Static Balance Checking for First-Class Modular Systems of Equations

John Capper and Henrik Nilsson

Functional Programming Laboratory,
School of Computer Science,
University of Nottingham,
United Kingdom
{jjc,nhn}@cs.nott.ac.uk

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 in particular makes it possible to treat equation system fragments as true first-class entities. 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.

Key words: Systems of equations; equation-based, non-causal modelling; first-class components; equation-variable balance; structural analysis; linear constraints; refinement types.

1 Introduction

Systems of equations [3], also known as simultaneous equations, are abundant in science and engineering. Applications include modelling, simulation, optimisation, and more. Such systems of equations are often parametrised, describing not just a specific problem instance, but a set of problems. The size and nature of the systems frequently necessitates numerical methods and computers for solving them. The equations thus need to be turned into programs that can be used to solve for various problem instances. Such programs can be written manually, but a more expedient route is to transcribe the equations into a high-level language, e.g. a modelling language, thus making it possible to automatically translate the equations into a program that attempts to compute a solution given specific values for any parameters. 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, *components*, is often a practical necessity.

Of course, as with any large and complex task, there is always a risk of mistakes being made. In this case, mistakes may render the system of equations unsolvable. In a modular development, an error in a component might not manifest itself until an attempt is made to use that component. In the worst case, problems might not become apparent until much later when the final program is run. In some applications, the system of equations may even evolve dynamically, say *during* a simulation run, meaning that it may take a long time indeed to discover certain errors. Static checks that catch mistakes early, preferably applicable to individual components in isolation, can thus be very helpful.

One might hope to statically impose sufficient constraints to guarantee that a system of equations has a solution. Unfortunately, the question of whether such a system has a solution or not can in general only be answered by studying complete systems with full knowledge of all coefficients, ruling out checking of components in isolation as well as parametrisation. Moreover, without actually attempting solving, the question can only be answered for relatively simple systems (e.g. linear systems of equations). In other words, if the setting is reasonably general, we *cannot* hope to develop e.g. a type system that guarantees that a well-typed equation system or fragments thereof are solvable.

However, there are simple criteria that if violated are *indicative* of problems, or that may even imply that an attempt to solve a system by a specific method (e.g. as embodied by a tool that translates equations to a program for solving them) will necessarily fail. One such criterion is that the number of variables, or *unknowns*, must equal the number of equations. A more refined criterion is that there should exist a bijective mapping between variables and equations. Some of these kinds of criteria *can* be enforced statically, e.g. through a type system.

Enforcing the balance of systems of equations is considered very useful in practise. For example, the state-of-the-art, equation-based modelling and simulation language Modelica insists that complete models are balanced [9, pp.40–46]. Indeed, translation to simulation code will fail if systems are unbalanced. Broman et al. propose a similar but more refined approach [1].

These criteria stem from the fact that a *linear* system of equations has a unique solution if and only if the equations are linearly independent and the number of equations and unknowns agree. However, they are useful heuristic criteria more generally, intuitively because each equation commonly can be used to solve for one variable occurring in it. For a (very) simple example, consider:

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

Here (2) can be used to solve for x, and the value of x can then be substituted into (1), enabling it to be used to solve for y. Note that both the variable-equation balance criterion and the pairing criterion are satisfied.

On the other hand, it is easy to see that neither criterion is sufficient to guarantee solvability. Consider:

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

Note that the system now is parametrised on a coefficient c. The two criteria are still satisfied, but whether the system has a unique solution or not depends on the value of c: for $c = 0$ there is no unique solution. Conversely, violation of the criteria does not necessarily mean a system is unsolvable; for example, consider adding an extra copy of (2) to the first system. The resulting system can of course still be solved, despite both criteria now being formally violated.

The existing approaches to balance checking have weaknesses. For example, in Modelica, a component either has to be balanced, or it is explicitly declared to be possibly unbalanced, in which case no balance checking is performed for that component. See Sect. 4 for a more in-depth discussion. In this paper we develop an approach that is both more flexible and capable of catching more problems:

– The type of a component is refined by adding a balance variable to it, reflecting the number of equations the component contributes to the overall system. This is a refinement type system [4] in that erasure of the extra type information recovers a term that is well-typed in the original system.
– Parameterised components may also have a parameterised balance.
– Balance information can be inferred for components in isolation, even when parametrised on other components and without any explicit declaration of balance information for such parameters.
– Additional structural constraints beside the balance are exploited for a more refined analysis. For example, in certain cases, it can be established that a component necessarily would render a system imbalanced whenever it is used, which thus can be reported as an error.

The upshot of this is that if a complete system is assembled modularly from components that are well-typed in the refined sense, and if the assembled system is balanced overall, then the "flat" system that results by unfolding all definitions will also be balanced.

