are an important example of acausal equations in the domain of modelling and simulation.

As a concrete example, consider Pell’s equation [3] over the two unknowns x and y and parametrised by an integer n:

$$x^2 - ny^2 = \pm 1$$

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

$$y := \sqrt{(x^2 \pm 1)/n} \qquad\qquad x := \sqrt{\pm 1 + ny^2}$$

The advantage of the acausal formulation is that the equations are more reusable (above, Pell’s equation is used in two ways) and more declarative (the equations can be expressed in whatever way is most clear, without undue concerns about how they are going to be solved).

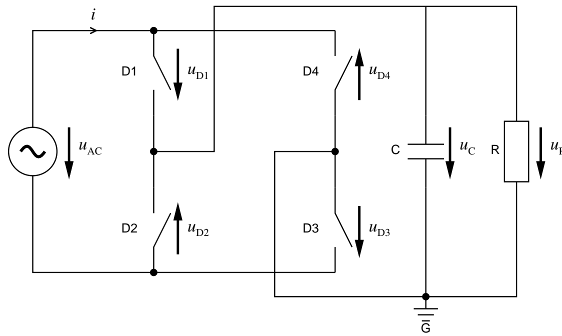

Fig. 2: Full-wave rectifier modelled using ideal diodes.

These points are crucial advantages in many domains, including modelling and simulation of large-scale physical systems [9]. A number of successful acausal modelling languages have thus been developed, with Modelica [21] being a prominent, state-of-the-art example.

2.4 Structural Dynamism

In a temporal setting, where equations express relations among time-varying entities, the equations themselves may change at various points in time to capture changes in the system configuration. A system of equations that evolve over time is known as *structurally dynamic*.

As an example, consider the model in Fig. 2 of the full-wave rectifier from Fig. 1 [24]. 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.

Structurally dynamic systems offer greater expressivity, but also create many problems [25, 27, 14, 32]. Of particular concern in this article is that errors in a system with a large or possibly even unbounded number of configurations may take a very long time to surface if this error only manifests itself when specific system configurations become active. Thus, the larger the class of such errors that can be caught statically by enforcing suitable structural invariants, the better. We consider invariants for structural dynamism in Sect. 4.3.

3 Functional Hybrid Modelling and Hydra

We now turn our attention to a particular approach to acausal, equation-based languages for modelling and simulation of physical systems, Functional Hybrid Modelling (FHM), as this provides a useful, concrete setting for our work. *Hydra* [25, 14] is an example of an FHM language. Its syntax will be adopted until a formal core language is presented in Sect. 5.2.

3.1 Functional Reactive Programming

FHM is inspired by Functional Reactive Programming (FRP) [12, 31], in particular as embodied by *Yampa* [23]. FHM can be viewed as a generalisation of Yampa, hence, we will

begin by introducing the two central concepts of Yampa: *signals* and *signal functions*. Conceptually, a signal is a time-varying value: i.e: a function from time to a value of type τ:

$$\begin{aligned} \mathit{Time} &\approx \mathbb{R}^+ \\ \mathit{Signal}\ \tau &\approx \mathit{Time} \rightarrow \tau \end{aligned}$$

Time is represented as $\mathbb{R}^+$, the continuous, non-negative real numbers. The type of the time-varying value is represented by τ. A *signal function* is then simply a function on signals:

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

The $\approx$ symbol is used to emphasise the conceptual nature of the above definitions, which are not used directly in the implementation of Yampa. In particular, signal functions are required to be *temporally causal*: the output of a signal function at time t may only depend on the input signal on the interval $[0,t]$. Despite this, the conceptual definitions are useful for developing an intuition about the semantics of both FRP and FHM.

3.2 Signal Relations

The above definitions of signals and signal functions describe systems that are inherently causal. Signal functions are *directed*, taking an input signal and returning an output signal. FHM generalises signal functions to *signal relations*. Signal relations are acausal: they do not distinguish between inputs and outputs but rather express how signals are related to one another. A signal relation may be viewed as a predicate on a signal:

$$\mathit{SR}\ \tau \approx \mathit{Signal}\ \tau \rightarrow \mathit{Prop}$$

To recover n-ary relations, one can observe the following isomorphism: $\forall\, \alpha\, \beta\, .\, \mathit{Signal}\ \alpha \times \mathit{Signal}\ \beta \simeq \mathit{Signal}\ (\alpha \times \beta)$. Unary signal relations thus suffice for representing n-ary relations. For example, given a binary predicate $(\equiv) : \mathbb{R} \times \mathbb{R} \rightarrow \mathit{Prop}$, the following binary signal relation can be constructed:

$$\begin{aligned} &(\equiv_{sr}) : \mathit{SR}\ (\mathbb{R},\mathbb{R}) \\ &(\equiv_{sr})\ s \approx \forall\, t : \mathit{Time}\ .\ (\equiv)\ (s\ t) \end{aligned}$$

3.3 Hydra

Hydra [25, 14] is a functional hybrid modelling language embedded in the functional programming language Haskell [17]. There are two levels to FHM and thus to Hydra: the *functional level*, concerned with defining ordinary functions operating on time-invariant values, and the *signal level*, concerned with the definition of relations between signals (time-varying values), and, indirectly, the definition of the signals themselves as solutions satisfying the constraints imposed by the signal relations. The 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.

Signal relations are constructed using the **sigrel** primitive, marking the boundary between the two levels:

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

Signal relations are first-class, time-invariant, function-level objects that encapsulate a set of *equations*. These equations range over signal variables introduced by the *pattern*, similar to the abstraction mechanism presented in Sect. 2.2. A pattern is a (possibly nested) tuple (e.g. $((x,y),z)$ constitutes a valid pattern introducing 3 signal variables). 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. Signals are *not* first class entities in the language. There are two basic forms of equations:

atomic equation: $e_1 = e_2$
signal relation application: $sr \diamond e_3$

Here, *sr* is a *time-invariant* expression (signal variables must not occur in it) denoting a signal relation, and $\diamond$ denotes signal relation application. To illustrate, consider a component *twoPin* encapsulating equations common to all electrical components with two pins:

$$
\begin{aligned}
&\textit{type Pin} = (\mathbb{R},\mathbb{R})\\
&\textit{twoPin} : \textit{SR}\ (\textit{Pin},\textit{Pin},\textit{Voltage})\\
&\textit{twoPin} = \textbf{sigrel}\ (p,n,u)\ \textbf{where}\\
&\quad p.i + n.i = 0\\
&\quad p.v - n.v = u
\end{aligned}
$$

Pin is a pair of values representing an electrical junction with projections for current and voltage. For clarity, we allow ourselves to use dot-notation for the projections; e.g. $p.i$ for the current through pin p and $p.v$ for the voltage (potential) at pin p. We can now define models of concrete electrical components by adding equations to the basic *twoPin*-model. For example, a model of a resistor:

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

The *resistor* component extends *twoPin* by adding an equation that describes the behaviour of a resistor. The component shows how a system can be parametrised by a coefficient, in this example by the parameter r. The syntax **local** has been adopted to make explicit the quantification of the local variable u, and distinguish it from r, which is a time-invariant, functional-level parameter.

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. Here, the keyword **der** indicates the time derivative of a signal:

$$
\begin{aligned}
&\textit{inductor} : \textit{Inductance} \rightarrow \textit{SR}\ (\textit{Pin},\textit{Pin})\\
&\textit{inductor}\ i = \textbf{sigrel}\ (p,n)\ \textbf{where}\\
&\quad \textbf{local}\ u
\end{aligned}
$$

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

Elaboration of Hydra models proceeds by substitution of variables under signal relation application, resulting in either a flat set of equations, or a λ-expression. To illustrate, consider the modular system of equations in the relation *resistor 220*:

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

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*.

$$
\begin{array}{l}
p.i \;\; + n.i \;\; = \; 0 \\
p.v \;\; - n.v \; = \; u \\
220 * \;\; p.i \;\; = \; u
\end{array}
$$

3.4 Structural Dynamism in Hydra

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

$$
\begin{array}{l}
\mathbf{initially} \; [; \mathbf{when} \; condition] \; \Rightarrow \\
\quad equations_1 \\
\mathbf{when} \; condition \; \Rightarrow \\
\quad equations_2 \\
\ldots \\
\mathbf{when} \; condition_n \; \Rightarrow \\
\quad equations_n
\end{array}
$$

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. 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.

Additional complications arise due to the need to properly *initialise* the new system of equations after a switch. This is a hard problem in general, but it can be addressed at least to some extent by providing separate initialisation and reinitialisation equations [14, 24]. However, we will not consider this further here as this just amounts to additional systems of equations that can be subjected to much the same invariants as the main system.

For a concrete example, consider the following Hydra model of an ideal diode; i.e., essentially a voltage-controlled electrical switch [24]:

$$
\begin{array}{l}
\mathit{icDiode} \,:\, \mathit{SR}\,(\mathit{Pin}, \mathit{Pin}) \\
\mathit{icDiode} \,=\, \textbf{sigrel}\,(p, n)\ \textbf{where} \\
\quad \textbf{local}\ u \\
\quad \mathit{twoPin} \diamond (p, n, u) \\
\quad \textbf{initially}; \textbf{when}\ p.v - n.v > 0 \Rightarrow \\
\qquad u \,=\, 0 \\
\quad \textbf{when}\ p.i < 0 \Rightarrow \\
\qquad p.i \,=\, 0
\end{array}
$$

Whenever the voltage across the diode becomes positive, the diode starts conducting, meaning the switch closes resulting in the voltage across the diode becoming zero. Conversely, whenever the current starts to flow backwards through the diode, the diode stops conducting, meaning the switch opens and the current through the diode becomes zero.

