Embedding a Functional Hybrid Modelling Language in Haskell

George Giorgidze and Henrik Nilsson

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

Abstract. In this paper we present the first investigation into the implementation of a Functional Hybrid Modelling language for non-causal modelling and simulation of physical systems. In particular, we present a simple way to handle connect constructs: a facility for composing model fragments present in some form in most non-causal modelling languages. Our implementation is realised as a domain-specific language embedded in Haskell. The method of embedding employs quasiquoting, thus demonstrating the effectiveness of this approach for languages that are not suitable for embedding in more traditional ways. Our implementation is available on-line, and thus the first publicly available prototype implementation of a Functional Hybrid Modelling language.

1 Introduction

Functional Hybrid Modelling (FHM) [21,22] is a new approach to designing *non-causal modelling languages* [4]. This class of languages is intended for modelling and simulation of systems that can be described by Differential Algebraic Equations (DAEs). Examples primarily include *physical systems*, such that electrical (see Figure 1), mechanical, hydraulic, and thermal systems. But any domain where models can be expressed in terms of DAEs, or any combination of such domains, is fine. *Non-causal* refers to treating equations as being *undirected*: an equation can be used to solve for any of the variables occurring in it. This is in contrast to *causal* modelling languages where equations are restricted to be *directed*: only known variables on one side of the equal sign, and only unknown variables on the other. Consequently, in a causal language, an equation is effectively little more than a (possibly parallel) assignment statement.

The main advantage of the causal languages is that simulation is relatively straightforward thanks to equations being directed. The advantages of the non-causal languages over the causal ones include that models are more *reusable* (as the equations can be used in many ways) and more *declarative* (as the modeller can focus on *what* to model, worrying less about *how* to model it to enable simulation) [4]. Modelica [18] is a prominent, state-of-the-art representative of the class of non-causal modelling and simulation languages.

S.-B. Scholz and O. Chitil (Eds.): IFL 2008, LNCS 5836, pp. 138–155, 2011.

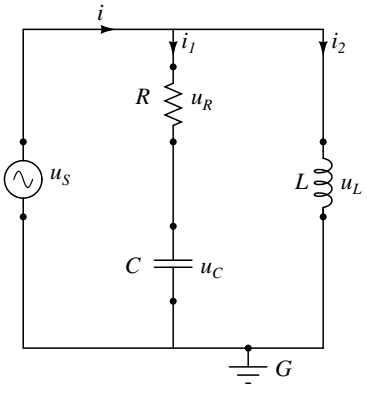

Fig. 1. A simple electrical circuit

However, current non-causal languages have their drawbacks. For example, the language designs tend to be relatively complex, being based around class systems inspired by object-oriented programming languages[1]. Modelling support for *hybrid systems* (systems that exhibit both continuous and discrete dynamic behaviour) is often not very declarative. To facilitate efficient simulation, the languages are designed on the assumption that the model is translated into simulation code once and for all. This limits the possibilities for describing *structurally dynamic systems*, where the structure evolve over time through the addition or removal of components, and in particular *highly* structurally dynamic systems, where the number of structural configurations (the *modes*) is large, unbounded, or impossible to determine in advance.

The idea of FHM is to enrich a purely functional language with a few key abstractions for supporting hybrid, non-causal modelling. By leveraging the abstraction power of the functional host language, much of the scaffolding of current non-causal modelling languages becomes redundant, and it becomes possible to describe highly structurally dynamic systems [21]. Our hypothesis is that the FHM approach will result in non-causal modelling languages that are relatively simple, have clear, purely declarative semantics, and, aided by this, advance the state of the art by supporting e.g. highly structurally dynamic systems, thus addressing the problems of current non-causal designs.

FHM is inspired by Functional Reactive Programming (FRP) [8,26], in particular Yampa [20,12]. Yampa is an *Embedded Domain-Specific Language* (EDSL) [11]. The host language is Haskell, and the central domain-specific abstraction is the *signal function*: first-class functions on signals (time-varying values). In a dynamic systems setting, signal functions are essentially directed differential equations on signals. Yampa further provides *switch constructs* that are capable

[1] Indeed, non-causal languages are often referred to as *object-oriented modelling languages*, even though they do not feature central object-oriented concepts like mutable objects and methods.

of switching between signal functions during simulation. Yampa thus supports *causal* modelling of highly structurally dynamic systems [6,5,10].

The key insight of FHM is that *non-causal* modelling can be supported by enriching a functional language with first-class *signal relations*: relations on signals described by undirected differential equations. While the idea of FHM is not predicated on an embedded implementation, we are currently pursuing an EDSL approach for FHM to enable us to focus on research problems related to non-causal modelling as opposed to functional language implementation.