Our immediate motivation comes from Functional Hybrid Modelling (FHM) [11, 12, 5] where it is desired to treat components as true first-class entities, including the possibility to modify the overall system of equations *during* simulation, at "run-time", as alluded to earlier. Static checks that help prevent accidentally changing a system from one that can be simulated (solved) to one that cannot are thus of particular interest. We do not explicitly consider structurally dynamic systems of equations here, but our type system can be easily extended to that setting thanks to the first-class notion of components.

However, it should be noted that the essence of the ideas presented in this paper are not at all specific to FHM: in principle, it should be relatively straightforward to adapt them to other equation-based modelling languages, like Modelica, or to any language featuring a notion of modular system of equations.

The structure of the remainder of this paper is as follows. Sect. 2 explains the idea of modular systems of equations in more depth. Sect. 3 describes the type system developed. Sect. 4 gives a comparative review of the related work. Sect. 5 looks at possible avenues for expansion of the type system. Finally, Sect. 6 provides some concluding remarks.

2 Modular Systems of Equations

This section introduces the idea of modular systems of equations in more detail. As FHM provided the immediate motivation for this work, we will draw on FHM for examples and we will adopt a concrete syntax derived from Hydra, an FHM language currently being developed. We will only explain FHM and Hydra to the extent needed for this paper; for further details, please consult Nilsson et al. [11, 12] or Giorgidze & Nilsson [5].

Hydra, like Modelica, is concerned with modelling of dynamic, physical systems using Differential Algebraic Equations (DAE). The solution to such a system of equations is a set of time-varying reals, i.e. real-valued functions of time. In practise, it is usually the case that only approximate solution through numerical simulation is feasible. However, for our formal type system development, the domain of the variables and the exact form of the equations is of no consequence: all that matters is which variables occur in each equation. This is reflected in the precise syntax of terms for which our type system is defined (see Fig. 3.2 in Sect. 3.2), where equations are only considered in the abstract as a set of occurring variables.

2.1 Equation System Basics

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. Again, for the type system developed in this paper, the domain of the variables is not important. However, in our examples, the domain is either the reals, $\mathbb{R}$, or time-varying reals.

A *linear* system of equations has a unique solution if all equations are independent and there are equally many equations and variables. If there are more independent equations than variables, the system is *over-constrained*. Such a system has no solution as there are too many constraints, some of which will be in conflict. If there are fewer independent equations than variables, the system is *under-constrained*. Such a system has infinitely many solutions.

The *equation-variable balance* of a system of equations is the difference between the number of equations and variables. Note that this is strictly a structural property: the details of exactly what the equations look like is of no consequence. This is true in general in our development: we only consider structural properties, i.e. equations in the abstract, as we cannot assume that all details are known. By analogy, we refer to a system with positive equation-variable balance as over-constrained, and one with negative balance as under-constrained, regardless of whether the equations actually are independent or even linear.

2.2 Abstraction over Systems of Equations

The equation systems needed to describe real-world problems are usually large and complicated. On the other hand, there tends to be a lot of repetitive structure 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 the components.

While the exact syntactic details vary between languages, the idea, in essence, 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:

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

This makes r a *relation* that constrain the possible values of the two *interface variables* x and y according to the encapsulated equations. The variable z is *local*, not visible from the outside.

The relation r 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 application, denoted by $\diamond$:

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

After unfolding and renaming, a process we refer to as *flattening*, we get:

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

Note that each application of r effectively *contributes* one equation to the overall system as one of the instances of the encapsulated equations in each case must be used to solve for the corresponding instance of the local variable, z_1 and z_2.

2.3 FHM and Hydra

In this section, we introduce the FHM framework as embodied by the language Hydra [11, 12, 5]. We use this as the setting for the rest of the paper. The central idea of FHM is to embed an abstraction mechanism over equations as described in Sect. 2.2 into a pure functional language, allowing equation system abstractions to be *first-class entities* at the functional level. The equations are Differential Algebraic Equations (DAE), meaning that the domain of the variables is time-varying reals, or *signals*. An abstraction over an equation is therefore referred to as a *signal relation*. In the case of Hydra, the host language is Haskell [6].