4 Structural Properties

4.1 Structural Properties and Solvability

An important question regarding a system of equations is whether it has a solution and, if so, if that 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 as it would rule out checking components in isolation. Furthermore, as typical application domains, such as physical systems modelling, necessitates 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 restricting the systems of equations to be linear would be of very limited use.)

However, there are simple criteria that, while neither necessary nor sufficient for *guaranteeing* solvability, are such that violation 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 tool. Enforcing that such criteria be met through the static semantics of an equation-based language can thus be useful, and is in fact often done. The following are two commonly used criteria for checking the well-formedness of systems [4,5,7,21,22]:

1. Balanced system: the number of equations is equal to the number of variables.
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.

As we are only considering systems of finite size, 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:

$$x + y = z \tag{5}$$

$$x + 3 = 12 \tag{6}$$

$$y^2 + 9 = z^2 \tag{7}$$

The bijection $\{x \mapsto 6, y \mapsto 7, z \mapsto 5\}$ between the set of variables $\{x, y, z\}$ and the set of equations $\{5, 6, 7\}$ pairs each variable with an equation in which it occurs. This system is thus both balanced and structurally non-singular. Furthermore, as it happens, it has a solution: $x = 9$, $y = -4$, $z = 5$.

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

$$
\begin{aligned}
x &= 2 \\
x^2 + 1 &= 5
\end{aligned}
$$

This system is not even balanced. Yet $x = 2$ is clearly a solution. This shows that the above criteria are not necessary for the existence of a solution, and it is also easy to demonstrate that the criteria are not sufficient either.

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.

This latter intuition is broadly valid also 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 solution methods will definitely fail. 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, it is easy to check, and if violated, then that 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. Consequently, both criteria constitute useful static checks that can help find errors early during compilation.

In the following, we will develop criteria along the lines discussed above, but insist that they can be checked modularly, to enable integration into a type system and support first-class equation system fragments, and that they also work in a setting with structural dynamism. We will aim for a better approximation of structural non-singularity than basic balance checking by taking *some* account of which variables occur in which equations, but we will stay well short of attempting a full check for structural non-singularity, as a precise check cannot be done modularly or in a structurally dynamic setting, and as taking individual variable occurrences into account for a more precise approximation leads to a very complicated and expensive system [22]. We will refer to the difference between the number of equations and variables in a system as the *equation-variable balance*. By analogy to the terminology used for linear systems, but regardless of whether the equations are independent or even linear, we will refer to a system where the balance is positive as *overconstrained* and one where the balance is negative as *underconstrained*.

4.2 Criteria for Structurally Well-formed Signal Relations

The crux of the type system discussed in this paper is the insistence that a modular system of equations satisfy certain structural properties. This is enforced by introducing constraints at the type level (Sect. 5). We introduce 5 criteria below, stemming from the setting of FHM, from which such constraints can be generated. It is conceivable that different application

domains could require constraints specific to that domain. This is not a problem, provided the constraints are linear inequalities, as the system developed in this article is independent of the criteria used to generate constraints.

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. 3.3) 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.

The number of interface, local, and mixed equations is denoted i_Q, l_Q, and m_Q respectively. Consequently, the total number of equations $a_Q = i_Q + l_Q + m_Q$.

A signal relation is *structurally well-formed* if the following 5 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. $l_Q \geqslant 0, m_Q \geqslant 0, and\ i_Q \geqslant 0$: When considering structurally dynamic systems, we will permit negative contributions at intermediate stages, but insist that ultimately, the contribution of each equation kind should be non-negative.

To illustrate, let us return to the *resistor* example from Sect. 3.3. 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 [22]. 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 to 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.

4.3 Well-formedness and Structural Dynamism

Recall that a structurally dynamic system of equations is one where the equations are allowed to vary over time (Sect. 2.4). As FHM permits structurally dynamic systems (Sect. 3.4), 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 ascertain the well-formedness of the entire system.

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 $q_1 \ldots q_n$, 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:

6. $l \geqslant l_k \geqslant 0$: The reconciled system contributes at least as many local equations as the systems being reconciled. All local contributions must be positive.
7. $i \geqslant i_k \geqslant 0$: The reconciled system contributes at least as many interface equations as the systems being reconciled. All interface contributions must be positive.
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 interface and local variables respectively. The reason is that we then can 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. Note that these two numbers 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 $dynamism_1$:

$dynamism_1 \; : \; SR\,(\mathbb{R},\mathbb{R}) \;\to\; SR\,\mathbb{R}$
$dynamism_1\; sr \;=\;$ **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 of the switch 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:

$dynamism_2 \; : \; SR\,\mathbb{R}$
$dynamism_2 \;=\;$ **sigrel** x **where**
  local u
  initially
    $f\,x$
  when $x > 0 \;\Rightarrow$
    $g\,u$
  $h\,x\,u$

Applying the fair approach results in the following constraints: $l \geqslant 0 \geqslant 0, l \geqslant 1 \geqslant 0,$ $i \geqslant 1 \geqslant 0, i \geqslant 0 \geqslant 0, m \leqslant 0-(l-0)-(i-1), m \leqslant 0-(l-1)-(i-0), l+m+i = 1,$ $l+m+i = 1$. Simplifying the constraints yields a solution at $l = 1$, $i = 1$, $m = -1$. Thus, the switch block contributes (or in this instance *requires*) -1 mixed equations. The interpretation of the above is that the switch block may be reconciled 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:

$dynamism_3 \; : \; SR\,\mathbb{R}$
$dynamism_3 \;=\;$ **sigrel** x **where**
  local $u\;v$
  initially
    $u \quad = \; v$

$$
\begin{array}{l}
\quad f\,u\,v \;=\; 0 \\
\textbf{when}\; u + v < 0 \Rightarrow \\
\quad g\,x \;=\; 0 \\
\quad x \quad = u
\end{array}
$$

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.

5 A Structural Type System

This section constitutes the main technical contribution of the article, presenting and formalising a type system for checking structural properties of equation-based languages. The type system is developed for a modest language of equations embedded into the simply-typed λ-calculus. Such an embedding reflects the two-tiered approach also used by FHM. The type system includes polymorphic types, similar to those found in a Hindley-Milner-style polymorphic system. However, here, polymorphic types are used to describe systems that are somehow *flexible* in their equation structure. Fig. 3 gives the notational conventions used throughout the remainder of this section.

Description	Symbol
t	λ-terms
v	λ-values
x, y, z	λ-variables
Γ	contexts
q	equation terms
qv	equation values
sw	switch blocks
sv	switch values
u, v, w	local signal variables

Description	Symbol
τ	functional monotypes
σ	functional polytypes
ν	equation types
μ	constraint equation types
k	kinds
n, m, o	balance variables
C, D, E	constraints
c, d	constraint expressions

Fig. 3: Notational Conventions

5.1 Key Ideas

The fundamental idea behind the type system is to refine the type of a signal relation by including structural information. Specifically, a *balance* is associated with each signal relation type to indicate the number of equations that a signal relation *contributes* when used as a component of a larger system of equations.

FHM is a language for higher-order modelling, permitting equation systems to appear as parameters. As a result, the structural information required to compute an exact contribution

is not necessarily known à priori. Hence signal relation types are generally parametric in their balance, through a *balance variable*, allowing for polymorphic signal relations that contribute a varying number of equations depending on the usage context.

However, the contribution of a signal relation (represented by a balance variable) is subject to *balance constraints* (simply *constraints* from now on) dictated by the structural criteria from Sect. 4. The context in which a signal relation is applied is used to generate the constraints, which must be satisfiable for a type to be valid. The constraints restrict the contribution of a component to a contiguous interval. Constraints may involve the contributions of several components, and are thus not directly associated with a single signal relation in general, as will become clear in the following.

To illustrate, consider the refined type for *resistor* from Sect. 3.3. We adopt a syntax similar to Haskell's for type class constraints to express constraints on balance variables:

$$\mathit{resistor} : (n = 2) \Rightarrow \mathit{Resistance} \rightarrow \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\;n$$

Here, the balance variable n is constrained to the value 2. This can be verified by first flattening the signal relation applications to obtain a set of 3 equations over 5 variables (note that each *Pin* contains two variables), then removing one equation which must be used to solve for the local variable u, giving a net contribution of two equations.

Note that a representation of expressions containing integers and linear inequalities has been introduced at the type level. This extension may appear to be a restricted form of dependent types [19]. However, these type level representations, whilst determined by the structure of terms, are not value-level terms themselves. Hence, we do not consider our system to be dependently typed.