This paper contributes at two levels: firstly to the area of design and implementation of declarative, non-causal modelling languages; secondly, and more generally, to the area of EDSLs. Specifically:

- We present the first implementation of *Hydra*[2], a language following the FHM approach. The present implementation covers all key aspects of the continuous[3] part of the FHM paradigm.
- We describe how to translate *connect constructs*, a facility for composing model fragments present in most non-causal modelling languages, into equations in the FHM setting. Our method is simpler than in other non-causal languages like Modelica [18], although it remains to be seen to what extent our approach can be used outside of FHM.
- We show how *quasiquoting* [16] makes a convenient embedded implementation of a non-causal modelling language possible, thus extending the EDSL approach to a new class of languages and further demonstrating the effectiveness of quasiquoting for embedding domain-specific languages, as pioneered by Mainland et al. [17]. This approach is particularly relevant for languages that are sufficiently different from the host language that more conventional methods of embedding, such as combinator libraries [11], are a poor fit.

The rest of this paper is organised as follows: In Section 2 we outline fundamental concepts of FHM and Hydra. In Section 3 we implement Hydra as a domain-specific language embedded in Haskell. Section 4 considers related work. Finally, we discuss future work in Section 5, notably support for highly structurally dynamic hybrid systems, and give conclusions in Section 6.

2 Fundamental Concepts of FHM and Hydra

2.1 Signals and Signal Functions

Before turning to FHM, let us review 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 some type α:

$$Signal\ \alpha \approx Time \to \alpha$$

[2] The source code of the prototype is publicly available on-line (`http://cs.nott.ac.uk/~ggg/`) under the open source BSD license.

[3] E.g., structurally dynamic systems are not supported at present.

Time is continuous, and is represented as a non-negative real number. The type parameter α specifies the type of values carried by the signal. A *signal function* can be thought of as a function from signal to signal:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

However, signal functions are abstract, and to ensure that they are realisable, they are additionally required to be *temporally causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$.

Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of a signal function. Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. The first class nature of signal functions enables programming of highly structurally dynamic systems using Yampa's switching combinators [20].

2.2 Signal Relations

FHM generalises the notion of signal functions to *signal relations*. A signal relation is simply a relation on signals. Stating that some signals are in a particular relation to each other imposes *constraints* on those signals. Assuming these constraints can be satisfied, this allows some of the signals to be determined in terms of the others depending on which signals are known and unknown in a given context. That is, signal relations are non-causal, unlike signal functions where the knowns and unknowns (inputs and outputs) are given a priori. Like signal functions in Yampa, signal relations are first class entities in Hydra.

Because a product of signals, say $Signal\ \alpha$ and $Signal\ \beta$, is isomorphic to a signal of the product of the carried types, in this case $Signal\ (\alpha, \beta)$, unary signal relations actually suffice for handling signal relations of any arity. We thus introduce the type $SR\ \alpha$ for a signal relation on a signal of type α.

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

$$SR\ \alpha \approx Signal\ \alpha \to Bool$$

Solving a relation thus means finding a signal that satisfies the predicate. As an example, equality is a binary signal relation:

$$(=) :: SR\ (\alpha, \alpha)$$
$$(=)\ s \approx \forall\ t.fst\ (s\ t) \equiv snd\ (s\ t)$$

Hydra adopts the following syntax for defining signal relations (inspired by the arrow notation [24]):

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

The pattern binds *signal variables* that scope over the equations that follow. The equations are DAEs stated using *signal relation application* (the operator $\diamond$). Signal relation application is how the constraints embodied by a signal relation are imposed on particular signals:

$sr \diamond s$

Equations must be well typed. In this example, if sr has type $SR\ \alpha$, s must have type $Signal\ \alpha$. Additionally, Hydra provides a more conventional-looking syntax for equality between signals. For example: $a * x + b = 0$ is equivalent to $(=) \diamond (a * x + b, 0)$.

2.3 The Hydra Syntax

The abstract syntax of Hydra is given below. The aspects that have not yet been discussed, such as flow variables and the connect construct, are covered in the following sections. Note that, because Hydra is implemented as an embedded language, we are able to reuse Haskell for the functional part, as described in Section 3.

⟨*SigRel*⟩ ::= `sigrel` ⟨*Pattern*⟩ `where` { ⟨*ListEquation*⟩ }
⟨*Pattern*⟩ ::= ⟨*PatNameQual*⟩ ⟨*Identifier*⟩
| `(` ⟨*ListPattern*⟩ `)`
⟨*ListPattern*⟩ ::= ϵ
| ⟨*Pattern*⟩
| ⟨*Pattern*⟩ `,` ⟨*ListPattern*⟩
⟨*PatNameQual*⟩ ::= ϵ
| `flow`
⟨*Equation*⟩ ::= ⟨*SigRel*⟩ <> ⟨*Expr*⟩
| ⟨*Expr*⟩ = ⟨*Expr*⟩
| `connect` ⟨*Identifier*⟩ ⟨*Identifier*⟩ ⟨*ListIdentifier*⟩
⟨*ListEquation*⟩ ::= ϵ
| ⟨*Equation*⟩
| ⟨*Equation*⟩ `;` ⟨*ListEquation*⟩
⟨*ListIdentifier*⟩ ::= ϵ
| ⟨*Identifier*⟩ ⟨*ListIdentifier*⟩
⟨*Expr*⟩ ::= ⟨*Expr*⟩ ⟨*Expr*⟩
| ⟨*Expr*⟩ + ⟨*Expr*⟩
| ⟨*Expr*⟩ − ⟨*Expr*⟩
| ⟨*Expr*⟩ `*` ⟨*Expr*⟩
| ⟨*Expr*⟩ `/` ⟨*Expr*⟩
| ⟨*Expr*⟩ `^` ⟨*Expr*⟩
| − ⟨*Expr*⟩
| ⟨*Identifier*⟩
| ⟨*Integer*⟩
| ⟨*Double*⟩
| `(` ⟨*ListExpr*⟩ `)`
⟨*ListExpr*⟩ ::= ϵ
| ⟨*Expr*⟩
| ⟨*Expr*⟩ `,` ⟨*ListExpr*⟩