In Hydra, the type of a signal relation is written $SR\ \alpha$. A signal relation can be thought of as a predicate on a signal:

$$SR\ \alpha \approx Signal\ \alpha \rightarrow Bool$$

where $Signal\ \alpha$ is a time-varying value of type α. As a product of signals is isomorphic to a signal of products, unary signal relations suffice to represent n-ary relations. For example, given a binary predicate $\equiv$ on $\mathbb{R}$:

$(\equiv_{sr}) :: SR\ (\mathbb{R}, \mathbb{R})$
$(\equiv_{sr})\ s = \forall\ (t :: Time).\quad fst\ (s\ t) \equiv snd\ (s\ t)$

First-class signal relations are constructed as follows:

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

The pattern introduces *interface variables* that scope over the equations. The latter may refer to additional, implicitly declared, *local variables*. Together, these two kinds of variables are referred to as *signal variables* as they stand for time-varying quantities. There are two forms of equations:

$$e_1 = e_2 \tag{5}$$

$$sr \diamond e_3 \tag{6}$$

where sr is a *time-invariant* expression (*free* signal variables must not occur in it[1]) denoting a signal relation, and $\diamond$ denotes signal relation application, similarly to Sect. 2.2. Functional level objects can be used as time-invariant entities inside signal relations. In particular, functional-level variables can be used as coefficients in equations, thus allowing the equations to be parametrised: see the *resistor* example below for an example. On the other hand, time-varying signal-level entities are not permitted to escape to the functional level.

Signal variables scope over the time-varying, top-level equations of a signal relation. Since only time-invariant expressions may appear to the left of an application, nested signal relations are not permitted.

To illustrate, consider a component *twoPin*, encapsulating equations common to all electrical components with two pins, and a component *resistor*, defined as an extension of *twoPin* by adding an equation that describes the behaviour of a resistor:

[1] However, a manifest signal relation expression is fine as it binds *all* signal variables occurring in it. That is, signal relations can be “nested”, but the signal variable scope is flat.

$$\begin{array}{l}
\textbf{type } \mathit{Pin} = (\mathbb{R}, \mathbb{R}) \\
\\
\mathit{twoPin} :: \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}, \mathit{Voltage}) \\
\mathit{twoPin} = \textbf{sigrel}\ (p, n, u)\ \textbf{where} \\
\quad \mathit{fst}\ p - \mathit{fst}\ n \quad = u \\
\quad \mathit{snd}\ p + \mathit{snd}\ n = 0 \\
\\
\mathit{resistor} :: \mathit{Resistance} \rightarrow \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \\
\mathit{resistor}\ r = \textbf{sigrel}\ (p, n)\ \textbf{where} \\
\quad \mathit{twoPin} \diamond (p, n, u) \\
\quad r * \mathit{snd}\ p = u
\end{array}$$

Note that the resistor is modelled by a function that maps a resistance to a signal relation. In the definition of *resistor*, r is thus a time-invariant value, not an unknown. Note also that u is local. Flattening the signal relation that results from the *function* application *resistor* 220 yields the flat equation system:

$$\begin{array}{ll}
\mathit{fst}\ p - \mathit{fst}\ n & = u \\
\mathit{snd}\ p + \mathit{snd}\ n & = 0 \\
220 * \mathit{snd}\ p & = u
\end{array}$$

3 The Type System

The type system is presented as an embedding of an equation-based language into the simply-typed λ-calculus. An embedding into the λ-calculus reflects the two-level approach taken by FHM, from which much of the expressivity of the language is gained. The type system has been implemented in the dependently typed programming language Agda [13], giving us assurances that the algorithm is both total and terminating.

Description	Symbol
λ-bound variables	x, y
Expressions (λ-terms)	$e \in \Lambda$
Signal-variables	z
Balance type-variables	n,m,$o \in \mathbb{Z}$
Signal level expressions	s

Description	Symbol
Equations	q
Simple types	τ
Type schemes	σ
Typing environments	Γ
Constraint sets	C

Fig. 1. Notational Conventions

The notation $\overline{\chi}$ is used to denote a sequence $\chi_1, \ldots, \chi_n$ without repetition of elements. We will also allow ourselves to treat $\overline{\chi}$ as sets equipped with the usual set-theoretic operations. One should also note that x (and y) and z are *meta-variables*, ranging over the names of concrete function-level and signal-level variables, respectively.

3.1 Overview

As signal relations are first-class entities, it cannot be assumed that components can be flattened in order to determine the equation-variable balance. The only reasonable assumption is that all that is known statically is the *type* of a relation.

To track the equation-variable balance, the type of a signal relation is refined by annotating it with the number of equations it is able to contribute to a system. The contribution of a signal relation may also depend on the contribution of the parameters to the signal relation. Hence, signal relations can behave in a polymorphic fashion, contributing varying numbers of equations depending on the context in which the relation is used. See Sect. 4 for a comparative review of alternative type system designs.

Since the structural information required to determine a precise contribution may not always be available, the context in which a signal relation is applied is used to generate *balance constraints* (from now on, simply constraints). These constraints restrict the balance of a component to an interval.

Note that a representation of integers and linear inequalities has been introduced at the type level. This extension may appear to be a restricted form of dependent types [8]. However, these type level representations, whilst determined by the structure of terms, are not value level terms themselves. As such, we do not consider our system to be dependently typed.

