Yale University
Department of Computer Science

Hydra: A Functional Hybrid Modeling Language

Hai Liu

YALEU/DCS/TR-1356
May 2006

Hydra: A Functional Hybrid Modeling Language [*]

Hai Liu[†]

Abstract

This is a summary of the design of *Hydra*, a language prototype for Functional Hybrid Modeling. Hydra is used for non-causal modeling specifications, and based on first-class signal relations, complex systems can be composed by components in a functional way. Reactivity is introduced in *Hydra* to handle event and discontinuity in such hybrid systems. The experience in studying the semantics and implementing *Hydra* allows us more insights into the much unexplored field of functional hybrid modeling, so as to guide our future research.

1 Introduction

Systems that exhibit both continuous-time and discrete-time behavior are called *hybrid* systems. Such system dynamism is often viewed as discrete transitions between multiple modes, each of which describes a continuous system state. Many areas about hybrid systems have been studied, including modeling, simulation, sensitivity analysis, and numerical optimization [6, 1, 4, 2, 23, 3].

A number of formalisms have been proposed to model hybrid systems [1, 4, 15, 2, 8, 6, 7], among which hybrid automata [Alur et al. 1993, Back et al. 1993, Galan and Barton 1998] is found most useful in mathematical and numerical analysis. However, because hybrid automata has to explicitly specify all modes and the transitions between them, it is often impractical to describe systems with very large or even unbounded number of modes. Systems whose number of modes cannot be practically predetermined are called *structurally dynamic*.

Special *modeling languages* have also been developed to facilitate modeling and simulation. Languages such as Simulink [21] and Ptolemy

[*]This material is based upon work supported by the National Science Foundation under Grant No. CCF-0306046

[†]Department of Computer Science, Yale University, CT

II [14] employ a *causal* model of computation in which the direction of signal flow is explicit, while languages such as Dymola [11] and Modelica [22] adopt an *non-causal* approach to infer causality from the interconnection of system components.

Non-causal modeling refrains users from committing the model to a specific causality and hence improves the modularity and reusability. But current non-causal modeling languages lack the flexibility at reacting to system change as a response to extern signals. Functional Hybrid Modeling (FHM) [18], a combined paradigm of functional programming and non-causal modeling, has been proposed to allow the description of structurally dynamic models. FHM is closely related to the prior work of *Functional Reactive programming*, or FRP [28], a framework embodied in a language call *Yampa* [17] as an extension of Haskell.

This paper examines the language design issues outlined by *Hydra* and FHM. The rest of this paper is organized as follows. Section 2 reviews the key design issues in FHM. Section 3 presents the language syntax, semantics and types, and section 4 further explores the details of equation systems and reactivity as the core of functional non-causal modeling, and gives an more operational semantics for the language. The end result is by no means a complete solution for FHM, but hopefully it will facilitate further studies of FRP in the field of non-causal modeling.

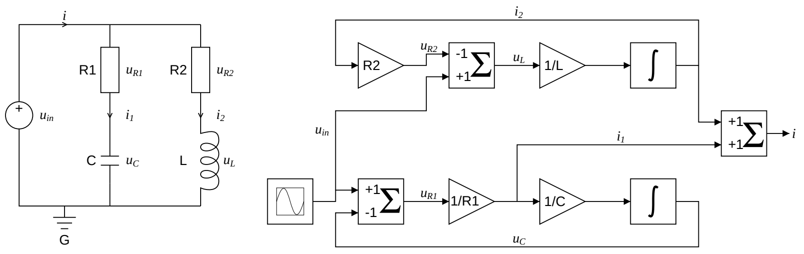
2 Non-Causal Modeling

This is a review of non-causal modeling as *signal relations*, adapted from the early work on FHM [18] at Yale.

2.1 Non-Causal Modeling

A causal language makes the cause-and-effect relationship explicit, or in other words, it is a step-by-step computation from *unknown* quantities to define *known* quantities. For example, a function in the C programming language computes a return value based on given parameters. Consider the simple electrical circuit in Figure 1(a) (adapted from [22]). A corresponding *block diagram* in Simulink is given in Figure 1(b).

It is important to note that the causality flow is fixed in the diagram, i.e., the causal representation exactly defines how the model is to be solved mathematically to arrive at the output from its inputs. This has little structural resemblance to the physical circuit, because it will require a completely different model if the current i is maintained by the source, and u_{in} is the unknown.



(a) Electrical circuit (b) Causal model

Figure 1: A simple electrical circuit and its causal model.