Instead of semicolons and curly braces, the modeller can use layout syntax in the same way as in Haskell. All examples in this paper use layout. We used the BNF Converter [25], a compiler front-end generator taking a labelled BNF grammar as input, to generate the lexer and parser of Hydra.

3 Embedding Hydra

In this section we implement Hydra as a domain-specific language embedded in Haskell. The method of embedding is inspired by Mainland et al. [17] and employs quasiquoting [16].

3.1 Why Quasiquoting?

Because of the non-causal nature of Hydra, an implementation needs the ability to manipulate models symbolically; e.g., to solve parts of models symbolically, to transform models into a form suitable for numerical simulation, and to compile models to efficient simulation code. This suggests a *deep embedding* where embedded language terms are represented as Abstract Syntax Trees (ASTs) [13,7,1].

One way to achieve this is to design a combinator library for building ASTs representing the embedded language terms. However, the use of combinators implies that the domain-specific syntax fundamentally needs to conform to the syntax of Haskell. While this can work really well in many cases (thanks to clever use of overloading, carefully crafted infix operators, and the like), the result is not always satisfying. Indeed, this observation has lead to proposals for syntactic extensions of Haskell, such as the arrow notation [24], to allow certain kinds of combinator libraries to be used in a more convenient way.

Hydra is an example of a language that does not quite fit with Haskell's syntax (or established extensions like the arrow syntax). Designing a clean combinator library without sacrificing certain aspects of the desired syntax, or introducing distracting "syntactic noise", proved to be hard. Instead, we opted to use *quasiquoting*, a meta-programming feature provided by Glasgow Haskell Compiler (GHC) as of version 6.10, which allows us to use almost exactly the syntax we want at the cost of having to provide our own parser. This parser takes a string in the domain-specific concrete syntax and returns the corresponding AST, additionally allowing for ASTs resulting from evaluating Haskell expressions to be "spliced in" where needed. Our embedding of Hydra thus allows a modeller to use a syntax that is very close to that proposed in earlier FHM-related publications [21,22].

This embedding technique clearly separates embedded language terms (in our case, non-causal Hydra models) from host language terms (in our case, the genuine Haskell expressions). Specifically, signal variables in Hydra are clearly separated from Haskell variables, which is important as signal variables must only be used in a **sigrel** abstraction.

3.2 The Haskell Embedding

Let us introduce the Haskell embedding of Hydra by modelling the circuit in Figure 1. We first define a *twoPin* model: a signal relation that captures the common behaviour of electrical components with two connectors (see Figure 2):

```
twoPin :: SigRel
twoPin = [$hydra|
  sigrel ((flow p_i, p_v), (flow n_i, n_v), u) where
    p_v − n_v = u
    p_i + n_i = 0
|]
```

The signal variables p_i and p_v represent the current into the component and the voltage at the positive pin. The signal variables n_i and n_v represent the current into the component and the voltage at the negative pin. The signal variable u represents the voltage drop across the electrical component. Signal variables in the **sigrel** pattern qualified as **flow** are called *flow* signal variables. Signal variables without any qualifier are called *potential* signal variables. The distinction between flow and potential variables is central to the meaning of the **connect** construct as discussed in Section 3.3.

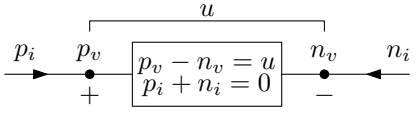

Fig. 2. An electrical component with two connectors

The symbols [**$hydra**| and |] are the quasiquotes. At compile time, GHC applies the user-defined parsing function named in the opening quote to the text between the quotes. Here, the function is called *hydra*. It has type *String* → *SigRel* and parses the concrete version of the Hydra syntax defined in Section 2.3. Values of type *SigRel* are ASTs representing Hydra signal relations. This enables the embedded Hydra compiler to process them symbolically and ultimately compile them into simulation code.

We can now use *twoPin* to define a model for a resistor parametrised with respect to the resistance. Note that a parametrised model simply is an ordinary function returning a signal relation:

```
resistor :: Double → SigRel
resistor r = [$hydra|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $r$ ∗ p_i = u
|]
```

Expressions between dollar signs are *antiquoted* Haskell expressions. All variables in antiquoted expressions must be in the Haskell scope. Using this technique, a modeller can splice in Haskell expressions in the Hydra models.