Constraints may mention the contributions of several components, and hence are not directly associated with a single signal relation. As a result, the type of a signal relation is restricted to being annotated by a *balance variable* which is then further refined using constraints. The type checking algorithm generates a fresh balance variable for each signal relation, with type equality defined up to alpha equivalence of balance variables. As an example, the refined type for *resistor* from Sect. 2.3 is:

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

Haskell's type class constraint syntax has been adopted to express that the balance type 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.

3.2 Generating Constraints

In this section we address the issue of what constraints should be generated. It is conceivable that different application domains could generate constraints specific to that domain. This is not a problem, as the system developed is independent of the constraints generated. For the purposes of this paper, 4 criteria for generating constraints have been chosen. Before introducing the criteria, a number of definitions are required.

Fig. 2 and 3 give the syntax of terms and types from which the type checking algorithm will be developed. A number of simplifications have been made to the FHM framework in order to keep the presentation of the type system concise. Note that all simplifications are superficial and do not fundamentally change the nature of the problem.

$$
\begin{array}{lll}
e ::= x & \sigma ::= \overline{C} \Rightarrow \tau & C ::= ce_1 = ce_2 \\
\quad \mid\ e_1\ e_2 & & \quad \mid\ ce_1 \geqslant ce_2 \\
\quad \mid\ \lambda x.e & \tau ::= \tau_1 \rightarrow \tau_2 & \\
\quad \mid\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \quad \mid\ \mathit{SR}\ \mathbb{R}^m\ n & ce ::= n \\
\quad \mid\ \textbf{sigrel}\ \overline{z}\ \textbf{where}\ \overline{q} & \quad \mid\ \mathit{LEqn}\ n & \quad \mid\ \underline{\mathit{IntLit}} \\
 & \quad \mid\ \mathit{IEqn}\ n & \quad \mid\ ce + ce \\
q ::= \mathit{Atomic}\ \overline{z} & \quad \mid\ \mathit{MEqn}\ n & \quad \mid\ -\ ce \\
\quad \mid\ e \diamond \overline{z} & &
\end{array}
$$

Fig. 2. Syntax of terms, types, and constraints.

We consider the simply-typed λ-calculus, given by e, augmented with first-class signal relation constructs. Signal relations abstract over sets of signal variables, denoted $\overline{z}$, and embed a new syntactic category of *equations* into the calculus, given by q.

Signal relations range over sets of equations, which may take one of two forms. An atomic equation of the form $s_1 = s_2$ is abstracted to just the set of distinct signal variables occurring in the signal expressions s_1 and s_2. Similarly, an equation of the form $e \diamond \overline{s}$ is abstracted to the expression denoting the applied signal relation and the set of signal variables that occur on the right-hand-side of the application. More detailed comments on theses syntactic categories are given in Sect. 3.3.

An equation q is said to *mention* a signal variable z if and only if $z \in \mathit{vars}\ (q)$. The function *total* returns the raw number of atomic equations contributed by an equation. Whereas $|\overline{q}|$ denotes the cardinality of the set of *modular* equations. Both *vars* and *total* are also overloaded for sets of equations.

$$
\begin{array}{llll}
\mathit{vars}\ (\mathit{Atomic}\ \overline{z}) & = \overline{z} & \mathit{total}\ (\mathit{Atomic}\ \_) & = 1 \\
\mathit{vars}\ (\_ \diamond \overline{z}) & = \overline{z} & \mathit{total}\ (e : \mathit{SR}\ \_\ n \diamond \_) & = n \\
\mathit{vars}\ (\overline{q}) & = & \mathit{total}\ (\overline{q}) & = \\
\quad \bigcup \{\, \mathit{vars}\ (q) \mid q \in \overline{q} \} & & \quad \sum \{\, \mathit{total}\ (q) \mid q \in \overline{q} \} &
\end{array}
$$

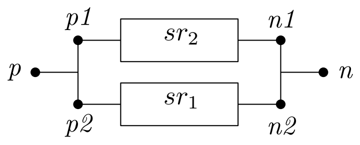
Given a signal relation **sigrel** $\overline{z}$ **where** $\overline{q}$, the set of interface variables is defined $I_Z = \overline{z}$, and the set of local variables $L_Z = \mathit{vars}\ (\overline{q}) \backslash \overline{z}$. The set of equations $\overline{q}$ can be partitioned into the disjoint subsets of interface equations I_Q, local equations L_Q, and mixed equations M_Q, where I_Q is the set of equations mentioning only interface variables, L_Q is the set of equations mentioning only local variables, and $M_Q = (\overline{q} \backslash I_Q) \backslash L_Q$. Finally, the balance of a signal relation, written *bal* (sr), is given as *bal* $(\textbf{sigrel}\ \overline{z}\ \textbf{where}\ \overline{q}) = \mathit{total}\ (\overline{q}) - |L_Q|$. Intuitively, balance is an aggregate of the equations in the body of a signal relation, excluding sufficiently many equations to solve for the local variables.