In contrast, non-causal modeling allows structural composition of a system by it components, because the causality is not embedded into the model and only to be derived when a model is explicitly solved. The non-causal model of the above example is given below as a system of equations.

$$
\begin{aligned}
u_{R_2} &= R_2 i_2 \\
u_L &= u_{in} - u_{R_2} \\
i_2{}' &= \frac{u_L}{L} \\
u_{R_1} &= u_{in} - u_C \\
i_1 &= \frac{u_{R_1}}{R_1} \\
u_C{}' &= \frac{i_1}{C} \\
i &= i_1 + i_2
\end{aligned}
$$

Non-causal modeling allows the user to define models for components and then create interconnected instances by structural combination. Modelica [22] supports object-oriented modeling by abstracting common aspects of similar models into a superclass and defining multiple sub-classes through inheritance. A model is then built by inter-connecting various class instances (objects). For example, the model in Figure 1(a) can be described in Modelica in the following way (adapted from [22]):

```
connector Pin
  Voltage    v;
  Current    i;
end Pin;

partial class TwoPin
```

```
  Pin        p, n;
  Voltage    v;
  Current    i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

class Resistor
  extends TwoPin;
  parameter Real R     "Resistance";
equation
  R * i = v;
end Resistor;

class Capacitor
  extends TwoPin;
  parameter Real C;
equation
  C * der(v) = i       "Capacitance";
end Capacitor;

class Inductor
  extends TwoPin;
  parameter Real L     "Inductance";
equation
  v = L * der(i);
end Inductor;

class VsourceAC
  extends TwoPin;
  parameter Voltage VA = 220    "Amplitude";
  parameter Real    f  = 50     "Frequency";
  constant  Real    PI = 3.141592653589793;
equation
  v = VA * sin(2 * PI * f * time);
end VsourceAC;

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

```
model SimpleCircuit
  Resistor   R1(R=10);
  Capacitor  C(C=0.01);
  Resistor   R2(R=100);
  Inductor   L(L=0.1);
  VsourceAC  AC;
  Ground     G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(AC.n, G.p);
  connect(AC.n, G.p);
end SimpleCircuit;
```

where `connect(pin1, pin2)` can be expanded to

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

After a non-causal model is built, we are free to solve its equation systems to get the desired output in any way that is permitted mathematically without changing the model itself. The causality is only analyzed during the equation solving process, which can be automated by the computer system and enables the user to deal with very complex systems conveniently.

2.2 First-Class Signal Relations

A *signal* is a value changing with time, or conceptually, a function of time. Signals are the basic elements in a dynamic system, and they are bounded by the relationship under physical laws. Or abstractly, they must satisfy a set of mathematical equations. We'll view such relationship as *signal relations*.

Conceptually, if we define signals as type $\mathsf{Signal}\ \alpha \approx \mathsf{Time} \rightarrow \alpha$, i.e., a time function with the instance value of type α, we then introduce a type $\mathsf{SR}\ \alpha$ for a relation on a signal of type $\mathsf{Signal}\ \alpha$. For example, the derivative relation, *der*, has the type $\mathit{der}\ :: \mathsf{SR}\ (\mathtt{Real}, \mathtt{Real})$. We can view it as a binary relation on two signals of Real value, i.e., one being the derivative of the other.

Signal relations can be applied to signals in similar ways to first-class functions. We'll use the symbol $\diamond$ to denote it. For example

$$\mathit{der} \diamond (x, y)$$

has the same meaning as the differential equation: $x' = y$, or we'll just use $\mathbf{der}(x)$ as a shorthand for y.

In fact, all equations can be regarded as a form of signal relation application. One can also write $f(x,y) = g(x,y)$ as

$$= \diamond (f(x,y), g(x,y))$$

which can be further simplified by defining a relation $=_{f,g}$ annotated by function f and g, so that $f(x,y) = g(x,y)$ becomes the same as

$$=_{f,g} \diamond (x,y)$$

It is easily seen that every equation can be transformed into a relation application form of $r \diamond \mathbf{x}$, where $\mathbf{x}$ is a vector (or tuple) of signal variables.

Using the signal relation syntax, the example given in Fig. 1(a) can be written as follows (adapted from [18])

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

resistor :: Resistance → SR (Pin, Pin)
resistor(r) = sigrel (p, n) where
                twoPin ◇ (p, n, v)
                r · p.i = v

inductor :: Inductance → SR (Pin, Pin)
inductor(l) = sigrel (p, n) where
                twoPin ◇ (p, n, v)
                l · der(p.i) = v

capacitor :: Capacitance → SR (Pin, Pin)
capacitor(c) = sigrel (p, n) where
                 twoPin ◇ (p, n, v)
                 c · der(v) = p.i

vSourceAC :: Voltage → SR (Pin, Pin)
vSourceAC(u) = sigrel (p, n) where
                  twoPin ◇ (p, n, v)
                  v = u · sin(2 · PI · 50 · time)

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

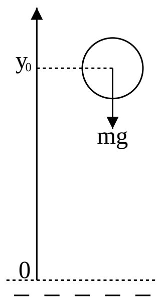


Figure 2: Ball over floor