At this point, it is useful to consider the meaning of constrained types. Intuitively, $t : C \Rightarrow \tau$ means that if it is possible to find a valuation for all balance variables such that the constraints in C are satisfied, then τ is a valid type for the term t. As an example, consider the contrived type:

$$(2 \leqslant m \leqslant 4, 3 \leqslant n \leqslant 5) \Rightarrow \mathit{SR}\;m \rightarrow \mathit{SR}\;n$$

This can be viewed as the type of a signal relation parametrised on a signal relation. The following types are all examples of valid instances of the above:

$$\begin{array}{ll} (m = 2, n = 3) & \Rightarrow \mathit{SR}\;m \rightarrow \mathit{SR}\;n \\ (m = 4, n = 4) & \Rightarrow \mathit{SR}\;m \rightarrow \mathit{SR}\;n \\ (m = 3, 4 \leqslant n \leqslant 5) & \Rightarrow \mathit{SR}\;m \rightarrow \mathit{SR}\;n \end{array}$$

An appropriate way to think of a signal relation parametrised on signal relations is that it is required to *accept* relations with any specific balances as long as the associated constraints are satisfied, and that once applied to relations with appropriate balances is capable of *contributing* a number of equations within bounds defined by the remaining constraints.

The power and utility of the type refinements can be seen in the examples given below. The examples define higher-order combinators over two-pinned circuit components, giving rise to refined types that are parametric in the resultant contribution. The combinators define serial and parallel composition of two-pinned circuit components. The visual representation in Fig. 4 shows the topological structure that is captured by the two combinators. In both cases, 4 new pins, p_1, p_2, n_1, and n_2 are created *internally* to wire together the components, where the naming conventions p and n refer to the positive and negative pins of a component, respectively. In Hydra, the internal pins correspond to new local variables. The equations in

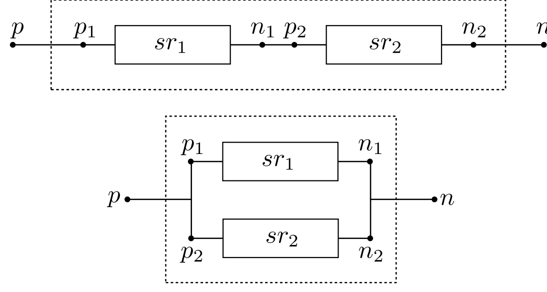

Fig. 4: Serial (top) and parallel (bottom) composition of two-pinned circuit components.

parallel and *serial* are then just applications of the subcomponents along with Kirchhoff's first and second laws for electrical circuits [18]:

$$
\begin{array}{l}
\mathit{parallel} : \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \rightarrow \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \rightarrow \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \\
\mathit{parallel}\,\mathit{sr}_1\ \mathit{sr}_2\ = \\
\quad \textbf{sigrel}\ (p,n)\ \textbf{where} \\
\qquad \textbf{local}\ p_1\ p_2\ n_1\ n_2 \\
\qquad \mathit{sr}_1 \diamond (p_1,n_1) \\
\qquad \mathit{sr}_2 \diamond (p_2,n_2) \\
\qquad p.i + p_1.i + p_2.i = 0 \\
\qquad n.i + n_1.i + n_2.i = 0 \\
\qquad p.v \ \ = p_1.v \\
\qquad n.v \ \ = n_1.v \\
\qquad p_1.v = p_2.v \\
\qquad n_1.v = n_2.v
\end{array}
$$

$$
\begin{array}{l}
\mathit{serial} : \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \rightarrow \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \rightarrow \mathit{SR}\,(\mathit{Pin},\mathit{Pin}) \\
\mathit{serial}\,\mathit{sr}_1\ \mathit{sr}_2\ = \\
\quad \textbf{sigrel}\ (p,n)\ \textbf{where} \\
\qquad \textbf{local}\ p_1\ p_2\ n_1\ n_2 \\
\qquad \mathit{sr}_1 \diamond (p_1,n_1) \\
\qquad \mathit{sr}_2 \diamond (p_2,n_2) \\
\qquad -p.i + p_1.i = 0 \\
\qquad n_1.i \ \ + p_2.i = 0 \\
\qquad n_2.i \ \ - n.i \ \ = 0 \\
\qquad p.v \ \ = p_1.v \\
\qquad n_1.v = p_2.v \\
\qquad n_2.v = n.v
\end{array}
$$

Upon applying the constraint criteria from Sect. 4 to compute the refined types, it transpires that parallel and serial share the same constraints, and thus, the same refined type. This give us some reassurance that the type system is imposing sensible constraints: the two circuits are connected in different ways, with different equations describing the composition, but in both cases we arrive at the same "composition constraints":

$parallel, serial : (o = n + m - 2, 0 \leqslant o \leqslant 4) \Rightarrow$
$SR\ (Pin, Pin)\ n \rightarrow SR\ (Pin, Pin)\ m \rightarrow SR\ (Pin, Pin)\ o$

Note that in both cases there are 8 local variables (as each pin constitutes two variables) and 6 atomic equations that can be used to solve for them, meaning that the two signal relation applications at least have to contribute two more equations.

The constraints are also parametric in *n*, *m*, and *o*. Hence, they may be safely instantiated to any set of values satisfying these constraints, for example: $\{n \mapsto 1, m \mapsto 2\}$ or $\{n \mapsto 3, m \mapsto 3\}$. For further reassurance, consider using *parallel* to compose two resistors. The refined type of the composition admits the same constraints as the resistor component in isolation, i.e. the composition of two resistors is, as expected, just another resistor:

$parRes : (n = 2) \Rightarrow SR\ (Pin, Pin)\ n$
$parRes = parallel\ (resistor\ 1000)\ (resistor\ 2200)$

The goal of a type checker is not merely to accept well-typed programs, but also to reject certain ill-formed programs as ill-typed. The definition *broken* given below is such a program. The program is flawed in that there is no relation to which it can safely be applied. The relation *sr* must contribute at least *3* equations for the local variables $\{u, v, w, x\}$ (the forth being accounted for by the local atomic equation), but must not exceed a contribution of 2 equations as dictated by the second application. Consequently, our type system detects this by attempting to generate inconsistent constraints. Note that this program would happily be accepted by the unrefined type system.

$broken : (\ldots, n \leqslant 2, n \geqslant 3, \ldots) \Rightarrow SR\ Pin\ n \rightarrow SR\ Pin\ m$
broken sr = **sigrel** (a, b) **where**
local *u v w x*
$sr \diamond (u + v, w + x)$
$sr \diamond (a, b)$
$v + x = 0$

5.2 An FHM Core Language

One of the primary goals of this article is to formalise the intuition of the type system discussed earlier in this section. A prerequisite to formalising such a system is to make precise the object language of study. Figures 5–8 define such a language, an FHM *core language*, which shall be used as the basis for metatheoretical study of our refined type system throughout the remainder of the article.

However, to allow us to focus on the structural aspects of the type system, the core language has been simplified compared with what would be needed for actual modelling. Thus, the core language deviates from the description of FHM and Hydra given in Sect. 3 in a number of ways. Most notable is the exclusion of signal-level constructs and signal types. Whilst this exclusion might seem like a major departure from Hydra, the refined type system is only concerned with the *kind* of equations and signal relation applications that arise in an equation system, information that can easily be determined prior to type checking. The core language is also based upon the simply-typed λ-calculus rather than Haskell. However, as we shall see, the type system also shares many similarities with Hindley-Milner-style polymorphic calculi [20].

The *types* in the core language are grouped into three categories: monotypes (τ), polytypes (σ), and equation types (μ, ν), see Fig. 5. The monotypes describe the simple function spaces and signal relation types, which are monomorphic in any balance variables. This small set of monotypes would be far richer in a real implementation, but is kept minimal for the sake of formalisation. Dispensing with signal-level values also means that signal relation types need now only be parametrised on a balance (rather than signal-level type and balance), eliminating the need to represent signal types in the core language at all.

Polytypes allow balance variables occurring in monotypes to be *bound*. This is very similar to the notion of binding of type variables found in Hindley-Milner-style type systems. The definition of polytypes is also a convenient place to introduce constraints on monotypes.

Equations in Hydra are essentially untyped, requiring only that their components be well-typed where appropriate. However, in the core language it is necessary to introduce simple equation types (ν) for the sake of constraint tracking. Equation types indicate the number of local, mixed, and interface equations that a compound equation (i.e., system of equations) contains. A numeric literal is not sufficient to describe the contribution, as an equation may contain elements whose contribution cannot be determined statically, for example the application of a signal relation introduced by a λ-abstraction. Thus, it is important to recognise that these expressions can only be approximations of the actual contribution. The category μ associates an equation type with constraints that may be inherited from any signal relations being applied within the compound equation.

$\sigma ::=$	polytype:		
$\forall n \,.\, \sigma$	quantified type	$\mu ::=$	equation type:
$C \Rightarrow \tau$	constrained type	$C \Rightarrow \nu$	constrained equation
$\tau ::=$	monotype:	$\nu ::=$	simple equation:
$\tau_1 \rightarrow \tau_2$	function space	$Eq\; c_1\; c_2\; c_3$	equation
$SR\; n$	signal relation		

Fig. 5: Types.

The syntax of constraints is given in Fig. 6. Constraint expressions (c) are essentially $\langle \mathbb{Z}, + \rangle$: the group of integers closed under addition. This makes it easy to normalise and compare expressions, which is essential for determining type equality. A minimal language of constraints (C) provides inequality and conjunction. Equality constraints $c_1 = c_2$ are translated to $c_1 \leqslant c_2, c_2 \leqslant c_1$, while the shorthand notation $c_1 \leqslant c_2 \leqslant c_3$ is expanded to $c_1 \leqslant c_2, c_2 \leqslant c_3$.

Type equality for monotypes is syntactic. Equality of polytypes is up to α-renaming of bound balance variables and equality of any constraints. Two constrained monotypes are equal when their constraints agree on the intervals of each balance variable. See section 5.6 for a discussion on our present approach to checking constraint equality.