The current implementation only allows antiquoting of Haskell expressions of type *SigRel* in the left hand side of signal relation applications and of type *Double* in signal expressions. The result spliced in to the left in a signal relation application is thus an entire AST representing a signal relation, as required by the abstract syntax (see Section 2.3). Antiquoted expressions must have the correct type, i.e *SigRel* and *Double* respectively. Type-incorrect, antiquoted expressions are detected by GHC at compile time.

In this case, note how antiquoting is used to splice in a copy of the *twoPin* model; that is, its equations are *reused* in the context of the resistor model. Alternatively, this can be viewed as defining the resistor model by extending the *twoPin* model with an equation that characterises the specific concrete electrical component, in this case Ohm's law.

To clearly see how *twoPin* contributes to the definition of *resistor*, let us consider what happens when the resistor model is *flattened* as part of flattening of a complete model, a transformation that is described in detail in Section 3.4. Intuitively, flattening can be understood as "inlining" of applied signal relations. Thus, the arguments of a signal relation application is substituted into the body of the applied signal relation, and the entire application is then replaced by the instantiated signal relation body. In our case, the result of flattening the signal relation *resistor* 10 is:

$$
\begin{aligned}
&\textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where}\\
&\quad p_v - n_v = u\\
&\quad p_i + n_i = 0\\
&\quad 10 * p_i = u
\end{aligned}
$$

Models for an inductor, a capacitor, a voltage source and a ground are defined similarly:

$$
\begin{aligned}
&\mathit{inductor} :: \mathit{Double} \to \mathit{SigRel}\\
&\mathit{inductor}\ l = [\textbf{\$hydra}|\\
&\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where}\\
&\qquad \$\mathit{twoPin}\$ \diamond ((p_i, p_v), (n_i, n_v), u)\\
&\qquad \$l\$ * \mathit{der}\ p_i = u\\
&|]\\
&\mathit{capacitor} :: \mathit{Double} \to \mathit{SigRel}\\
&\mathit{capacitor}\ c = [\textbf{\$hydra}|\\
&\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where}\\
&\qquad \$\mathit{twoPin}\$ \diamond ((p_i, p_v), (n_i, n_v), u)\\
&\qquad \$c\$ * \mathit{der}\ u = p_i\\
&|]\\
&\mathit{vSourceAC} :: \mathit{Double} \to \mathit{Double} \to \mathit{SigRel}\\
&\mathit{vSourceAC}\ v\ f = [\textbf{\$hydra}|
\end{aligned}
$$

```
    sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
      $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
      u = $v$ * sin (2 * $π$ * $f$ * time)
|]
ground :: SigRel
ground = [$hydra|
  sigrel (flow p_i, p_v) where
    p_v = 0
|]
```

3.3 Non-causal Connections

To facilitate composition of signal relations, Hydra provides a Modelica-inspired **connect** construct. Using this, a complete model for the circuit of Figure 1 can be defined as follows:

```
simpleCircuit :: SigRel
simpleCircuit = [$hydra|
  sigrel (flow i, u) where
    $vSourceAC 1 1$ ◇ ((acp_i, acp_v), (acn_i, acn_v))
    $resistor 1$    ◇ ((rp_i, rp_v), (rn_i, rn_v))
    $inductor 1$    ◇ ((lp_i, lp_v), (ln_i, ln_v))
    $capacitor 1$   ◇ ((cp_i, cp_v), (cn_i, cn_v))
    $ground$        ◇ (gp_i, gp_v)
    connect acp_i rp_i lp_i
    connect acp_v rp_v lp_v
    connect rn_i cp_i
    connect rn_v cp_v
    connect acn_i cn_i ln_i gp_i
    connect acn_v cn_v ln_v gp_v
    i = acp_i
    u = acp_v − acn_v
|]
```

Note how the above code is a direct textual representation of how the components are connected in the example circuit.

In the setting of Hydra, the **connect** construct is just syntactic sugar with the following rules[4]:

- The special keyword **connect** takes two or more signal variables.
- A signal variable may not appear in more than one connect statement.
- Connection of flow signal variables with potential signal variables is not allowed.

[4] These rules may be relaxed in the future to allow connection of, for example, aggregated signal variables.

For connected flow variables, sum-to-zero equations are generated. In the electrical domain, this corresponds to Kirchhoff's current law. For potential variables, equality constraints are generated. In the electrical domain, this asserts that the voltage at connected pins is equal. The connect constructs of *simpleCircuit* are thus expanded to the following equations:

$$
\begin{aligned}
acp_i &+ rp_i &+ lp_i &= 0 \\
acp_v &= rp_v &= lp_v \\
rn_i &+ cp_i &&= 0 \\
rn_v &= cp_v \\
acn_i &+ cn_i &+ ln_i &+ gp_i = 0 \\
acn_v &= cn_v &= ln_v &= gp_v
\end{aligned}
$$

Note that the notion of flows and potentials are common to many physical domains. For example, the Modelica standard library employs connections for electrical, hydraulic, and mechanical applications, among others.