```
simpleCircuit :: SR Current
simpleCircuit = sigrel (i) where
                    resistor(1000) ◇ (r1p, r1n)
                    resistor(2200) ◇ (r2p, r2n)
                    capacitor(0.00047) ◇ (cp, cn)
                    inductor(0.01) ◇ (lp, ln)
                    vSourceAC(220) ◇ (acp, acn)
                    ground ◇ gp
                    connect acp, r1p, r2p
                    connect r1n, cp
                    connect r2n, lp
                    connect acn, cn, ln, gp
                    i = r1p.i + r2p.i
```

where the **connect**-notation is a similar abbreviation as in Modelica.

Compared to the Modelica program, functional abstraction in the above example is both simpler and more expressive. First-class signal relations are analogous to first-class functions in the way that they can be used to define new relations.

2.3 Modeling Reactivity

A hybrid system not only exhibits the continuous behavior of system variables, but must also react to event changes, which is often introduced as a certain kinds of discontinuity, either in variables or the system structure.

Take the example of bouncing ball [16] as illustrated in Figure 2. Suppose a ball with negligible radius drops from a place of height `y0`, it will bounce off the ground as the consequence of a fully elastic collision. The event of the ball hitting the ground triggers an immediate change of sign in its velocity.

A simple free falling ball can be described as follows in *Hydra*

```
$freefall$ :: SR (Real, Real)
$freefall$ = **sigrel** $(v, l)$ **where**
            $der(l) = v$
            $der(v) = -9.8$
```

The ball will hit ground when l goes from positive to 0, an event that can be described by an (in)equality test. *Hydra* models this event handling as follows