Finally, the syntax of core terms and values are given in Fig. 7 and Fig. 8, respectively. As the simply-typed λ-calculus has been chosen as a template for the core language, the functional-level terms (t) contain the expected λ-terms: variables, function abstraction, and function application. Let bindings are introduced to allow us to *generalise* balance variables, Hindley-Milner style, allowing for polymorphic balance. To keep the system focused on balance aspects, no other polymorphism is supported. It would be straightforward to add support for other forms of polymorphism.

$\Gamma ::=$	context:		
$\bullet$	empty	$c ::=$	constraint expression:
$\Gamma, x : \sigma$	extension	n	balance variable
		zero	zero
$C ::=$	constraint:	**succ** c	successor
ε	empty constraint	$c_1 + c_2$	addition
$c_1 \leqslant c_2$	expression inequality	$-\,c$	negation
c_1, c_2	constraint conjunction		

Fig. 6: Contexts and constraints.

The abstraction over the signal-level is evident in the **sigrel** construct where the pattern introducing the bound variables has been replaced by two natural numbers, i and l, giving the number of interface variables and local variables in scope, respectively. While we are no longer concerned with the binding of signal-level variables, it is still necessary to keep track of the *number* of interface and local variables for the purpose of generating and checking constraints. Local variables are thus accounted for explicitly in the core calculus. Additionally, the number of local variables will increase in a non-trivial way during evaluation. Care is taken in the semantics to correctly account for this non-standard reduction behaviour. Rather than being eliminated due to substitution, local variables are instead propagated up and aggregated in the top level signal relation.

The most important simplification compared with Hydra can be seen in the productions for equations (q). Instead of classifying equations according to what signal variables occur in them as part of the type system, equations are classified directly by labelling them with a *kind*: **local**, **mixed**, or **interface**. This simplification makes our presentation of the type system substantially clearer, without compromising on any of the fundamental concepts of FHM as the labelling can easily be carried out prior to type checking.

Equations may also take the form of a *switch block* (*sw*). The syntax of switch blocks defines a non-empty list of equations, with the initially active branch tagged by the keyword **initially**. In Hydra (Sect. 3.4), each **when**-branch carries a signal expression that provides the condition for activating the branch. As the core language is not concerned with signal-level expressions, this condition is omitted from the syntax. Figure 9 illustrates how Hydra is mapped into the core lanuage.

$t ::=$	functional term:	$q ::=$	equation term:
x	λ-bound variable	**atomic** k	atomic equation
$t_1\ t_2$	application	$t \diamond k$	sig. rel. application
$\lambda\, x\, .\, t_1$	abstraction	$q_1 \wedge q_2$	pairing
let $x = t_1$ **in** t_2	let binding	sw	switch block
sigrel $i\ l$ **where** q	signal relation		
		$k ::=$	equation kind:
$sw ::=$	switch block:	**local**	local equation
initially q	initial branch	**mixed**	mixed equation
sw **when** q	conditional branch	**interface**	interface equation

Fig. 7: Terms.

The next section gives a small-step reduction semantics for the core language. This semantics depends on the definition of values, which describe terms that have been reduced

v ::= functional 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 — conditional branch

Fig. 8: Values.

$parallel\ sr_1\ sr_2\ =$ **sigrel** (p, n) **where**
  local $p_1\ n_1\ p_2\ n_2$
  $sr_1 \diamond (p_1, n_1)$
  $sr_2 \diamond (p_2, n_2)$
  $p.i + p_1.i + p_2.i\ =\ 0$
  $n.i + n_1.i + n_2.i\ =\ 0$
  $p.v\ =\ p_1.v$
  $n.v\ =\ n_1.v$
  $p_1.v\ =\ p_2.v$
  $n_1.v\ =\ n_2.v$

(a) Hydra: parallel composition

$parallel\ =\ \lambda\ sr_1\ .\ \lambda\ sr_2\ .$ **sigrel** $4\ 8$ **where**
  $sr_1 \diamond$ **local** $\wedge$
  $sr_2 \diamond$ **local** $\wedge$
  atomic mixed $\wedge$
  atomic mixed $\wedge$
  atomic mixed $\wedge$
  atomic mixed $\wedge$
  atomic local $\wedge$
  atomic local

(b) Core: parallel composition

$icDiode\ =$ **sigrel** (p, n) **where**
  local u
  $twoPin \diamond (p, n, u)$
  initially; **when** $p.v - n.v > 0 \Rightarrow$
    $u\ =\ 0$
  when $p.i < 0 \Rightarrow$
    $p.i\ =\ 0$

(c) Hydra: ideal diode

$icDiode\ =$ **sigrel** $4\ 1$ **where**
  $twoPin \diamond$ **mixed** $\wedge$
  initially
    atomic local
  when
    atomic local
  when
    atomic interface

(d) Core: ideal diode

Fig. 9: Comparison of Hydra and Core.

as far as is desirable. The body of an abstraction value is a term, meaning that reduction is not performed under binders. This is desirable, as it means open values (values containing free variables) need not be considered. Furthermore, equation values need not contain a production for signal relation applications, as all relation applications will be eliminated from a closed term. Finally, switch values only insist that the **initially** branch be an equation value, as this is the only active branch at the start of simulation. Should a branch condition fire, causing that branch to be switched and the the previously active branch to be switched out, this new branch will be treated as initial, and evaluated before simulation resumes.

5.3 Semantics

Meaning is ascribed to the core language via a small-step reduction semantics [29] given in Fig. 10. The semantics consist of two relations that describe the valid individual reductions

for terms and equations. The reflexive transitive closure of these relations can then be taken as the sequences of valid reductions from terms to values.

The relation $t_1 \longrightarrow t_2$ gives meaning to terms, stating that the term t_1 reduces to the term t_2 in one step. The relation $q_1 \xrightarrow{l} q_2$ relates three objects, stating that the equation q_1 reduces to the equation q_2 in one step, introducing l new local variables in the process. If the semantics were to account for bound signal variables individually, such a reduction would have the form $\Delta \vdash q_1 \longrightarrow \Delta, \Sigma \vdash q_2$, describing the reduction of an equation q_1 in the local variable context Δ, to the equation q_2 in the local variable context Δ extended by a set of new local variables Σ.

$$\frac{t_1 \longrightarrow t_2}{t_1\, t_3 \longrightarrow t_2\, t_3}\ (\textsc{S-App1}) \qquad \frac{t_1 \longrightarrow t_2}{v\, t_1 \longrightarrow v\, t_2}\ (\textsc{S-App2}) \qquad \frac{}{(\lambda\, x\,.\, t)\, v \longrightarrow [x \mapsto v]\, t}\ (\textsc{S-AppAbs})$$

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

$$\frac{t_1 \longrightarrow t_2}{\textbf{let}\ x = t_1\ \textbf{in}\ t_3 \longrightarrow \textbf{let}\ x = t_2\ \textbf{in}\ t_3}\ (\textsc{S-Let}) \qquad \frac{}{\textbf{let}\ x = v\ \textbf{in}\ t \longrightarrow [x \mapsto v]\, t}\ (\textsc{S-LetV})$$

$$\frac{t_1 \longrightarrow t_2}{t_1 \diamond k \xrightarrow{0} t_2 \diamond k}\ (\textsc{S-RApp}) \qquad \frac{}{(\textbf{sigrel}\ i\ l\ \textbf{where}\ qv) \diamond k \xrightarrow{l} qv \downarrow k}\ (\textsc{S-RAppAbs})$$

$$\frac{q_1 \xrightarrow{l} q_3}{q_1 \wedge q_2 \xrightarrow{l} q_3 \wedge q_2}\ (\textsc{S-Pair1}) \qquad \frac{q_1 \xrightarrow{l} q_2}{qv \wedge q_1 \xrightarrow{l} qv \wedge q_2}\ (\textsc{S-Pair2})$$

$$\frac{q_1 \xrightarrow{l} q_2}{\textbf{initially}\ q_1 \xrightarrow{l} \textbf{initially}\ q_2}\ (\textsc{S-Initial}) \qquad \frac{sw_1 \xrightarrow{l} sw_2}{sw_1\ \textbf{when}\ q \xrightarrow{l} sw_2\ \textbf{when}\ q}\ (\textsc{S-When})$$

Fig. 10: Small step semantics.

There are no surprises regarding the reduction rules for λ-terms, with S-APP1, S-APP2, S-APPABS, S-LET, and S-LETV describing the normal rules for a call-by-value λ-calculus with **let**-expressions. A signal relation may undergo a step of reduction by reducing the equation that it contains (S-SIGREL); the current set of local variables (l_1) must also be extended with any new local variables that result from this equation reduction (l_2).

The left branch of an equation pair is reduced first, with the right branch being reduced only after a value is found for the left, as dictated by the rules S-PAIR1 and S-PAIR2. This ordering is arbitrary, but necessary for a deterministic semantics.

A deterministic reduction strategy for $\diamond$ is achieved by first reducing the term being applied. This does not bring any new local variables into scope (S-RAPP). Once the left-hand side of $\diamond$ has been reduced to a signal relation value, the application is reduced to the equation values contained within the relation (S-RAPPABS). If the signal level were not treated in the abstract, the signal expression appearing to the right of $\diamond$ would need to be substituted into the body of these equations. However, the casting of equation kinds ($\downarrow$ below) reflects aspects of this substitution. Finally, the new local variables brought into scope by this reduction step are the local variables declared within the signal relation value.