1. $|L_Q| + |M_Q| \geqslant |L_Z|$. The local variables are not under-constrained.
2. $|L_Q| \leqslant |L_Z|$. The local variables are not over-constrained.
3. $|I_Q| \leqslant |I_Z|$. The interface variables are not over-constrained.
4. $0 \leqslant \mathit{bal}\ (sr) \leqslant |I_Z|$. A signal relation must contribute equations only for its interface variables. It should not be capable of removing equations from other components (negative balance), or adding equations for variables not present in its interface.

The above criteria produce constraints that give adequate assurances for detecting structural anomalies. There is potential to further refine these criteria. However, for the purposes of this paper, these criteria are sufficient to demonstrate the value of the type system.

To illustrate the application of the above five criteria, consider the Hydra example *par* that connects two circuit components in parallel. The operational details of this example are not important; the only important aspect is that of equations *mentioning* variables. The type signature gives the type of *par* under the simply typed approach. The reader may wish to refer back to Sect. 2.3 at this point for clarification on **sigrel** terms.

```
$par :: SR\ (Pin, Pin) \rightarrow SR\ (Pin, Pin) \rightarrow SR\ (Pin, Pin)$
$par\ sr_1\ sr_2 =$
  $\textbf{sigrel}\ ((pi, pv), (ni, nv))\ \textbf{where}$
    $sr_1 \diamond ((p1i, p1v), (n1i, n1v))$
    $sr_2 \diamond ((p2i, p2v), (n2i, n2v))$
    $pi + p1i + p2i = 0$
    $ni + n1i + n2i = 0$
    $pv = p1v = p2v$
    $nv = n1v = n2v$
```



Under the new type system, the signal relations in *par* are annotated by balance variables, which are then constrained by the criteria producing the following refined type:

$$par :: \{m = n + o - 2, 6 \geqslant n + o \geqslant 2, 0 \leqslant m \leqslant 4, 0 \leqslant n \leqslant 4, 0 \leqslant o \leqslant 4\} \Rightarrow$$
$$SR\ (Pin, Pin)\ n \rightarrow SR\ (Pin, Pin)\ o \rightarrow SR\ (Pin, Pin)\ m$$

While this type may appear daunting at first, all balance variables and constraints can be inferred without requiring the programmer to annotate the terms explicitly. It is also useful to see an example of a program that fails to type check under the new type system – a program that previously would have been accepted, despite being faulty.

```
$broken\ sr = \textbf{sigrel}\ (a, b)\ \textbf{where}$
  $sr \diamond (w + x, y + z)$
```

$sr \diamond (a, b)$
$x + z = 0$

The above function *broken* is flawed in that there is no relation to which it can be safely applied. The relation *sr* is required to provide at least 3 equations for local variables, but must not exceed a contribution of 2 variables as dictated by the second relation application. As expected, our type system catches this error by attempting to impose the following inconsistent set of constraints:

$$\begin{aligned}&\mathit{broken} :: (m = n + n - 3, 0 \leqslant m \leqslant 2, 0 \leqslant n \leqslant 2, 4 \leqslant n + 1 \leqslant 4)\\ &\quad \Rightarrow \mathit{SR}\ (\mathbb{R}, \mathbb{R})\ n \rightarrow \mathit{SR}\ (\mathbb{R}, \mathbb{R})\ m\end{aligned}$$

During type checking, the Fourier-Motzkin elimination method is used to check the consistency of constraint sets [7]. The method allows one to check not only if a set of linear inequalities is satisfiable, but also finds a continuous interval for each bound variable. It is expected that this will be useful when reporting type errors to the programmer.

The elimination algorithm has worst case exponential time complexity in the number of balance variables. However, as shown by Pugh [15], 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 most exponential cases still feasible to check.

3.3 Formalising the Type System

Fig. 3 presents a small-step semantics for our calculus by way of a flattening for a system of equations. Values in our system are closed lambda-terms of the form $\lambda x.e$, signal relations encapsulating atomic equations, and atomic equations.

The notation $\{\overline{z_1}/\overline{z_2}\}$ denotes the substitution that occurs when reducing signal relation application. Our abstract treatment of equations allows us to read this notation as substituting every variable in $\overline{z_1}$ for all variables in $\overline{z_2}$, a simplification of the substitution discussed in Sect. 2.2. The symbol *fresh* denotes a fresh sequence of signal variables, used in S-SigApp2 to rename local variables to prevent name clashes during flattening (again, see Sect. 2.2).