```
$ball$ :: (Real, Real) → SR (Real, Real)
$ball(v_0, l_0)$ = **sigrel** $(v, l)$ **where**
               **init** $v, l$ **by**
                    $ini(v) = v_0$
                    $ini(l) = l_0$
               **in**
                    $freefall \diamond (v, l)$
               **once**
                    $cur(l) = 0$
               **become**
                    $ball(-cur(v), cur(l)) \diamond (v, l)$
```

where *ini* and *cur* represent respectively the initial and current values of a signal and one can view them as similar signal relations just like *der*.

The language construct **init-by-in** introduce clauses for variable initialization purposes, while **once-become** would switch signal relation from one into the other once the event test becomes *true*. Because the bouncing event is recurrent, it switches into the same signal relation except with different initialization values.

Having seen the examples of applying functional concepts in modeling non-causal systems, we still need extra details on how to run such programs, as a useful program not only has to describe the model but also to give the intended output as results.

3 Syntax, Types and Modes

The syntax of *Hydra* is given in Figure 3. Most notably, *Hydra* only allows function definitions over signal relations, but not over the usual value domains. The use of mathmatical functions such as *sin*, *cos*, etc. must all be pre-defined. There is also no support for nested definition of functions or signal relations.

For simplicity, we'll use x for variables, n for numbers, e for expressions, c for clauses, and f for functions that yield values, and g for functions that yield signal relations.

The different kinds of clauses are explained as follows

$$\begin{array}{rcl}
\mathit{Var} & := & x|y|\dots \\
\mathit{Vars} & := & \mathit{var}_1,\dots,\mathit{var}_n \\
\mathit{Exp} & := & n \mid \mathit{var} \mid \mathit{var}(\mathit{exp}) \mid (\mathit{exp}_1,\dots,\mathit{exp}_n) \\
\mathit{Clause} & := & \mathit{exp}_1 = \mathit{exp}_2 \mid \mathit{var} \diamond \mathit{exp} \mid \mathit{clause}_1 \;;\; \mathit{clause}_2 \\
 & & \mid \mathit{clause}_1 \;\textbf{once}\; \mathit{exp} \;\textbf{become}\; \mathit{clause}_2 \\
 & & \mid \textbf{local}\; \mathit{vars} \;\textbf{in}\; \mathit{clause} \;\mid \textbf{init}\; \mathit{vars} \;\textbf{by}\; \mathit{clause}_1 \;\textbf{in}\; \mathit{clause}_2 \\
\mathit{Sigrel} & := & \textbf{sigrel}\; (\mathit{vars}) \;\textbf{where}\; \mathit{clause} \\
\mathit{Decl} & := & \mathit{var} = \mathit{sigrel} \mid \mathit{var}\; (\mathit{vars}) = \mathit{sigrel} \\
\mathit{Program} & := & \mathit{sigrel} \mid \textbf{let}\; \mathit{decl} \;\textbf{in}\; \mathit{program}
\end{array}$$

Figure 3: *Hydra* Syntax

$e_1 = e_2$ defines an equational constraint that the values of all variables.

$g \diamond e$ applies a signal relation function g to an expression e, which can be a list of signal variables.

$c_1; c_2$ behaves as c_1 followed by c_2. This is like the concatenation of two clauses.

c_1 **once** e **become** c_2 is a switch that behaves as c_1 initially when e evalutes to *false*, and as c_2 (and keeps it that way) once e evaluates to *true*.

local $x_1, \dots, x_n$ **in** c introduces new variables to be used locally in c.

init $x_1, \dots, x_n$ **by** c_1 **in** c_2 initializes (or re-initializes) variables by evaluating c_1 before continuing with c_2.

This language supports relation declaration with arguments, but not nested declaration or high order ones. Neither does it support ordinary function declarations, which means only pre-defined functions can be used.

The eventual program is just a signal relation, which is defined by using the **sigrel-where** syntax. A signal relation can be applied to a list of variables to define other signal relations, which is like a function except only applicable to signal variables. The **let-in** syntax makes it possible have recursively defined signal relations.

3.1 Typing Rule

The typing rules for *Hydra* are given in Figure 4, in which α, μ, ν are type variables. For the moment, we'll use a blackbox type C to represent the type of clauses, the reason for which will be explained in later chapters.

$$\frac{}{\Gamma \vdash \textbf{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \mathsf{Bool}}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Real}} \qquad \frac{}{\Gamma \vdash x : \alpha}\; x : \alpha \in \Gamma$$