Note in the rule S-RAPPABS how the equations from within the applied signal relation are transformed by the function $\downarrow$ to bring them into the enclosing context. This transformation is included to keep the notion of equation kinds in line with the kinds expected by the enclosing relation. As discussed previously, the constrained types are necessarily approximations of the actual structure of the flattened system of equations. As reduction proceeds, new information about the flattened structure becomes available, specifically, information regarding the precise kinds of each equation contained with a signal relation abstraction.

$$
\begin{array}{lcl}
\textbf{atomic local} \downarrow k & = & \textbf{atomic local} \\
\textbf{atomic mixed} \downarrow k & = & \textbf{atomic mixed} \\
\textbf{atomic interface} \downarrow k & = & \textbf{atomic}\ k \\
qv_1 \wedge qv_2 \downarrow k & = & (qv_1 \downarrow k) \wedge (qv_2 \downarrow k) \\
\textbf{initially}\ qv \downarrow k & = & \textbf{initially}\ (qv \downarrow k) \\
sv\ \textbf{when}\ q \downarrow k & = & (sv \downarrow k)\ \textbf{when}\ q
\end{array}
$$

The function $\downarrow$ is thus *not* an abstraction of substitution of signal-level variables from the enclosing context into the equations of the applied signal relation, except that local equations must remain local because they are not affected by substitution (as no interface variables occur in them) and because they are not included in the balance of the applied signal relation. Mixed equations also remain **mixed** to retain an optimistic view that they still might be used to solve for any variable, including interface variables in a larger enclosing context, or **local** variables from the initial context in which the equation was defined.

The **initially** branch of a switch is the only branch that will be active at the start of simulation. Thus, the semantics of **initially** and **when** are only concerned with reducing the equations within this initial branch.

5.4 Typing Rules

The type system is defined through three relations (Fig. 11). We have opted for a declarative presentation: in particular, our approach describes polymorphic aspects of the system using non-deterministic *generalisation* and *instantiation* rules in the spirit of the original paper by Milner [20]. The algorithm used to compute types in our system is omitted here as it is essentially the standard algorithm: see Sect. 5.6 for a brief discussion, or the supporting implementation for details [6]. Alternatively, we could have taken the approach found in Pierce [28], making type reconstruction constraints appear explicitly in the rules. However, we felt that this obfuscated the presentation by hiding the important details of the type refinements. The nature of the type system also requires a number of simple type-level computations to be performed for which we define ancillary functions.

The relation for checking λ-terms is denoted by $\Gamma \vdash t : \sigma$, stating that a term t has the type scheme σ in the typing context Γ. Similarly, the relation $\Gamma \vdash_q q : \mu$ states that an equation q has the constrained equation type μ in the context Γ. Equations require the functional typing context Γ as terms may appear in equations (to the left of $\diamond$).

The rules for variables (T-VAR), application (T-APP), abstraction (T-ABS), and let (T-LET) are the normal rules of the simply-typed λ-calculus with additional plumbing to handle the accumulation of constraints. Type checking signal relations (T-SIGREL) requires slightly more attention, using the type of the equations contained within the relation to generate a new set of constraints against the *fresh* balance variable n. The *free* function is responsible for ensuring that the new balance variable n does not already appear free in the context.

The rules for typing equations suggest a simple algorithm for traversing a tree of equations and accumulating the number of local, mixed, and interface equations that a compound equation is capable of *contributing*, whilst simultaneously aggregating constraints from applied signal relations. The strategy can be seen in the rule for relation application (T-RELAPP): given a relation of type $SR\ n$ (i.e. a signal relation contributing n equations), create an equation type contributing n equations of kind k using the *kind* helper function.

Typing switch blocks is a matter of typing the equation fragments at each branch, and then reconciling these types using the approach outlined earlier in Sect. 4.3. Rather than considering all the branches at once, the typing rules provide a syntax-directed approach that reconciles each **when** branch with the remaining branches of the switch.

The instantiation (T-INST) and generalisation rules (T-GEN) allow types to be instantiated or generalised with respect to their balance variables, respectively. Instantiation insists that the new type be more specific according to the ordering relation $\sqsubseteq$. Generalisation allows a free balance variable to be bound, provided that it does not appear free in the environment. Note the strong correspondence between our notion of balance variables and the notion of type variables found in other Hindley-Milner-style systems.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \text{(T-VAR)} \qquad \frac{\Gamma \vdash t_2 : D \Rightarrow \tau_1 \quad \Gamma \vdash t_1 : C \Rightarrow \tau_1 \rightarrow \tau_2}{\Gamma \vdash t_1\, t_2 : C,D \Rightarrow \tau_2}\ \text{(T-APP)}$$

$$\frac{\Gamma, x : \varepsilon \Rightarrow \tau_1 \vdash t : C \Rightarrow \tau_2}{\Gamma \vdash \lambda\, x\, .\, t : C \Rightarrow \tau_1 \rightarrow \tau_2}\ \text{(T-ABS)} \qquad \frac{\Gamma \vdash t_1 : C \Rightarrow \tau_1 \quad \Gamma, x : C \Rightarrow \tau_1 \vdash t_2 : D \Rightarrow \tau_2}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : D \Rightarrow \tau_2}\ \text{(T-LET)}$$

$$\frac{}{\Gamma \vdash_q \mathbf{atomic}\ k : \varepsilon \Rightarrow kind\ (k, 1)}\ \text{(T-ATOMIC)} \qquad \frac{\Gamma \vdash t : C \Rightarrow SR\ n}{\Gamma \vdash_q t \diamond k : C \Rightarrow kind\ (k, n)}\ \text{(T-RELAPP)}$$

$$\frac{\Gamma \vdash_q q_1 : C \Rightarrow \nu_1 \quad \Gamma \vdash_q q_2 : D \Rightarrow \nu_2}{\Gamma \vdash_q q_1 \wedge q_2 : C,D \Rightarrow \nu_1 \oplus \nu_2}\ \text{(T-PAIR)} \qquad \frac{C = cons\ (\nu, n, i, l) \quad \Gamma \vdash_q q : D \Rightarrow \nu \quad fresh\ (n)}{\Gamma \vdash \mathbf{sigrel}\ i\ l\ \mathbf{where}\ q : C,D \Rightarrow SR\ n}\ \text{(T-SIGREL)}$$

$$\frac{D = cons_{sw}\ (Eq\ l\ m\ i, \nu) \quad \Gamma \vdash_q q : C \Rightarrow \nu \quad fresh\ (l, m, i)}{\Gamma \vdash_{sw} \mathbf{initially}\ q : C,D \Rightarrow Eq\ l\ m\ i}\ \text{(T-INITIAL)} \qquad \frac{\Gamma \vdash_q q : D \Rightarrow \nu_2 \quad \Gamma \vdash_{sw} sw : C \Rightarrow \nu_1 \quad E = cons_{sw}\ (\nu_1, \nu_2)}{\Gamma \vdash_{sw} sw\ \mathbf{when}\ q : C,D,E \Rightarrow \nu_1}\ \text{(T-WHEN)}$$

$$\frac{\Gamma \vdash x : \sigma_1 \quad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash x : \sigma_2}\ \text{(T-INST)} \qquad \frac{\Gamma \vdash t : \sigma \quad n \notin free\ (\Gamma)}{\Gamma \vdash t : \forall n\, .\, \sigma}\ \text{(T-GEN)}$$

Fig. 11: Typing rules.

The final matter that requires attention are the various ancillary functions used in the typing rules. The function *free* is left abstract, but it just returns the set of free variables of a context. The (pseudo) predicate *fresh* is also left abstract: it simply enforces that new variables are picked to prevent unintended interference with variables already in use. The operator $\oplus$ is used in T-PAIR to aggregate the contributions of two simple equation types:

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

The *cons* and $cons_{sw}$ functions are responsible for generating constraints, as discussed in Sect 4. The signature of *cons* requires an equation type ν, a fresh balance variable n, and the number of interface variables i and local variables l. The $cons_{sw}$ function requires two equation types ν_1 and ν_2 to be reconciled:

$$
\begin{array}{l}
cons\ (Eq\ i_Q\ m_Q\ l_Q, n, i_Z, l_Z)\ = \\
\quad n\ =\ i_Q + m_Q + l_Q - l_Z, \\
\quad n\ \leqslant\ i_Z, \\
\quad i_Q\ \leqslant\ i_Z, \\
\quad l_Q\ \leqslant\ l_Z\ \leqslant\ l_Q + m_Q, \\
\quad i_Q\ \geqslant\ 0, m_Q\ \geqslant\ 0, l_Q\ \geqslant\ 0
\end{array}
\qquad
\begin{array}{l}
cons_{sw}\ (Eq\ l\ m\ i, Eq\ l_k\ m_k\ i_k)\ = \\
\quad l\ \geqslant\ l_k\ \geqslant\ 0, \\
\quad i\ \geqslant\ i_k\ \geqslant\ 0, \\
\quad m\ \leqslant\ m_k - (l - l_k) - (i - i_k), \\
\quad l + m + i\ =\ l_k + m_k + i_k
\end{array}
$$

The *kind* function provides a convenient method for constructing equation types with a given contribution and of a particular kind:

$$
\begin{array}{l}
kind\ (\mathbf{local}, \quad\ \ c)\ =\ Eq\ 0\ 0\ c \\
kind\ (\mathbf{mixed}, \quad c)\ =\ Eq\ 0\ c\ 0 \\
kind\ (\mathbf{interface}, c)\ =\ Eq\ c\ 0\ 0
\end{array}
$$

Finally, we define the $\sqsubseteq$ 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:

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

5.5 Metatheoretical Properties

The soundness of a type system is often specified via two properties: progress and preservation (also referred to as subject reduction). A type system has progress if, for every closed, well-typed term t, either t is a value or else there is some t' for which $t\ \longrightarrow\ t'$. A type system has subject reduction if evalution of an expression preserves the type of that expression [28].

