Embedding a Functional Hybrid Modelling Language in Haskell

George Giorgidze   Henrik Nilsson

Functional Programming Laboratory
School of Computer Science
University of Nottingham

25th British Colloquium on Theoretical Computer Science
University of Warwick, United Kingdom
April 7, 2009

- Functional Hybrid Modelling (**FHM**) is a new approach to the design of **modelling and simulation** (M&S) languages
- **Haskell** is a purely **functional programming language**

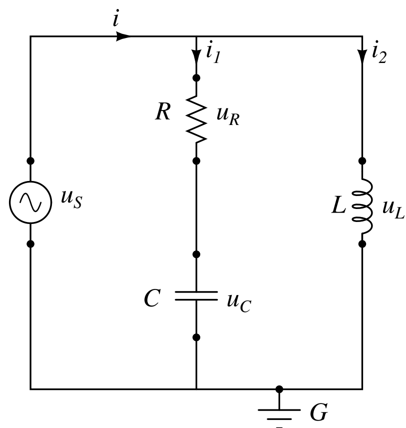
Outline

Modelling and Simulation Languages

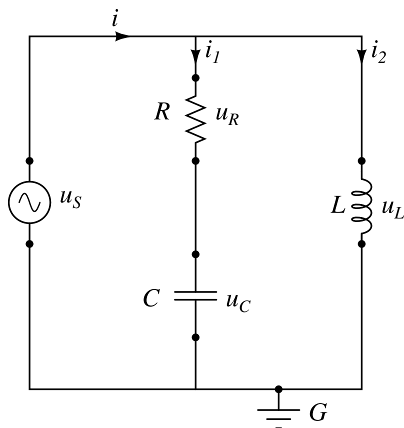
FHM as a Domain Specific Language Embedded in Haskell

Future Work and Conclusions

Simple Electrical Circuit



Simple Electrical Circuit



Differential Algebraic Equations
(**DAE**s)

$$f(\frac{d\vec{x}}{dt}, \vec{x}, \vec{y}, t) = 0$$

$$\begin{aligned}
u_S &= sin(2\pi t) \\
u_R &= R \cdot i_1 \\
i_1 &= C \cdot \frac{du_C}{dt} \\
u_L &= L \cdot \frac{di_2}{dt} \\
i_1 + i_2 &= i \\
u_R + u_C &= u_s \\
u_S &= u_L
\end{aligned}$$

M&S with General Purpose Programming Languages

Ordinary Differential Equation (**ODE**) in **explicit** form

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t)$$

$$
\begin{aligned}
u_S &= sin(2\pi t) \\
u_L &= u_S \\
\frac{di_2}{dt} &= \frac{u_L}{L} \\
u_R &= u_S - u_C \\
i_1 &= \frac{u_R}{R} \\
\frac{du_C}{dt} &= \frac{i1}{C} \\
i &= i_1 + i_2
\end{aligned}
$$

M&S with General Purpose Programming Languages

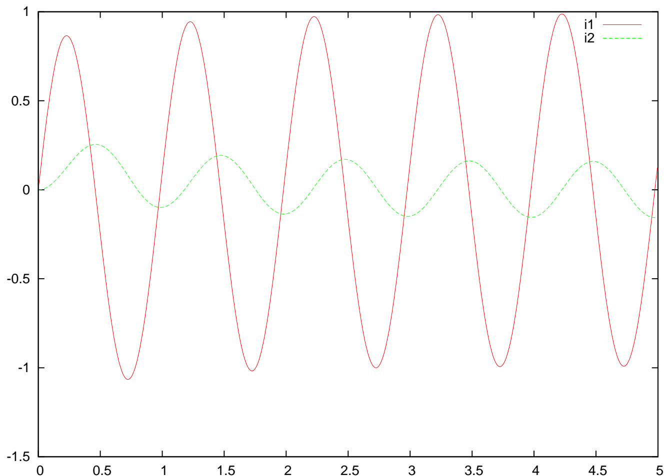
Ordinary Differential Equation (**ODE**) in **explicit** form