$$\frac{\Gamma \vdash e_1 : \alpha_1 \quad \dots \quad \Gamma \vdash e_n : \alpha_n}{\Gamma \vdash (e_1, \dots, e_n) : \alpha_1 \times \dots \times \alpha_n}$$

$$\frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash f : \alpha \to \mu}{\Gamma \vdash f(e) : \mu}$$

$$\frac{\Gamma \vdash e_1 : \alpha \qquad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash e_1 = e_2 : \mathsf{C}} \qquad \frac{\Gamma \vdash c_1 : \mathsf{C} \qquad \Gamma \vdash c_2 : \mathsf{C}}{\Gamma \vdash c_1 \,;\, c_2 : \mathsf{C}}$$

$$\frac{\Gamma, x_1 : \alpha_1, \dots, x_n : \alpha_n \vdash c : \mathsf{C}}{\Gamma \vdash \textbf{local}\; x_1, \dots, x_n \;\textbf{in}\; c : \mathsf{C}}$$

$$\frac{\Gamma \vdash c_1 : \mathsf{C} \qquad \Gamma \vdash c_2 : \mathsf{C}}{\Gamma \vdash \textbf{init}\; x_1, \dots, x_n \;\textbf{by}\; c_1 \;\textbf{in}\; c_2 : \mathsf{C}}\; x_1 : \alpha_1, \dots,\; x_n : \alpha_n \in \Gamma$$

$$\frac{\Gamma \vdash c_1 : \mathsf{C} \qquad \Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash c_2 : \mathsf{C}}{\Gamma \vdash c_1 \;\textbf{once}\; e \;\textbf{become}\; c_2 : \mathsf{C}}$$

$$\frac{\Gamma, x_1 : \alpha_1, \dots, x_n : \alpha_n \vdash c : \mathsf{C}}{\Gamma \vdash \textbf{sigrel}\; x_1, \dots, x_n \;\textbf{in}\; c : \alpha_1 \times \dots \times \alpha_n \to \mathsf{C}}$$

$$\frac{\Gamma \vdash g : \alpha \to \mathsf{C} \qquad \Gamma \vdash e : \alpha}{\Gamma \vdash g \diamond e : \mathsf{C}}$$

$$\frac{\Gamma, g : \mu \vdash u : \mu \qquad \Gamma, g : \mu \vdash v : \nu}{\Gamma \vdash \textbf{let}\; g = u \;\textbf{in}\; v : \nu}$$

$$\frac{\Gamma, g : \alpha_1 \times \dots \times \alpha_n \to \mu, x_1 : \alpha_1, \dots, x_n : \alpha_n \vdash u : \mu \qquad \Gamma, g : \alpha_1 \times \dots \times \alpha_n \to \mu \vdash v : \nu}{\Gamma \vdash \textbf{let}\; g(x_1, \dots, x_n) = u \;\textbf{in}\; v : \nu}$$

Figure 4: *Hydra* Typing Rule

The only place where new variables are introduced are the **local** and **sigrel** constructs, and because *Hydra* only allows top level **sigrel** definitions, it is safe to assume there is no nested relations.

3.2 Modes

Aside from types, *Hydra* uses modes to enforce certain properties of a given program, which is related to causality analysis. It is one thing to assert that there exists a solution for a set of given equations, and quite another to turn the specification into a compuation that eventually yield the correct solution. Functions have a fixed causality as the inputs and outputs are already specified, which is certainly not the case for relations. A causality analysis will certainly help us to understand more about relations.

For example, equation $x + y = 1$ can be either regarded as a computation $x = 1 - y$ or $y = 1 - x$, where x is computed given the value of y in the former and y is computed given the value of x in the latter.

A slightly more sophisticated example is a 2-equation set

$$\begin{aligned} x + y &= 1 \\ x - y &= 1 \end{aligned}$$

which can be either translate into

$$\begin{aligned} x &= 1 - y \\ y &= 1 + x \end{aligned}$$

or

$$\begin{aligned} y &= 1 - x \\ x &= 1 + y \end{aligned}$$

where the mutual dependancy of variables x and y can be easily dealt with by saying that the solution (x, y) is the fixed point of function $f_1(x, y) = (1 - y, 1 + x)$ or $f_2(x, y) = (1 + y, 1 - x)$. Note that the least fixed point of either f_1 or f_2 is bottom, but nevertheless the real solution (1, 0) is a non-bottom fixed point of both. The quest of finding such a non-bottom fixed point is discussed in more detail in later chapters.

It becomes apparent that for each variable it needs to be only computed once, which corresponds to the necessary condition for solving m equations with n unknowns, i.e., $m = n$. This is often called *Single Assignment* property [cite as in Modelica]. By assigning a mode, either *input* represented by $\circ$ or *output* represented by $\bullet$, to every occurrence of all variables, we'll be able to verify this property for valid programs and rule out invalid ones for which no sound solution exists. Mode inference can also be viewed as causality analysis.

Formally, we'll define

$$\begin{array}{ll} \mathit{Mode} & = \circ | \bullet \\ \theta : \mathit{ModeMapSet} & = \{\mathit{Var} \rightarrow \mathit{Mode}\} \\ \delta : \mathit{RelSig} & = ((\mathit{Var}_1, \ldots, \mathit{Var}_n), \mathit{ModeMapSet}) \end{array}$$