In our setting, preservation of refined types does not hold. While this might come as a surprise, and may seem indicative of underlying problems, it is actually entirely unsurprising given our objectives. As has been discussed throughout this article, the type refinements we propose are only an approximation for detecting anomalies that would definitely lead to a structurally singular system of equations. Hence, in Hydra, there are modular systems of equations where the fact that they are structural singular only become apparent once they have been completely flattened. In other words, during the process of evaluation, a Hydra program may not necessarily remain structurally well-formed (Sect. 4.1) according to our refinements (but it will remain well-typed). This is expected: the main point is that the refined type system enables many mistakes to be caught early.

The lack of subject reduction is due entirely to the type refinements. Therefore, we shall show that progress and preservation does hold for the unrefined system. Such a proof — which is a new, albeit minor, contribution of this article — gives us some reassurance that the foundations of our system are sound.

Rather than define a new set of typing rules we instead choose to work with *erased* types; the additions made to the type system by our refinements are essentially ignored. Specifically, we erase constraint sets, balance variables on signal relations (*SR*), and constraint expressions on equations (*Eq*).

Progress can be defined formally as follows: given t where $\vdash\ t\ :\ \tau$, then either t is a value, or $\exists\, t'\ .\ t\ \longrightarrow\ t'$. The proof proceeds by induction on the typing derivations (see Fig. 11). We will omit proofs where they do not differ from the standard (and well known) approach to proving soundness of the simply-typed λ-calculus [28].

1. T-ATOMIC: Trivial, atomic equations are values.
2. T-RELAPP: By the induction hypothesis, either there exists a t' such that $t\ \longrightarrow\ t'$, or t is a value. In the first case, the rule S-RAPP simply applies. If t is a value, then by canonicity is must be a **sigrel**, in which case S-RAPPABS applies.
3. T-PAIR: By the induction hypothesis, both q_1 and q_2 may be either values or may take a step of evaluation. If q_1 can take a step, then S-PAIR1 applies. If q_1 is a value, and q_2 may take a step, then S-PAIR2 applies. If both q_1 and q_2 are values, then the entire expression is a value.
4. T-SIGREL: If the hypothesis q may take a step then the rule S-SIGREL applies. If q is a value, then the expression as a whole is a value.
5. T-INITIAL: As above, S-INITIAL applies when q is a value.
6. T-WHEN: As above again, S-WHEN applies when sw is a value. □

Preservation is defined formally as: given t, such that $\Gamma\ \vdash\ t\ :\ \tau$, and $t\ \longrightarrow\ t'$, then $\Gamma\ \vdash\ t'\ :\ \tau$. This proof proceeds by induction on the reduction step $t\ \longrightarrow\ t'$. As before, preservation proofs about the λ-calculus are omitted where they do not differ.

1. S-SIGREL: The induction hypothesis states that $\Gamma\ \vdash_q\ q_1\ :\ \nu$, and $\Gamma\ \vdash_q\ q_2\ :\ \nu$. By inversion of the rule T-SIGREL, we can deduce that ν is an equation type, and by extensions must be *Eq*. Thus, using T-SIGREL again directly on the left- and right-hand side, we can deduce the type *SR* for both.
2. S-RAPP: The induction hypothesis gives us $\Gamma\ \vdash\ t_1\ :\ \tau$, and $\Gamma\ \vdash\ t_2\ :\ \tau$. By inversion of T-RELAPP, $\tau\ =\ SR$, and hence by application again of T-RELAPP, both sides agree on the overall type *Eq*.
3. S-RAPPABS: Once again, we can decompose the hypothesis (**sigrel** *i l* **where** *qv*) $\diamond$ *k* by inversion, resulting in $qv\ :\ \nu$, and thus $qv\ :\ Eq$. The type *Eq* for the left-hand side follows from the rules T-SIGREL and T-RELAPP successively.
4. S-PAIR1: Inversion of T-PAIR along with the induction hypothesis give a straightforward proof of preservation.
5. S-PAIR2: As above, commuted.
6. S-INITIAL: Follows from the induction hypothesis derived from $q_1\ \longrightarrow\ q_2$ and inversion of T-INITIAL.
7. S-WHEN: Follows from the induction hypothesis derived from $sw_1\ \longrightarrow\ sw_2$ and inversion of T-WHEN. □

The above correctness properties are not the only opportunity to validate our type system, not least because they do not consider the type refinements developed in this article. However, the most practically relevant such properties can only be stated in a setting of a language that does not abstract away from which variables occur in which equations; i.e., a language like Hydra, but unlike our core language. As we have not formalised Hydra or its semantics in this article, we do not pursue this further here, but we will sketch the kind of correctness properties we expect to hold, and why, in future work (Sect. 8).

5.6 Implementation

We have implemented a prototype type checker for the type system described in this section. It is implemented in the dependently-typed programming language Agda [26], thus ensuring the totality and termination of the checker. The checker can infer refined types without the need for any type annotations, meaning that the inference algorithm is also total in this sense. The implemented inference algorithm operates in much the same way as *Algorithm* $\mathscr{W}$ with regards to generating type schemes. Specifically, we make no contributions to type inference and instead refer the interested reader to the supporting implementation for details [6].

As discussed in section 5.2, deciding type equality in part requires deciding the equality of constraints. Our present implementation does this by using the *Fourier-Motzkin Quantifier Elimination* algorithm [30], which determines a contiguous interval for each occurring balance variable. This allows the equality of two constrained monotypes to be checked by checking that the constraints agree on the intervals of each balance variable. Fourier-Motzkin's algorithm operates on linear systems of inequalities. Another alternative might have been Collin's Quantifier Elimination [11]. However, Collin's algorithm targets polynomial systems of inequalities which means that it may have higher time complexity that Fourier-Motzkin's algorithm. Thus, Fourier-Motzkin's algorithm is the better choice for us.

Fourier-Motzkin elimination has worst case exponential time complexity in the number of balance variables. However, as shown by Pugh [30], 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.

6 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 most 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 in the introduction, 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 how we can evaluate what we have achieved insofar as we at this point are not in a position to carry out any large usability studies. In this section, we attempt to address that 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 fairly substantial case study that covers all aspects of the language, including structural dynamism.

6.1 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 [21] in 2007. See Sect. 7.1 for a discussion of how Modelica compares to the work described in this paper from the perspective of variable and equation balance. Here we just point out that our system checks more fine-grained structural properties than Modelica as we distinguish between different kinds of equations. This means our system is capable of catching a strictly

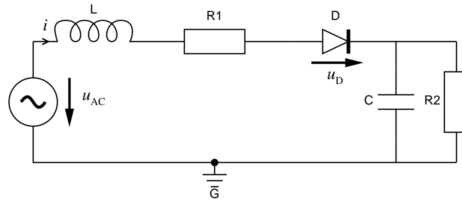

Fig. 12: Half-wave rectifier with in-line inductor.

larger set of errors, and thus is no less useful than the system presently used in Modelica. A concrete example is given towards the end of the case study in the next section. Additionally 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 [32].

The work by Bunus & Fritzson [5], discussed in Sec. 7.3, lies at the other end of the spectrum in terms of precision. Because they work on systems of equations after flattening, Bunus et al. are able to perform a global analysis, which is much more detailed than our type system, or Modelica's balance checking, is capable of. For example, Bunus & Fritzon show how their approach can identify specific equations as likely being the cause of a problem, and even prioritize 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 variable-equation balance.

In summary, in terms of "error finding power", the type system presented in this paper is somewhere between what currently is used in Modelica and the approach investigated by Bunus and Fritzson, both of which empirically are useful for finding problems. Yet, our type-based approach offer distinct advantages over both.

6.2 Case Study: Half-Wave Rectifier

To demonstrate the practical applications of the type system developed in this article, we now present a case study. At this point, the reader may want to first review the examples that were presented in Sect. 5.1. These demonstrated our 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. 12. 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 [24] and originally adapted from Cellier's and Kofman's book *Continuous System Simulation* [10, pp. 439-443], raises particular simulation challenges as the in-line inductor causes the causality to change when the model switches between the two different structural configurations (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:

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

There are manifestly two equations and no local variables to solve for, so the net contribution is two equations.

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

$$
\begin{array}{l}
\mathit{vSourceAC} : (n = 2) \Rightarrow \mathit{Voltage} \to \mathit{Frequency} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\; n \\
\mathit{vSourceAC}\; v\, f = \textbf{sigrel}\;(p,n)\;\textbf{where} \\
\quad \textbf{local}\; u \\
\quad u = v * \mathit{sin}\,(2 * \pi * f * \mathit{time})
\end{array}
$$

Applying the constraint criteria to the voltage source component gives an overall contribution of two equations. This contribution, along with the contributions of several of the components to follow, is easily justified: an application of *twoPin* contributes two equations, while the atomic equation is reserved for solving the local variable u. Applying the typing rules and then simplifying constraints yields the same result.