The simplification of substitution discussed above has introduced a slight disparity between our abstract formalisation and the concrete system. In the FHM system, applying a signal relation contributing n equations to a mixed set of variables results in n mixed equations. However, during evaluation, it may be discovered that some of the equations within the signal relation do not mention both local and interface variables. Hence, the number of mixed, local, and interface equations may be refined as a result of evaluation.

This problem is avoided in our semantics by the simplification to substitution mentioned above. However, this should not pose a real problem in the concrete system either. The preservation problem is reminiscent of the record subtyping problem addressed in Peirce [14], pages 259–260. It should be possible to adapt

$$\frac{e_1 \rightsquigarrow e_2}{e_1\ e_3 \rightsquigarrow e_2\ e_3}\ (\text{S-App1}) \qquad \frac{}{(\lambda x.e_1)\ e_2 \rightsquigarrow [x \mapsto e_2]\ e_1}\ (\text{S-App2})$$

$$\frac{}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightsquigarrow [x \mapsto e_1]\ e_2}\ (\text{S-Let}) \qquad \frac{e_1 \rightsquigarrow e_2}{e_1 \diamond \overline{z} \rightsquigarrow e_2 \diamond \overline{z}}\ (\text{S-SigApp1})$$

$$\frac{\exists q_1 \in \overline{q}.\ q_1 \rightsquigarrow q_2}{\textbf{sigrel}\ \overline{z}\ \textbf{where}\ \overline{q} \rightsquigarrow \textbf{sigrel}\ \overline{z}\ \textbf{where}\ [q_1 \mapsto q_2]\ \overline{q}}\ (\text{S-SigRel})$$

$$\frac{\overline{q_2} = \{(vars(\overline{q}) \backslash \overline{z_1}) / fresh\}\ \overline{q_1}}{(\textbf{sigrel}\ \overline{z_1}\ \textbf{where}\ \overline{q_1}) \diamond \overline{z_2} \rightsquigarrow \{\overline{z_1}/\overline{z_2}\}\ \overline{q_2}}\ (\text{S-SigApp2})$$

Fig. 3. Small-step semantics.

the technique of *stupid casts* used in Pierce to solve the preservation problems that would be present in a more concrete semantics. To be more precise, one could allow a *stupid cast* of local and interface equations back into mixed equations, thus retaining the same contribution and maintaining the same constraints. We leave this alteration as future work, as the current semantics are sufficient for the purposes of this paper.

The syntax of types is similar to that of the simply-typed λ-calculus. Simple types consist of functions, signal relations, and equation types specified by $\rightarrow$, *SR*, and $I/M/LEqn$ respectively. The three varieties of equation types give distinct representations for interface, mixed, and local equations. Signal relation types and equation types are parameterised with a balance variable that denotes the number of equations a system is capable of contributing. Simple types are then parameterised by a constraint set that refines the possible interval of balance variables.

Fig. 4 gives the typing judgements for terms in our language. The rules for λ-terms, T-Var, T-Abs, and T-App are similar to those of the simply-typed λ-calculus, with the addition of constraint sets. Operations that render a constraint sets inconsistent indicate that a term is ill-typed; e.g, a judgement that involves taking the union of two consistent sets of constraints is only valid when the resulting constraint set is also consistent.

The T-Atomic judgement assigns equation types to atomic equations by examining the variables that occur in the equation. The helper function *eqkind* checks how the variables in an equation coincide with the interface variables to determine whether the equation is local, interface, or mixed.

The T-RelApp judgement assigns an equation type to a relation application. The preconditions for this judgement state that the type of the expression e appearing to the left of the application must be a signal relation. Additionally, the contribution of such a signal relation must not exceed the number of interface variables to which it is being applied. T-RelApp and T-Atomic depend on the

$$\frac{\Gamma(x) = C \Rightarrow \tau}{\Gamma \vdash x : C \Rightarrow \tau}\ (\text{T–Var}) \qquad \frac{\Gamma, x : C_1 \Rightarrow \tau_1 \vdash e : C_2 \Rightarrow \tau_2}{\Gamma \vdash \lambda x.e : C_1 \cup C_2 \Rightarrow \tau_1 \rightarrow \tau_2}\ (\text{T–Abs})$$

$$\frac{\Gamma \vdash e_1 : C_1 \Rightarrow \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : C_2 \Rightarrow \tau_2}{\Gamma \vdash e_1\ e_2 : C_1 \cup C_2 \Rightarrow \tau_1}\ (\text{T–App})$$

$$\frac{\Gamma \vdash e_1 : C_1 \Rightarrow \tau_2 \qquad \Gamma, x : C_2 \Rightarrow \tau_2 \vdash e_2 : C_1 \Rightarrow \tau_1}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : C_1 \cup C_2 \Rightarrow \tau_1}\ (\text{T–Let})$$