In Hydra, the expansion of connect constructs into the sum-to-zero and equality constraints is straightforward. In particular, note that all signal variables are counted positively in the sum to zero equations. This is different from Modelica [18] where a special "rule of signs" is used to determine which flow variables go with a plus sign and which go with a minus sign. Hydra obviates the need for a rule of signs by treating flow signal in signal relation applications specially, thus keeping the generation of connection equations simple. The idea is to consider a flow variable in a **sigrel** pattern as two variables, one internal and one external, related by the equation

$$i = -i'$$

where i is the internal variable and i' is the external variable. This way, flows are always directed from an interface into a component, as it were, making it possible to always count flows into connection nodes as being positive.

3.4 Model Flattening

Once the quasiquoting has been processed and the **connect** constructs translated into equations, the model is turned into a single system of equations through a process called *flattening*. This is accomplished by substituting the arguments of signal relation applications into the body of the applied signal relation. The following example illustrates the process. It also shows how flow variables are handled.

$$
\begin{aligned}
&par :: SigRel \to SigRel \to SigRel \\
&par\ sr1\ sr2 = [\mathbf{\$hydra}| \\
&\quad \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v))\ \mathbf{where} \\
&\qquad \$sr1\$ \diamond ((p1_i, p1_v), (n1_i, n1_v)) \\
&\qquad \$sr2\$ \diamond ((p2_i, p2_v), (n2_i, n2_v)) \\
&\qquad \mathbf{connect}\ p_i\ \ p1_i\ \ p2_i
\end{aligned}
$$

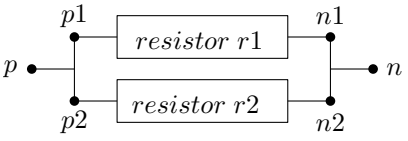

Fig. 3. Two resistors connected in parallel

```
    connect p_v p1_v p2_v
    connect n_i n1_i n2_i
    connect n_v n1_v n2_v
|]
```

The function *par* takes two models of electrical components and returns a model where these components are connected in parallel. We use this function to model the component in Figure 3 and show that:

$$par\ (resistor\ r1)\ (resistor\ r2) \equiv resistor\ ((r1 * r2)\ /\ (r1 + r2))$$

First we perform the substitution of function arguments:

```
par (resistor r1) (resistor r2) = [$hydra|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $resistor r1$ ◇ ((p1_i, p1_v), (n1_i, n1_v))
    $resistor r2$ ◇ ((p2_i, p2_v), (n2_i, n2_v))
    connect p_i  p1_i  p2_i
    connect p_v  p1_v  p2_v
    connect n_i  n1_i  n2_i
    connect n_v  n1_v  n2_v
|]
```

We then generate connection equations and unfold the signal relation applications:

```
par (resistor r1) (resistor r2) = [$hydra|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((−p1_i, p1_v), (−n1_i, n1_v), u1)
    $r1$ ∗ (−p1_i) = u1
    $twoPin$ ◇ ((−p2_i, p2_v), (−n2_i, n2_v), u2)
    $r2$ ∗ (−p2_i) = u2
    p_i + p1_i + p2_i = 0
    p_v   = p1_v
    p1_v = p2_v
    n_i + n1_i + n2_i = 0
    n_v   = n1_v
    n1_v = n2_v
|]
```

By further unfolding of signal relation applications we get:

$$
\begin{aligned}
&\mathit{par}\ (\mathit{resistor}\ \mathit{r1})\ (\mathit{resistor}\ \mathit{r2}) = [\mathbf{\$hydra}| \\
&\quad \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v))\ \mathbf{where} \\
&\qquad p1_v - n1_v = \mathit{u1} \\
&\qquad (-(-p1_i)) + (-(-n1_i)) = 0 \\
&\qquad \$\mathit{r1}\$ * (-p1_i) = \mathit{u1} \\
&\qquad p2_v - n2_v = \mathit{u2} \\
&\qquad (-(-p2_i)) + (-(-n2_i)) = 0 \\
&\qquad \$\mathit{r2}\$ * (-p2_i) = \mathit{u2} \\
&\qquad p_i + p1_i + p2_i = 0 \\
&\qquad p_v \ \ = p1_v \\
&\qquad p1_v = p2_v \\
&\qquad n_i + n1_i + n2_i = 0 \\
&\qquad n_v \ \ = n1_v \\
&\qquad n1_v = n2_v \\
&|]
\end{aligned}
$$

We note that $u = \mathit{u1} = \mathit{u2} = p_v - n_v$ and simplify:

$$
\begin{aligned}
&\mathit{par}\ (\mathit{resistor}\ \mathit{r1})\ (\mathit{resistor}\ \mathit{r2}) = [\mathbf{\$hydra}| \\
&\quad \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v))\ \mathbf{where} \\
&\qquad p_v - n_v = u \\
&\qquad p1_i + n1_i = 0 \\
&\qquad \$\mathit{r1}\$ * (-p1_i) = u \\
&\qquad p2_i + n2_i = 0 \\
&\qquad \$\mathit{r2}\$ * (-p2_i) = u \\
&\qquad p_i + p1_i + p2_i = 0 \\
&\qquad n_i + n1_i + n2_i = 0 \\
&|]
\end{aligned}
$$