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

$$
\begin{array}{l}
\mathit{resistor} : (n = 2) \Rightarrow \mathit{Resistance} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\; n \\
\mathit{resistor}\; r = \textbf{sigrel}\;(p,n)\;\textbf{where} \\
\quad \textbf{local}\; u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad r * p.i = u \\
\mathit{inductor} : (n = 2) \Rightarrow \mathit{Inductance} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\; n \\
\mathit{inductor}\; i = \textbf{sigrel}\;(p,n)\;\textbf{where} \\
\quad \textbf{local}\; u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad l * \mathit{der}\; p.i = u \\
\mathit{capacitor} : (n = 2) \Rightarrow \mathit{Capacitance} \to \mathit{SR}\,(\mathit{Pin},\mathit{Pin})\; n \\
\mathit{capacitor}\; c = \textbf{sigrel}\;(p,n)\;\textbf{where} \\
\quad \textbf{local}\; u \\
\quad \mathit{twoPin} \diamond (p,n,u) \\
\quad c * \mathit{der}\; u = p.i
\end{array}
$$

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 the reasons given above. After all, 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 is that the atomic equation in these cases is mixed.

The *ground* component, unlike previous components, is connected via only a single pin:

$$\begin{aligned}
&\mathit{ground} : (n = 1) \Rightarrow \mathit{SR\ Pin}\\
&\mathit{ground} = \textbf{sigrel}\ p\ \textbf{where}\\
&\quad p.v = 0
\end{aligned}$$

Its purpose is to set a reference voltage level. Thus, the 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 initially closed, ideal diode:

$$\begin{aligned}
&\mathit{icDiode} : (n = 2) \Rightarrow \mathit{SR}\ (\mathit{Pin},\mathit{Pin})\ n\\
&\mathit{icDiode} = \textbf{sigrel}\ (p,n)\ \textbf{where}\\
&\qquad \textbf{local}\ u\\
&\quad \mathit{twoPin} \diamond (p,n,u)\\
&\quad \textbf{initially}; \textbf{when}\ p.v - n.v > 0 \Rightarrow\\
&\qquad u = 0\\
&\quad \textbf{when}\ p.i < 0 \Rightarrow\\
&\qquad p.i = 0
\end{aligned}$$

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 conflict is resolved thanks to the *fair* policy (Sect. 4.3) 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. Thus, the net contribution of the diode is two: 2 mixed equations from the application of twoPin and 1 equation from the switch block, 1 of which must be used to solve for the local variable. 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:

$$\begin{aligned}
&\mathit{halfWaveRectifier} : (n = 0) \Rightarrow \mathit{SR}\ ()\ n\\
&\mathit{halfWaveRectifier} = \textbf{sigrel}\ ()\ \textbf{where}\\
&\quad \textbf{local}\ lp\ ln\ rp_1\ rn_1\ rp_2\ rn_2\\
&\quad \textbf{local}\ dp\ dn\ cp\ cn\ acp\ acn\ gp\\
&\quad \mathit{resistor\ 1.0} \qquad\quad\ \diamond (rp_2, rn_2)\\
&\quad \mathit{icDiode} \qquad\qquad\quad \diamond (dp, dn)\\
&\quad \mathit{capacitor\ 0.0} \qquad\ \ \diamond (cp, cn)\\
&\quad \mathit{vSourceAC\ 1.0\ 1.0} \diamond (acp, acn)\\
&\quad \mathit{ground} \qquad\qquad\quad\ \diamond gp\\
&\quad \textbf{connect}\ acp\ lp\\
&\quad \textbf{connect}\ ln\ rp_1\\
&\quad \textbf{connect}\ rn_1\ dp\\
&\quad \textbf{connect}\ dn\ cp\ rp_2\\
&\quad \textbf{connect}\ acn\ cn\ rn_2\ gp
\end{aligned}$$

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 a straightforward matter to construct the appropriate type. There are 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** $p_1 \dots p_x$ desugars to x atomic equations: a sum-to-zero equation and $x - 1$ 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. *twoPin* $\diamond$ (p,n,u)), the application was instead made to a set of interface variables (i.e. *twoPin* $\diamond$ $(p,n,0)$). In a setting with a fair number of both 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 equation kinds (see related work, Sect. 7.1 and Sect, 7.2), the aforementioned error would not be detected. Furthermore, in our system this error would be detected early while type checking *icDiode*, and not only once the full model has been assembled.

7 Related Work

Equation-based modelling is a broad and varied topic. In this section, the most relevant work relating to static checking of structural properties is reviewed and compared with our own.

7.1 Modelica

Modelica is an industrial-strength, equation-based language for acausal modelling of hybrid systems. The language design draws heavily from concepts in object-oriented programming with notions like classes and inheritance used to structure the models. As of version 3.0 of the Modelica specification [21, pp. 43–48, p. 270] models are required to be locally balanced. A model is locally balanced if it locally declares or inherits the same number of variables and equations. No attempt is made to classify equations depending on whether the variables occurring in them are local or not. Moreover, 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. The possibility of 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. Modelica also lacks a notion of true first-class models: there are methods for parametrising models on other models, but these do not approach the generality of FHM. However, this does mean that checking balances late, once parameters are fully known, suffices in the case of Modelica. Furthermore, because Modelica does not classify equations depending on which variables occur in them, the class of structural properties checked by Modelica is smaller than that covered by our type system (see Sect. 4).

7.2 Broman, Nyström & Fritzson

Broman et al. [4] developed a more flexible approach to modular balance checking than the approach described by the current Modelica specification [21] (to which it is a precursor). Most notably, models are not required to be locally balanced provided that the fully assembled system is balanced. The type system, dubbed Structural Constraint Delta (C_Δ), is developed for a subset of Modelica called *Featherweight Modelica*.

The core 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 [21]) and have the same variable-equation balance. This idea is extended to a subtyping relationship where $S <: C$ holds only when S is a Modelica subtype of C, and S and C have the same variable-equation balance. This refinement is motivated by the principle of safe substitution; in this instance, stating that it is only safe to replace one class by another if the replacement preserves the global balance of a 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 Sect. 5 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 parameteric 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 is strictly balance oriented. In contrast, our system captures some structural properties beyond simple balance. For example, signal relations are valid only when they do not over- or under-constrain their local variables.

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. [25,22] and Broman et al., with the latter giving the first detailed account of such an approach.

7.3 Bunus & Fritzson

Bunus & Fritzson [5] 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 structural properties, as we are, but, allowing systems to be flattened before analysis grants them the capacity 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 will partition 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 & 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.

Because the methods outlined are intended to be used after a modularly constructed system has been flattened, the methods are in many ways complimentary to the type system presented in this article. The methods could even be performed during simulation, making them potentially very useful for analysis of iteratively-staged, structurally-dynamic systems [15]. In any case, the work by Bunus & Fritzson illustrates the benefits from going beyond basic balance checking for finding problems with systems of equations. Some of those benefits are also realised by our system thanks to the classification of equations into different kinds depending on the variables that occur in them; i.e., an approximation of individual variable occurrences.

7.4 Furic

Furic [13] proposes a novel approach for model composition for Modelica with improved guarantees of compositionality. A notion of variable and equation balance is central to this. Like in the approach adopted by Modelica, no classification of equations is made depending on whether occurring variables are local or not. Furic's balance checking algorithm works on a *physical connection graph* describing the structure of an assembled system. Its present formulation is thus 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. However, 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 variable and equation balance for modularly constructed systems of equations. Moreover, his approach to composition offers a number of advantages over Modelica's, and it would be interesting to see if it can be recast into a type-based approach, and maybe even adapted to the FHM setting.

7.5 Nilsson

The work by Nilsson [22], which is a precursor to this work, outlines an approach to static checking that makes stronger guarantees about the structure of equations and variables beyond that of simple balance. In many cases, Nilsson's *structural types* are able to rule out systems with structural singularities that would otherwise be accepted under a simple balance checking approach. As with the system developed in this paper (Sect. 5), Nilsson develops his approach for the FHM framework.

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 Bunus and Fritzson's technique [5], while retaining the capability of checking fragments in isolation. Partitioning equations into classes

depending on whether the occurring variables are local or interface or both is central to Nilsson's approach and led to the notion of equation kinds in this paper.

Nilsson's work is a preliminary investigation into structural types. It does not consider first-class models, and 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. Structurally dynamic systems are considered, but only briefly. The time complexity of the given algorithm to compute structural types is also a concern as it relies on partitioning the set of mixed equations in all possible ways. Moreover, the the flip-side of the precision of the types is that they may be hard to understand and cumbersome to use in practice. Suitable methods by which 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. Additionally, this paper also considers structural dynamic systems in depth.

7.6 Capper & Nilsson

The key ideas in this article were first presented by Capper et al. [7], which contained a preliminary investigation into capturing structural properties of equation systems using constrained types. Since the initial investigation, the type system has been improved and extended in a number of ways, as described in this paper.

The core language has seen major improvements: eliminating unnecessary noise from the language has lead to improvements in the presentation of the semantics and type system. Moreover, the semantics now give an accurate account of variables in a modular systems of equations, which has already shown to be useful for work in progress by Capper et al. based upon early work for a denotational model of FHM [8].

An important extension featured in this article is the handling of structurally dynamic systems. In particular, we outline constraints that define a *fair* policy (Sect. 4.3) for reconciling the branches of a switch, allowing structural properties of the branches to be verified. Furthermore, the existing constraint criteria for signal relations have been refined.