$$\frac{}{I \cdot \Gamma \vdash \mathit{Atomic}\ \overline{z} : \emptyset \Rightarrow \mathit{eqkind}_I(\overline{z}, 1)}\ (\text{T–Atomic})$$

$$\frac{\Gamma \vdash e : C \Rightarrow \mathit{SR}\ \mathbb{R}^m\ n \qquad |\overline{z}| \geqslant n}{I \cdot \Gamma \vdash e \diamond \overline{z} : C \Rightarrow \mathit{eqkind}_I(\overline{z}, n)}\ (\text{T–RelApp})$$

$$\frac{\begin{array}{c} L = \mathit{vars}(\overline{q}) \setminus \overline{z} \qquad \overline{z} \cdot \Gamma \vdash \overline{q} : \overline{C} \Rightarrow \overline{\tau} \qquad n_X = \Sigma\{\ b \mid X\mathit{Eqn}\ b \in \overline{\tau}\ \} \\ C = \{n = n_I + n_L + n_M - |L|, 0 \leqslant n \leqslant |\overline{z}|, n_I \leqslant |\overline{z}|, n_L \leqslant |L|, n_L + n_M \geqslant |L|\} \end{array}}{\Gamma \vdash \mathbf{sigrel}\ \overline{z}\ \mathbf{where}\ \overline{q} : \bigcup \overline{C} \cup C \Rightarrow \mathit{SR}\ \mathbb{R}^{|\overline{z}|}\ n}$$

$$\mathit{eqkind}_I(Z, n) = \begin{cases} \mathit{IEqn}\ n & \text{if } \emptyset \subset Z \subseteq I \\ \mathit{LEqn}\ n & \text{if } Z \cap I = \emptyset \\ \mathit{MEqn}\ n & \text{otherwise} \end{cases}$$

Fig. 4. Typing rules.

read-only environment I which stores the set of interface variables the equations range over.

The final judgement assigns signal relation types to **sigrel** constructs and calculates constraints on the fresh balance variable of that signal relation. The first precondition defines the set of variables local to the relation. The second precondition is a pointwise judgement over the set of equations. The third precondition sums the number of equations of a given form in $\overline{q}$ specified by the parameter X, where $X \in \{I, L, M\}$. Finally, using the previous three conditions, a set of constraints is generated for the balance variables occurring in the type.

We have identified two key properties of soundness for our type system with respect to the semantics. Firstly, the preservation of types under evaluation for **sigrel** constructs ensures that flattening a modular system of equations does not alter the contribution of the system. Formally, if **sigrel** $\overline{z}$ **where** $\overline{q_1} \rightsquigarrow$ **sigrel** $\overline{z}$ **where** $\overline{q_2}$, and **sigrel** $\overline{z}$ **where** $\overline{q_1}$ $: C \Rightarrow SR\ \ r^{|\overline{z}|}\ n$, where C is a consistent set of constraints, then **sigrel** $\overline{z}$ **where** $\overline{q_2}$ $: C \Rightarrow SR\ \ r^{|\overline{z}|}\ n$. Hence, the contribution of the sets of equations q_1 and q_2 is equal under the same set of interface variables $\overline{z}$.

Secondly, a system can only be completely reduced to a simple set of equations if the top-level **sigrel** construct abstracts over an empty set of signal variables. In these circumstances, a fully assembled system should contribute no equations as no more signal variables will be introduced. Formally, if **sigrel** $\emptyset$ **where** $\overline{q_1}$: $C \Rightarrow SR$ () n, and C is consistent, then C should resolve the interval of n to [0,0].

At this point, it is interesting to note the equational embedding effectively operates as a form of heterogeneous meta-programming; a modular system of equations is first evaluated to flat set of equations which is then transformed into a program that is used to solve for the unknowns of the original system. Hence, the balance and structure of a system of equations are really properties of the flattened system of equations that rule out (a class of) *second stage* run-time/simulation-time problems. Hence, a soundness statement regarding balance and structure falls to the meta-theory of a type system at the second stage. In summary, attempting to capture these properties during the initial phase make the soundness properties of our system are quite unusual. As such, we leave the investigation of soundness of other structural properties as future work.

The type checking algorithm has been implemented in the dependently typed programming language Agda [13]. The source code can be found on the primary authors website at `http://cs.nott.ac.uk/~jjc`. The implementation guarantees that the algorithm is both total and termination. It should be noted that the function for computing the most general unifier of two types is postulated. We have yet to implement the semantics and prove that these are sound with respect to the typing judgements, this is left as future work.

4 Related Work

4.1 Modelica

Modelica, as of version 3.0 [9], requires that models be locally balanced. This is much more restrictive than our approach as components that are individually unbalanced may still be combined to produce a balanced system. When unbalanced components are needed, the current Modelica approach is to declare them as such, turning of all balance checking for that component. Moreover, models are not first-class entities in Modelica which simplifies the static checking.