The condition for a clause c to have a mode map set θ is that $\forall m \in \theta, dom(m) = FV(c)$. For convenience, we'll just define $dom(\theta) = FV(c)$.

The environment Δ maps (a relation) variable to its signature *RelSig*, which is represented by a list of formal variables and a *ModeMapSet*. A relation signature $((x_1, \ldots, x_n), \theta)$ is valid iff θ isn't empty and $dom(\theta) = \{x_1, \ldots, x_n\}$. The signature maintains the list of formal variables as a tuple instead of a set because the order is important for relation applications.

Mode checking rules are given in Figure 5.

$$\mathit{SEQ} \quad \frac{\Delta \vdash_c c_1 : \theta_1 \qquad \Delta \vdash_c c_2 : \theta_2}{\Delta \vdash_c c_1; c_2 : \theta_1 \oplus \theta_2}$$

$$\mathit{RAPP} \quad \frac{\Delta \vdash_r g : ((x_1, \ldots, x_n), \theta)}{\Delta \vdash_c g \diamond (y_1, \ldots, y_n) : \theta[y_1/x_1, \ldots, y_n/x_n]}$$

$$\mathit{SIGREL} \quad \frac{\Delta \vdash_c c : \theta}{\Delta \vdash_r \textbf{sigrel } (x_1, \ldots, x_n) \textbf{ where } c : ((x_1, \ldots, x_n), \theta)}$$

$$\mathit{LET} \quad \frac{\Delta, g : \delta \vdash_r u : \delta \qquad \Delta, g : \delta \vdash_r v : \delta'}{\Delta \vdash_r \textbf{let } g = u \textbf{ in } v : \delta'}$$

$$\mathit{LOCAL} \quad \frac{\Delta \vdash_c c : \theta}{\Delta \vdash_c \textbf{local } x_1, \ldots, x_n \textbf{ in } c : \theta \ominus \{x_1, \ldots, x_n\}}$$

$$\mathit{ONCE} \quad \frac{\Delta \vdash_c c_1 : \theta \qquad \Delta \vdash_c c_2 : \theta}{\Delta \vdash_c c_1 \textbf{ once } e \textbf{ become } c_2 : \theta}$$

Figure 5: *Hydra* Mode Checking Rule

where the operator $\oplus$, $\ominus$ and substitution $\_[\_/\_]$ are defined below

$$\theta_1 \oplus \theta_2 = \{m_1 \uplus m_2 | m_1 \in \theta_1, m_2 \in \theta_2, \forall x \in dom(m_1), m_1(x) = \circ \vee m_2(x) = \circ)\}$$

where $dom(m_1 \uplus m_2) = dom(m_1) \cup dom(m_2)$

and $\quad (m_1 \uplus m_2)(x) = \begin{cases} \bullet, \text{iff } m_1(x) = \bullet \vee m_2(x) = \bullet \\ \circ, \text{otherwise} \end{cases}$

$$\theta \ominus s \quad = \{m' | m \in \theta, dom(m') = dom(m) - s, m'(x) = m(x)\}$$

$$\theta[y_1/x_1, \ldots, y_n/x_n] = \{m' | m \in \theta, m'(y) = \text{if } y = y_i \text{ then } m(x_i) \text{ else } \bot\}$$

4 From Signal Relation to Function Program

With the type information of a *Hydra* program, we are then ready to transform the signal relation into a functional program. An simple example is given below

$$\begin{array}{l} \textbf{sigrel}\ (x, y)\ \textbf{where} \\ \qquad x + y = 5 \\ \qquad x - y = 3 \end{array}$$

After transforming equation into relation, it becomes

$$\begin{array}{l} \mathrm{r} = \textbf{sigrel}\ (x, y)\ \textbf{where} \\ \qquad\qquad =_{+(\_,\_,5)} \diamond (x, y) \\ \qquad\qquad =_{-(\_,\_,3)} \diamond (x, y) \end{array}$$

Then we can have a predefined type context

$$\Gamma = \left\{ \begin{array}{ll} =_{+(\_,\_,5)}: & (\mathsf{Real}^{\perp}, \mathsf{Real}^{\top}) \to \mathsf{C}, \\ =_{+(\_,\_,5)}: & (\mathsf{Real}^{\top}, \mathsf{Real}^{\perp}) \to \mathsf{C}, \\ =_{-(\_,\_,3)}: & (\mathsf{Real}^{\perp}, \mathsf{Real}^{\top}) \to \mathsf{C}, \\ =_{-(\_,\_,3)}: & (\mathsf{Real}^{\top}, \mathsf{Real}^{\perp}) \to \mathsf{C} \end{array} \right\}$$

So that after type inference, the program can be type annotated as

$$\begin{array}{l} \mathrm{r} = \textbf{sigrel}\ (x, y)\ \textbf{where} \\ \qquad\qquad (=_{+(\_,\_,5)}: (\mathsf{Real}^{\top}, \mathsf{Real}^{\perp}) \to \mathsf{C}) \diamond (x : \mathsf{Real}^{\top}, y : \mathsf{Real}^{\perp}) \\ \qquad\qquad (=_{-(\_,\_,3)}: (\mathsf{Real}^{\perp}, \mathsf{Real}^{\top}) \to \mathsf{C}) \diamond (x : \mathsf{Real}^{\perp}, y : \mathsf{Real}^{\top}) \end{array}$$

which is then transformed into a function (in Haskell syntax)

$$\begin{array}{l} \mathrm{r}\ (x, y) = \textbf{let}\ x = 5 - y \\ \qquad\qquad\quad\ \ y = x - 3 \\ \qquad\qquad \textbf{in}\ (x, y) \end{array}$$

because relations like $=_{+(\_,\_,5)}$ and $=_{-(\_,\_,3)}$ when annotated with modes become just like mathematical functions that computes from unknown to known.

Note that x and y are defined above in a mutually recursive way, so the solution has to be a fixed point of r. Although its least fixed point is bottom, we can compute a non-least fixed point (if it exists) by using a different fixed point operator.