8 Future Work

There are a number of avenues of potential future work stemming from the system developed in this article. In this section, these avenues are briefly explored, including discussion about the utility, complexity, and importance of each extension.

The early discussion of structural dynamism (Sect. 2.4) raised the issue of (re)-initialisation. Specifically, automatic initialisation of equation systems is in general a hard problem, and thus, we chose not to consider these aspects when designing the constraints for the refined type system. Currently, Hydra allows the modeller to express initialisation logic explicitly using (re)-initialisation equations. Whilst this approach remains the most practical solution, it would be desirable to capture structural properties of initialisation equations in the refined type system. For example, such equations might be considered as a new *kind* of equation, for which new structural invariants could be enforced.

A related problem is that of redundant equations, the utility of which underpins work by Nilsson et al. [24] on simulating ideal diodes by exploiting structural dynamism. The crux of the paper relies on introducing redundant equations — equations that do not specify

new constraints when added to a system — intentionally creating an *unbalanced* system of equations. Such a system would be rejected by our refined type system (indeed, this is the point of the refined types), where instead it would be preferable to allow the modeller to express such intentions (i.e., that an equation is redundant and is only present to ensure that a particular technique for solving the equations will succeed). One solution might be to allow the modeller to mark equations as *weak* or *dependent*, leaving it up to the type system to decided whether said *weak* equations should be considered when generating constraints.

Another important consideration is the usability of the type system. 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 equations system to become unrecognisable to the modeller. In such instances, the work by Bunus et al. [5] may prove useful in tracking the surface-level meaning of programs through syntactic transformations, allowing errors to be communicated in a more meaningful way.

In Sect. 4 we regard the constraint criteria as domain agnostic: the criteria are applicable regardless of the chosen domain. It would be useful to consider generating constraints for specific domains, where more information about the structure of equations means that stronger structural invariants can be expressed.

Finally, as mentioned in Sect. 5.5, an important undertaking would be to prove metatheoretical properties that related directly to the refinements of the type system. However, as stated in the aforementioned section, to state interesting properties beyond simple preservation of unrefined types, one would likely need to consider a system with *concrete* signal relation reduction (i.e. reduction that does not abstract away from variable occurances).

We would be particularly interested to show that if $t : C \Rightarrow SR\ ()\ n$ and $\neg\ \mathit{satisfiable}\ (C)$, then there exists a structural configuration (i.e., a particular choice of switch branches), such that t elaborates to a structurally singular system of equations. In other words, if a complete modular system of equations is well-typed but ill-formed, then, at least for one possible configuration, it really is a bad system, thus justly ruling it out. Intuitively, this follows directly from the criteria of Sect. 4. Any one of these criteria is unsatisfiable only when it is clear that there isn't going to be a way to pair each equation with a variable even when grossly overapproximating which variables occur in which equation. During elaboration, the exact set of variables occurring in each equation is gradually going to become manifest. But this set is necessarily a subset of the overapproximation of occurrences on which the criteria is defined. Thus, if pairing was not possible *before* elaboration, it is certainly not going to be possible *after* elaboration, where the number of choices of which equation to pair with which variable is not going to be greater than before (but likely much smaller).

9 Summary and Conclusions

This article presents a novel and powerful approach to detecting structural problems in modular systems of equations. Components can be analysed in isolation, rather than requiring assembly into a complete system of equations, thus allowing over- and underconstrained systems to be detected early, aiding in error localisation. In particular, we advance the current state-of-the-art by presenting a type system that is capable of handling first-class, structurally dynamic models. Furthermore, the type system is able to detect more structural properties than existing type systems by considering the *kinds* of equations that occurr in

a modular system of equations. We also put forth a concise small-step semantics for FHM that considers the non-trivial impact of evaluation on local variables.

Finally, it is worth remarking that the principles of the developed type system are in no way specific to FHM and should be applicable to modular equation systems in general.

Acknowledgements We would like to thank the anonymous reviewers of this paper and the preceding TFP paper [7] for many insightful comments and valuable suggestions.

References

1. Accellera Organization: Verilog-AMS language reference manual — analog & mixed-signal extensions to Verilog HDL version 2.3.1 (2009)
2. Aho, A.V.: The C Programming Language (1988)
3. Barbeau, E.J.: Pell's Equation, Problem Books in Mathematics. Springer-Verlag (2003)
4. Broman, D., Nyström, K., Fritzson, P.: Determining over- and under-constrained systems of equations using structural constraint delta. In: GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, pp. 151–160. ACM, Portland, Oregon, USA (2006)
5. Bunus, P., Fritzson, P.: A debugging scheme for declarative equation based modeling languages. In: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages (PADL 2002), *Lecture Notes in Computer Science*, vol. 2257, pp. 280–298. Springer-Verlag, OR, USA (2002)
6. Capper, J.J.: Source code repository. `www.cs.nott.ac.uk/~jjc`
7. Capper, J.J., Nilsson, H.: Static balance checking for first-class modular systems of equations. In: Proceedings of the 11th Symposium on Trends in Functional Programming. Oklahoma, USA (2010)
8. Capper, J.J., Nilsson, H.: Towards a formal semantics for structurally dynamic non-causal modelling languages. In: Types in Language Design and Implementation. Philadelphia, Pennsylvania, USA (2012)
9. Cellier, F.E.: Object-oriented modelling: Means for dealing with system complexity. In: Proceedings of the 15th Benelux Meeting on Systems and Control, Mierlo, The Netherlands, pp. 53–64 (1996)
10. Cellier, F.E., Kofman, E.: Continuous System Simulation. Springer-Verlag (2006)
11. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Proceedings Second GI Conference on Automata Theory and Formal Languages, *Lecture Notes in Computer Science*, vol. 33, pp. 134–183. Springer-verlag (1975)
12. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of ICFP'97: International Conference on Functional Programming, pp. 163–173 (1997)
13. Furic, S.: Enforcing model composability in Modelica. In: F. Casella (ed.) Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009, *Linköping Electronic Conference Proceedings*, vol. 43, pp. 868–879. Linköping University Electronic Press (2009)
14. Giorgidze, G., Nilsson, H.: Higher-order non-causal modelling and simulation of structurally dynamic systems. In: F. Casella (ed.) Proceedings of the 7th International Modelica Conference, Como, Italy, 20–22 September 2009, *Linköping Electronic Conference Proceedings*, vol. 43, pp. 208–218. Linköping University Electronic Press (2009)
15. Giorgidze, G., Nilsson, H.: Mixed-level embedding and JIT compilation for an iteratively staged DSL. In: J. Mariño (ed.) Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP 2010), *Lecture Notes in Computer Science*, vol. 6559, pp. 48–65. Springer-Verlag (2011)
16. IEEE Std 1076.1-2007: IEEE Standard VHDL Analog and Mixed-Signal Extensions. IEEE Press (2007)
17. Jones, S.P., et al.: Haskell 98 – A non-strict, purely functional language. `http://www.haskell.org/onlinereport` (1999)
18. Kirchhoff's circuit laws. Wikipedia. `http://en.wikipedia.org/wiki/Kirchhoff's_circuit_laws`, visited Oct. 2012
19. McKinna, J., Altenkirch, T., McBride, C.: Why Dependent Types Matter. ACM SIGPLAN Notices **41**(1)
20. Milner, R.: A theory of type polymorphism in programming. JCSS: Journal of Computer and System Sciences **17** (1978)
21. Modelica Association: Modelica — A Unified Object-Oriented Language for Systems Modelling; Language Specification Version 3.3 (2012). URL `http://www.modelica.org`
22. Nilsson, H.: Type-based structural analysis for modular systems of equations. In: P. Fritzson, F. Cellier, D. Broman (eds.) Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, no. 29 in Linköping Electronic Conference Proceedings, pp. 71–81. Linköping University Electronic Press, Paphos, Cyprus (2008)

23. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02), pp. 51–64. ACM Press, Pittsburgh, Pennsylvania, USA (2002)
24. Nilsson, H., Giorgidze, G.: Exploiting structural dynamism in Functional Hybrid Modelling for simulation of ideal diodes. In: Proceedings of the 7th EUROSIM Congress on Modelling and Simulation. Czech Technical University Publishing House, Prague, Czech Republic (2010)
25. Nilsson, H., Peterson, J., Hudak, P.: Functional hybrid modeling. In: Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages, *Lecture Notes in Computer Science*, vol. 2562, pp. 376–390. Springer-Verlag, New Orleans, Lousiana, USA (2003)
26. Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Tech. rep., Chalmers University of Technology (2007)
27. Nytsch-Geusen, C., Ernst, T., Nordwig, A., Schwarz, P., Schneider, P., Vetter, M., Wittwer, C., Nouidui, T., Holm, A., Leopold, J., Schmidt, G., Mattes, A., Doll, U.: MOSILAB: Development of a Modelica-based generic simulation tool supporting model structural dynamics. In: Proceedings of the 4th International Modelica Conference, pp. 527–535. Hamburg, Germany (2005)
28. Pierce, B.: Types and Programming Languages. The MIT Press (2002)
29. Plotkin, G.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark (1981)
30. Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing 91 (1991)
31. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation, pp. 242–252 (2000)
32. Zimmer, D.: Equation-based modeling of variable-structure systems. Ph.D. thesis, Swiss Federal Institute of Technology, Zürich (2010)