Solving and eliminating the variables $p1_i$, $n1_i$, $p2_i$ and $n2_i$ yields:

$$
\begin{aligned}
&\mathit{par}\ (\mathit{resistor}\ \mathit{r1})\ (\mathit{resistor}\ \mathit{r2}) = [\mathbf{\$hydra}| \\
&\quad \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v))\ \mathbf{where} \\
&\qquad p_v - n_v = u \\
&\qquad p_i + n_i = 0 \\
&\qquad \$(\mathit{r1} * \mathit{r2}) \,/\, (\mathit{r1} + \mathit{r2})\$ * p_i = u \\
&|]
\end{aligned}
$$

This is what we expected. This example also demonstrated how the first class nature of signal relations enables us to define a signal relation parametrised over other signal relations. Such models are called *higher-order non-causal models*. Broman et al. provide other motivating examples for the use of this modelling technique [3].

3.5 Simulating Hydra Models

Figure 4 illustrates the stages involved in compiling Haskell-embedded Hydra models into executable simulation code.

In the first stage, GHC is used to compile the Haskell-embedded Hydra models into the executable program. GHC first transforms all quasiquoted Hydra models into the ASTs, then type checks the Haskell program, and finally produces the executable binary.

In the second stage, the executable program is run. This compiles the Hydra ASTs into a single Modelica class. The executable internally performs type checking of the models, desugars connection statements, flattens the top level signal relation and translates it to Modelica code. For example, this type checking ensures that the type of an applied signal relation and the signal it is applied to agree. Separate type checking is necessary, because we have chosen not to embed Hydra's type system into Haskell's type system. GHC's Haskell type checking phase only guarantees that Hydra's signal relations are syntactically correct.

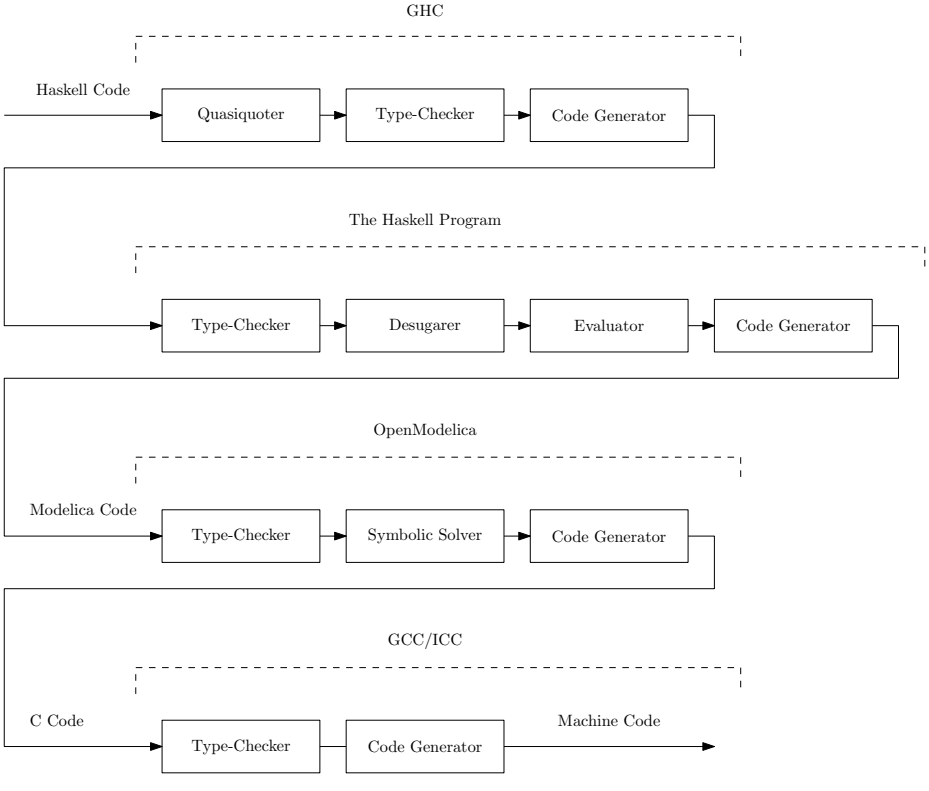

Fig. 4. The translation process in the prototype implementation from Haskell-embedded Hydra models to executable simulation code

This is well known issue in meta-programming using quasiquoting [16]. Because the type errors are detected before the start of the simulation, this is not a major drawback.

In the third phase, the OpenModelica compiler [9] is invoked to compile the generated Modelica code into the C code that then in the forth phase is compiled into the executable binary using the GNU C compiler. Finally, the actual simulation can be carried out by running this last executable.