4.2 Bunus & Fritzon

Bunus and Fritzon [2] describe an analysis technique for pinpointing problems with systems of equations developed in equation-based modelling languages such as Modelica. They look at structural properties, as we do, but, to allow fine-grained localisation, in much more detail by considering incidence matrices (which variables occur in which equations). This is only possible by analysing fully assembled systems, meaning the technique is primarily suitable for debugging. It could even be used *during* simulation to catch problems with structurally dynamic systems. Thus, this work is in many ways complementary to ours.

4.3 Structural Constraint Delta

Broman et al. [1] have developed a type system called structural constraint delta (C_Δ). The type system is developed for a simplified version of Modelica: Featherweight Modelica. The C_Δ represents the difference between the number of unknowns and the number of equations in an instance of a component. Hence, C_Δ improves upon the Modelica approach by allowing models to be unbalanced, provided that a fully assembled system is balanced. As the type (class) of a constituent component is always manifest, and as the rules for subtyping are such that a replaceable component can only be replaced by one having the same C_Δ, component balances can always be computed in a bottom-up fashion.

In contrast, the type system presented in this paper does not rely on manifest type information. Furthermore, it supports a more flexible notion of balance as, if there are more than one component parameter, what matters is the collective number of contributed equations, not the numbers contributed individually.

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. [11] and Broman, with the latter giving the first detailed account.

4.4 Structural Types

Nilsson [10] outlines an approach to static checking that safeguards against a much wider class of errors than what is possible by just considering the balance. This is done by making an approximation of the incidence matrix part of the type of an equation system fragment, allowing structural singularities to be detected in many cases and thus approaching the capabilities of Bunus and Fritzon's technique, while retaining the capability of checking fragments in isolation.

While Nilsson presents the work within the context of FHM, he forgoes the consideration of first-class models, concentrating on the handling of static models. In contrast, the type system presented here handles first-class models, but cannot find as many problems.

5 Future Work

The type system presented in this paper captures the essence of the idea of balance checking in a setting with first-class equation system fragments. The system is abstract, but as such a suitable starting point for a type system for any such language. There are two imminent avenues for developing this work further. One is to elaborate the system so as to bring it closer to a system suitable for a concrete language like FHM. Handling of compound signal variables such as matrices should also be considered, as the size of matrices can affect the balance if equations between matrices is supported. The other avenue is to formalise the system and the dynamic semantics to prove soundness.

6 Conclusion

In this paper, we presented a type system for modular systems of equations capable of detecting classes of errors related to the equation-variable balance. Components can be analysed in isolation, rather than requiring assembly into a complete system of equations first, thus allowing over- and under-constrained systems to be detected early, aiding error localisation. First-class equation system fragments are supported. Our system thus lays down the foundations for a practical yet strong type system. The context of this work is equation-based, non-causal modelling, but the ideas should be readily adaptable to other settings.

Acknowledgments The authors would like to thank David Broman, Neil Sculthorpe, and the anonymous reviewers for helpful and constructive feedback.

References

1. D. Broman, K. Nyström, and P. Fritzson. Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta. In *GPCE*, New York, NY, USA, 2006. ACM.
2. Peter Bunus and Peter Fritzson. A Debugging Scheme for Declarative Equation Based Modeling Languages. *Lecture Notes in Computer Science*, 2001.
3. N. B. Conkwright. *Introduction to the Theory of Equations.* Ginn, Boston, 1957.
4. T. Freeman and F. Pfenning. Refinement Types for ML. In *PLDI*, 1991.
5. G. Giorgidze and H. Nilsson. Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In Francesco Casella, editor, *Proceedings of the 7th International Modelica Conference.* Linköping Electronic Conference Proceedings, 2009.
6. S. Jones. *Haskell 98 Language and Libraries: the Revised Report.* 2003.
7. H. Kuhn. Solvability and Consistency for Linear Equations and Inequalities. *American Mathematical Monthly*, 63, 1956.
8. J. McKinna, T. Altenkirch, and C. McBride. Why Dependent Types Matter. *ACM SIGPLAN Notices*, 41(1), 2006.
9. The Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification Version 3.2*, 2010.
10. H. Nilsson. Type-Based Structural Analysis for Modular Systems of Equations. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, Linköping Electronic Conference Proceedings, 2008.
11. H. Nilsson, J. Peterson, and P. Hudak. Functional Hybrid Modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, lncs, 2003.
12. H. Nilsson, J. Peterson, and P. Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. *Simulation News Europe*, 2007.
13. U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007.
14. Benjamin Pierce. *Types and Programming Languages.* The MIT Press, 2002.
15. W. Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing 91*, 1991.