It becomes apparent to this point, at least intuitively, that a *Hydra* program, or a signal relation, can be transformed into a function, a fixed point of which is the meaning of the program.

4.1 ODE, DAE and Signals

Previous examples show that signal relations use equations to represent mathematical models. As signals are functions of time, we are primarily interested in differential equations, but algebraic equations are also part of the general concept of defining signal relations.

Differential equations involve both a vector of unknowns x and its derivative x'. The simplest *ordinary differential equation* system, or ODE, is

$$x' + Ax = 0 \tag{1}$$

where $x, x' \in \mathbb{R}^n, A \in \mathbb{R}^{n \cdot n}$. If the right hand side is a function of t instead of a zero vector, it is would be called an *inhomogeneous* ODE. Consider a system of

$$Ax' + Bx = C(t) \tag{2}$$

If A is invertible, multiplying both sides by A^{-1} will give us an ODE. If A is not invertible, it is then a system of *differential algebraic equations*, or DAEs. The most general form of DAE is

$$f(x', x, t) = 0 \tag{3}$$

To solve a linear ODE numerically is to initialize the vector x with some initial value x_0 at time t_0, and then the value of x' can be immediately calculated, and then a numerical integration technique such as Euler's method can be applied to obtain the sequence of instance values of x by time, and to any precisions as deemed necessary.

The usual mathematical integration works on a pair (x_0, d), where x_0 is the initial value and d is the derivative function that describes the change of v with respect to time. For instance, if we use Euler's method on a homogeneous ODE, the next instance value of $x(t_{n+1})$ can be obtained by

$$\begin{aligned} x(t_{n+1}) \approx &\quad x_{n+1} = x_n + f(x_n) \cdot dt \\ \text{where} &\quad t_n = t_0 + n \cdot dt \end{aligned}$$

Hence function d can be defined as

$$\begin{aligned} d &:: \mathsf{Time} \to \mathsf{V} \to \mathsf{V} \\ d\ dt\ x &= x + f(x) \cdot dt \end{aligned}$$

In other words, d can be viewed as a function that returns the variable value at the next time interval. So the solution to such an ODE becomes

$$\begin{aligned} x(t) = & \lim_{n\to\infty} \underbrace{d'(\ldots(d'}_{n \text{ times}}\ x_0)\ldots) \\ \text{where} \quad & d' = d\ (\frac{t}{n}) \end{aligned}$$

Reactivity introduces discontinuity in function x, which means d' may change anywhere during its n times application. Therefore, we have to define a different type for d

$$d :: \mathsf{D} = \mathsf{Time} \to \mathsf{V} \to (\mathsf{V}, \mathsf{D})$$

which means the evaluation of the value at the next time interval must not only produce the value itself, but also the next D function. Or in other words, the entire system, not just the variable value, changes along time. The integration then becomes

$$\begin{aligned} x(t) = & \mathit{fst}(\lim_{n\to\infty} \underbrace{d'(\ldots(d'}_{n \text{ times}}(d\ (\frac{t}{n})\ x_0))\ldots)) \\ \text{where} \quad & d'(x, d) = d\ (\frac{t}{n})\ x \\ & \mathit{fst} \text{ projects the first element from a tuple.} \end{aligned}$$

The recursive type of d captures the essense of reactivity in *Hydra*, which sees the *future* as a continuation of the *present*.

One may argue that the signal of Reals can be just represented by its value and derivative (which in turn is another signal) as in Signal Real = (Real, Signal Real), but this definition falls short when we introduce reactivity, because the next value of x can no longer be computed as in $d\ dt\ x = x + f(x) \cdot dt$.

4.2 Semantics

To simplify the semantics definition, we'll define signals in a slightly different way

$$\mathsf{Signal}\ V \quad = \quad (\mathsf{V}, \mathsf{Time} \to \mathsf{Signal}\ V)$$

Based on Euler's method, the function that computes the next value of x given its numerical derivative is

$$\mathit{next}\ x\ dx\ dt \quad = \quad x + dx \cdot dt$$

As seen earlier, a type-checked and mode-annotated signal relation can be translated into a functional program, which treats signal variables just as ordinary algebraic variables, and applies ordinary arithmetic functions such as $+$ or $*$ to them. It is indeed straight forward to lift functions over algebraic value domain to over the signal domain using the signal representation defined above.