4 Related Work

4.1 Flask

The implementation of Hydra is directly inspired by recent work on Flask [17]. Flask is a domain-specific language embedded in Haskell for programming applications for *sensor networks*: collections of devices with very limited computational and memory resources. The authors note that FRP is a suitable programming model for programming such networks. However, currently available Haskell embeddings cannot be used in this domain because that would necessitate running a Haskell run-time system. Such run-time systems are too large and heavy for typical sensor network nodes.

Flask thus uses a different embedding approach. Haskell is only used for meta-programming, not for running the actual programs. This is accomplished through quasiquoting. Haskell is used to manipulate program fragments written in the object language, which can be either Red, a restricted subset of Haskell where all functions are total and memory allocation is bounded, or nesC, a dialect of C used for programming sensor networks. NesC is provided for easy integration with existing sensor network code. The terms in the object-languages are composed using FRP-inspired combinators.

A Flask program is first compiled into the nesC code. This code is then compiled using the nesC compiler that generates code ready to be deployed on sensor network nodes. This embedding approach caught our attention as it allows for embedding of languages that are far removed from the host language, clearly separates the embedded language from the host language, and makes it possible to employ standard compiler technology to handle parsing, type checking, and code generation.

4.2 Modelling Kernel Language

Broman [2] is developing Modelling Kernel Language (MKL) that is intended to be a core language for non-causal modelling languages (e.g. Modelica). Broman takes a functional approach to non-causal modelling, which is similar to the FHM approach [21]. One of his main goal is to provide formal semantics of the core language. Currently, the formal semantics of MKL is based on an untyped λ-calculus.

Similarly to Hydra, MKL provides a λ-abstraction for defining functions and an abstraction similar to **sigrel** for defining non-causal models. Both functions

and non-causal models are first class entities in MKL. This enables higher-order non-causal modelling. The similarity of the basic abstractions leads to a very similar modelling style in both languages.

However, there are a number of differences as well. MKL introduces a special notion of *connector* to state non-causal connections. This is not the case in Hydra where the connect construct works on signal variables. As a result, both the syntax and the semantics (generation of connection equations) of the connection constructs of the two languages differ. In particular, in the formal semantics of MKL the λ-calculus is extended with *effectful* constructs to handle non-causal connections; i.e., functions in MKL have an effect and are not pure. In contrast, functions in Hydra are pure and non-causal connections are also handled in a purely functional manner. Broman's ultimate goal, though, is to provide a pure, declarative surface language for modelling.

4.3 FHM at Yale

Work on FHM has also been carried out at Yale by Hai Liu under the supervision of Paul Hudak [15]. This work is mostly complementary to ours, focusing on describing the *dynamic semantics* of FHM, including structural changes. Additionally, Liu developed a type system for FHM with a strict separation between ordinary variables and signal variables. This type system thus has some similarities to the arrow calculus [14], but note that Liu's work is earlier by a few years. This similarity is to be expected as FHM was inspired by Yampa which is based on arrows, and as the distinction between ordinary variables and arrow-bound variables is second nature to anyone who has programmed using Paterson's arrow syntax [24].

4.4 Non-causal Modelling and Simulation of Hybrid Systems

As one of the goals of Hydra is to support hybrid modelling, we will briefly survey some of the most closely related work in that area.

MOSILA is an extension of the Modelica language that supports the description of structural changes using object-oriented statecharts [23]. This enables modelling of structurally dynamic systems. However, the statechart approach implies that all structural modes must be specified in advance. This means that MOSILA does not support highly structurally dynamic systems.

Sol is a Modelica-like language [27]. It introduces language constructs which enable the description of systems where objects are dynamically created and deleted, with the aim of supporting modelling of highly structurally dynamic systems. At the time of writing, this work is in its very early stages and the design and implementation of the language has not yet been completed.

5 Future Work

In Section 3 we demonstrated the embedding into Haskell of Hydra, an FHM language that supports modelling with first class signal relations. The next major

step is to design and implement switching combinators capable of switching between signal relations during the simulation. This will make modelling of highly structurally dynamic systems possible. However, there are number of challenges that needs to be overcome, such as state transfer during switches and simulation code generation for highly structurally dynamic systems [21,22].

We also aim to investigate domain-specific type system aspects related to solvability of systems of equations and consistency of models in the presence of structural dynamism. The goal is to provide as many static guarantees at compile time as possible [19].

We intend to pursue our current implementation approach based on embedding and quasiquoting in our future work on extending Hydra as we have found this approach quick and flexible from an implementation perspective, while also allowing models to be written with very little syntactic "embedding noise". However, note that neither the FHM framework, nor Hydra, are predicated on this implementation approach. Ultimately, a stand-alone implementation may be the way to go.

6 Conclusions

In this paper, we showed how to realise the basic FHM notion of a signal relation and language constructs for composing signal relations into complete models as a domain-specific, deep, embedding in Haskell. We used quasiquoting, as pioneered by Mainland et al. [17], to achieve this, motivated by the fact that the syntax of the embedded language is quite far removed from Haskell, and a desire to avoid as much "syntactic embedding noise" as possible. We think quasiquoting is a promising approach for domain-specific embeddings as it, in addition to the usual benefits of embedded language implementations, allows standard compilation technology to be applied for analysis and code generation, something which can can be essential for performance reasons.

The main contribution of this paper is the first investigation into the implementation of the fundamental aspects of an FHM language. The paper is supported by the publicly available prototype implementation. It enables physical modelling with first class signal relations and can model and simulate systems with static structure. Support of highly structurally dynamic hybrid systems is the subject of future work, together with other topics outlined in Section 5.

Acknowledgements. This work was supported by EPSRC grant EP/D064554/1. Thanks to the anonymous referees for many useful suggestions. We would also like to thank Neil Sculthorpe and Zurab Khetsuriani for their helpful comments and feedback.

References

1. Augustsson, L., Mansell, H., Sittampalam, G.: Paradise: a two-stage DSL embedded in Haskell. In: ICFP 2008: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, pp. 225–228. ACM, New York (2008)

2. Broman, D.: Flow lambda calculus for declarative physical connection semantics. Technical Reports in Computer and Information Science No. 1. LIU Electronic Press (2007)
3. Broman, D., Fritzson, P.: Higher-order acausal models. In: Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, Paphos, Cyprus, pp. 59–69. LIU Electronic Press (2008)
4. 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)
5. Cheong, M.H.: Functional programming and 3D games. BEng thesis, University of New South Wales, Sydney, Australia (November 2005)
6. Courtney, A., Nilsson, H., Peterson, J.: The Yampa arcade. In: Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell 2003), Uppsala, Sweden, pp. 7–18. ACM Press, New York (2003)
7. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. Journal of Functional Programming 13(2) (2003); Updated version of paper by the same name that appeared in SAIG 2000 proceedings
8. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of ICFP 1997: International Conference on Functional Programming, pp. 163–173 (June 1997)
9. Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., Broman, D., Sandholm, A.: OpenModelica - a free open-source environment for system modeling, simulation, and teaching. In: 2006 IEEE International Symposium on Computer-Aided Control Systems Design, pp. 1588–1595 (October 2006)
10. Giorgidze, G., Nilsson, H.: Switched-on Yampa. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 282–298. Springer, Heidelberg (2008)
11. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of Fifth International Conference on Software Reuse, pp. 134–142 (June 1998)
12. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003)
13. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Proceedings of the 2nd Conference on Domain-Specific Languages, pp. 109–122. ACM Press, New York (1999)
14. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus, functional pearl (2008), `http://homepages.inf.ed.ac.uk/wadler/topics/links.html`
15. Liu, H.: CS690 report of FHM. Available from Computer Science, Yale University (May 2005)
16. Mainland, G.: Why it's nice to be quoted: Quasiquoting for Haskell. In: Haskell 2007: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop, pp. 73–82. ACM, New York (2007)
17. Mainland, G., Morrisett, G., Welsh, M.: Flask: Staged functional programming for sensor networks. In: Proceedings of the Thirteenth ACM SIGPLAN International Conference on Functional Programming (ICFP 2008), Victoria, British Columbia, Canada. ACM Press, New York (2008)
18. The Modelica Association. Modelica – A unified object-oriented language for physical systems modeling: Language Specification version 3.0 (September 2007), `http://www.modelica.org/documents/ModelicaSpec30.pdf`

19. Nilsson, H.: Type-based structural analysis for modular systems of equations. In: Fritzson, P., Cellier, F., Broman, D. (eds.) Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, Paphos, Cyprus. Linköping Electronic Conference Proceedings, vol. 29, pp. 71–81. Linköping University Electronic Press (July 2008)
20. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell 2002), Pittsburgh, Pennsylvania, USA, pp. 51–64. ACM Press, New York (2002)
21. Nilsson, H., Peterson, J., Hudak, P.: Functional Hybrid Modeling. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 376–390. Springer, Heidelberg (2002)
22. Nilsson, H., Peterson, J., Hudak, P.: Functional hybrid modeling from an object-oriented perspective. In: Fritzson, P., Cellier, F., Nytsch-Geusen, C. (eds.) Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools. Linköping Electronic Conference Proceedings, vol. 24, pp. 71–87. Linköping University Electronic Press (2007)
23. 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, Hamburg, Germany, pp. 527–535 (2005)
24. Paterson, R.: A new notation for arrows. In: Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming, Firenze, Italy, pp. 229–240 (September 2001)
25. Pellauer, M., Forsberg, M., Ranta, A.: BNF Converter: Multilingual front-end generation from labelled BNF grammars. Technical report, Computing Science at Chalmers University of Technology and Gothenburg University (September 2004)
26. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: Proceedings of PLDI 2001: Symposium on Programming Language Design and Implementation, pp. 242–252 (June 2000)
27. Zimmer, D.: Introducing Sol: A general methodology for equation-based modeling of variable-structure systems. In: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, pp. 47–56 (2008)