```
mkTuple :: Signal α → Signal β → Signal (α × β)
mkTuple (x, dx) (y, dy) = ((x, y), λdt.(dx dt, dy dt))

fstS :: Signal (α × β) → Signal α
fstS (z, dz) = (fst z, fstS ∘ dz)

integrate :: α → Signal α → Signal α
integrate x (x', dx') = (x, λdt.integrate(next x x' dt)(dx' dt))

liftS :: (α → β) → Signal α → Signal β
liftS f (x, dx) = (f x, λdt.f(dx dt))

switch :: (α → Bool) → (Time → Signal α) → (Signal α → Signal α)
            → Time → Signal α
switch test d_1 c_2 dt = let (x_1, dx_1)  = d_1 dt
                         in if  test x_1  then c_2 x_1
                                          else (x_1, switch test dx_1 c_2)
```

Then the program for the Bouncing Ball example becomes

```
freefall :: Signal (Real, Real) → Signal (Real, Real)
freefall x@(x^-, _) = integrate x^- (mkTuple g (fstS x))
                  where g      = integrate -9.8 zero
                        zero   = (0, λdt.zero)

ball :: (Real, Real) → Signal (Real, Real) → Signal (Real, Real)
ball (v_0, l_0) x = let c_1 x@(_, dx)          = freefall ((v_0, l_0), dx)
                        c_2 x@((v^-, l^-), _)  = ball (−v^-, l^-) x
                        (x_1^-, dx_1)          = c_1 x
                    in (x_1^-, switch test dx_1 c_2)
                   where
                        test (v^-, l^-) = l^-  == 0
```

And the solution to the Bouncing Ball relation is just the fixed point of function *ball*.

5 Conclusion and Future Work

This paper presents *Hydra*, a non-causal language designed for functional hybrid modeling based on the concept of first-class signal relations. As a language embedded in Haskell, *Hydra* enjoys both the modularity of non-causal modeling and the freedom at expressing dynamic hybrid models in a functional reactive programming style. *Hydra* uses mathematical equations to specify signal relations, and an event switch to capture the reactivity at its heart.

Because FHM is largely an unexplored research area, there are still many open ended questions. Existing language systems such as Haskell lack the functionality to fully support the implementation and integration of the non-causal language.

Being a specification language, *Hydra* introduces variables in its equation system without having to classify them as inputs or outputs, because the computation causality is only determined by the solver instead of by the model. Signal relations exhibit different natures when it is used for modeling systems and when it is used for the purpose of simulation.

The discontinuity introduced by event switch requires re-initialization of variable values, which is currently handled by explicit **init-by-in** clauses. This is however, not entirely satisfactory. As the discontinuity of one set of variable values may implicitly result in the discontinuity of another set that is outside the local scope where the event occurs. We may need extra language facilities to help in this regard.

Though we have much confidence, the soundness of Hydra type system is yet to be proved, and the correctness of the steps during *Hydra* program translation and compilation process needs formal investigation too. Our current type system doesn't fully explore the type of clauses and currently just leave it as a blackbox type C. The study if its internals is left for future work.

The mode analysis on *Hydra* programs has a close relationship to the mode systems studied in logic programming languages, and ours is much simpler because it doesn't deal with complex data structures.

As we have mentioned, the final solution is a fixed point of the resulting Haskell program, which requires a non-trivial fixed point solver to get a non-bottom result whenever there exists one. The usual $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$ wouldn't work for even trivial cases like $f\ x = 1 - x$. Interval arithmetic is one possible approach, while a conventional symbolic approach based on Gaussian Elimination could be more efficient. Other alternatives are yet to be explored.

The numeric solver used for *Hydra* does not meet the demand of simulations for real systems. Issues such as stability, discontinuity, and sensitivity have been subjects left aside. Linear DAEs are also insufficient to model systems of moderate complexity. The often cited pendulum example [24, 22, 18] requires a DAE of higher index. The performance of the techniques used for solving linear systems and integrating DAEs are not suitable for system of large scale. We may also need a solution for multi-step integration, because fixed time step tends to accumulate numerical errors and makes the simulation less useful for most practical purposes.

References

[1] Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In [Grossman et all. 1993]. 209-229.

[2] Avraam, M. P., Shah, H., and Pantelides, C. C. 1998. Modeling and optimization of general hybrid systems in the continuous time domain. *Comput. Chem. Eng. 22,* Suppl. S221-S228.

[3] Barton, P. I., Banga, J. R., and Galan, S. 2000. Optimization of Hybrid discrete/continuous dynamic systems. *Comput. Chem. Eng. 24,* 9-10, 2171-2182.

[4] Branicky, M. S., Borkar, V. S., and Mitter, S. K. 1998. A unified framework for hybrid control: model and optimal control theory. *IEEE T. Automat. Contr. 43*, 1, 31-45.

[5] Bundy, A., Welham, B., 1981. Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence 16*, 189-212.

[6] Cellier, F.E. 1986. Combined continuous/discrete simulation applications, techniques and tools. In *Proc. of the 1986 Winter Simulation Conf.* J. Wilson, J. Henriken, and S. Roberts, Eds. 24-33.

[7] Cuijpers, P.J.L., Reniers M.A., 2003. Hybrid Process Algebra. Journal of Logic and Algebraic Programming, 62(2):191-245. February 2005.

[8] David, R. and Alla, H. 2001. On hybrid Petri nets. *Discrete Event Dyn. S. 11*, 9-40.

[9] Wittenberg, D., 2004. CLP(F) Modeling of Hybrid Systems. Ph.D. thesis, Brandeis University, MA, USA.

[10] Diaz, D., 2002. GNU PROLOG: A Native Prolog Compiler with Constraint Solving over Finite Domains, 1.7 edn.

[11] Elmqvist, H., Cellier, F. E. and Otter, M. 1993. Object-oriented modeling of hybrid systems. In *Proceedings of ESS'93 European Simulation Symposium*, page xxxi-xli, Delft, The Netherlands.

[12] Elmqvist, H., Otter, M., and Cellier, F. E. 1995. Inline integration: A new mixed symbolic/numeric approach. In *Proceedings of ESM'95, European Simulation Multiconference*, page xxiii-xxxiv, Prague, Czech Republic.

[13] Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. 1992. The CLP(R) language and system. ACM Trans. Program. Lang. Syst. 14, 3 (May. 1992), 339-395.

[14] Lee, E. A. 2001. Overview of the ptolemy project. Technical memorandum UCB/ERLM01/11, Electronic Research Laboratory, University of California Berkeley.

[15] Monsterman, P. J. 1997. Hybrid dynamic systems: A hybrid bond graph modeling paradigm and its application in diagnosis. Ph.D. thesis, Vanderbilt University, Tennessee.

[16] Nilsson, H., 2003. Functional automatic differentiation with dirac impulses. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, page 153-164, Uppsala, Sweden.

[17] Nilsson, H., Courtney, A. and Peterson J., 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51-64, Pittsberge, Pennsylvania, USA.

[18] Nilsson, H., Peterson, J. and Hudak, P. 2003. Functional Hybrid Modeling. In *Proceedings of PADL'03: 5th International Workshop on Practical Aspects of Declarative Languages*, 376-390.

[19] Timothy J. Hickey. CLP(F) and constrained ODEs. In *Proceedings of the workshop on Constraints and Modelling*. 1994.

[20] Timothy J. Hickey. Analytic constraint solving and interval arithmetic. In *POPL'00 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338-351, 2000. published as vol. 27 of SIGPLAN notices.

[21] The MathWorks, Inc. *Using Simulink Version 4*, June 2001.

[22] The Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling: Language Specification version 2.0*, July 2002.

[23] Otter, M., Elmqvist, H., and Mattsson, S. E. 1999. Hybrid modeling in Modelica based on synchronous data flow principle. In *Proc. of the 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99*. IEEE Control Systems Society, 151-157.

[24] Pantelides, C. C. 1988. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213-231.

[25] Sterling, L., Shapiro, E., 1999. The Art of Prolog: advanced programming techniques, 2 edn. *MIT Press series in logic programming*. The MIT Press.

[26] Turner, D., 1982. Recursion equations as a programming language. In *Functional Programming and Its Application*, Cambridge University Press.

[27] Wadler, P., 1987. List comprehensions. In *The Implementation of Functional Languages*, Prentice Hall.

[28] Wan, Z. and Hudak, P. 2000. Functional reactive programming from first principles. In *Proceedings of PLDI'01: Symposium on Programming Language Design and Implementation*, 242-252.