$$\frac{d\vec{x}}{dt} = f(\vec{x}, t)$$

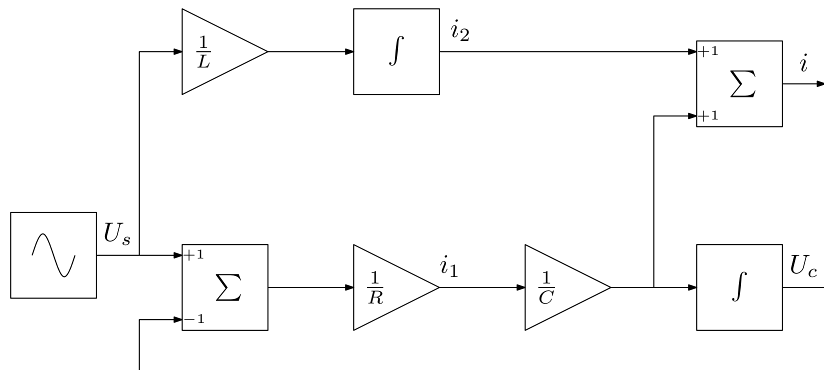
$$\begin{aligned}
u_S &= sin(2\pi t) \\
u_L &= u_S \\
\frac{di_2}{dt} &= \frac{u_L}{L} \\
u_R &= u_S - u_C \\
i_1 &= \frac{u_R}{R} \\
\frac{du_C}{dt} &= \frac{i1}{C} \\
i &= i_1 + i_2
\end{aligned}$$

```
integrateSimpleCircuit dt r c l = go 0 0 0
  where go t uc i2 = hd : tl
          where us  = sin (2 * pi * t)
                ul  = us
                di2 = (ul / l) * dt
                ur  = us - uc
                i1  = ur / r
                duc = (i1 / c) * dt
                i   = i1 + i2
                hd  = (t, i, i1, i2, us, ur, uc, ul)
                tl  = go (t + dt) (uc + duc) (i2 + di2)
```

Simulation Result



Causal Modelling Languages (e.g. Simulink)



Causal Modelling Languages

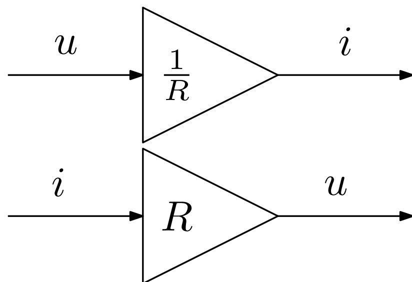
- Very complex/complicated
- Not Modular
- Not Reusable

Causal Modelling Languages

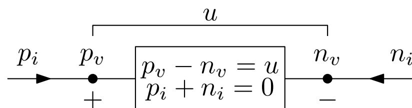
- Very complex/complicated
- Not Modular
- Not Reusable

$$i = \frac{u}{R}$$

$$u = R \cdot i$$



Non-causal modelling using FHM Approach



$$
\begin{aligned}
twoPin &= \quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v), u)\ \textbf{where} \\
&\qquad p_v - n_v = u \\
&\qquad p_i + n_i = 0 \\
resistor &= \lambda r \rightarrow \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad twoPin \diamond ((p_i, p_v), (n_i, n_v), u) \\
&\qquad r * p_i = u \\
resistor &= \lambda r \rightarrow \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad p_v - n_v = u \\
&\qquad p_i + n_i = 0 \\
&\qquad r * p_i = u
\end{aligned}
$$

Non-causal modelling using FHM Approach

$$
\begin{aligned}
\mathit{resistor} &= \lambda r \rightarrow \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
&\qquad r * p_i = u \\
\mathit{inductor} &= \lambda l \rightarrow \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
&\qquad l * \mathit{der}\ p_i = u \\
\mathit{capacitor} &= \lambda c \rightarrow \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
&\qquad c * \mathit{der}\ u = p_i
\end{aligned}
$$

Non-causal modelling using FHM Approach

$$
\begin{aligned}
simpleCircuit = \textbf{sigrel}\ (\textbf{flow}\ i, u)\ \textbf{where}\quad & \\
& vSourceAC\ 1\ 1 \diamond ((acp_i, acp_v), (acn_i, acn_v)) \\
& resistor\ 1 \diamond ((rp_i, rp_v), (rn_i, rn_v)) \\
& inductor\ 1 \diamond ((lp_i, lp_v), (ln_i, ln_v)) \\
& capacitor\ 1 \diamond ((cp_i, cp_v), (cn_i, cn_v)) \\
& ground \diamond (gp_i, gp_v) \\
& \textbf{connect}\ acp_i\ rp_i\ lp_i \\
& \textbf{connect}\ acp_v\ rp_v\ lp_v \\
& \textbf{connect}\ rn_i\ cp_i \\
& \textbf{connect}\ rn_v\ cp_v \\
& \textbf{connect}\ acn_i\ cn_i\ ln_i\ gp_i \\
& \textbf{connect}\ acn_v\ cn_v\ ln_v\ gp_v \\
& i = acp_i \\
& u = acp_v - acn_v
\end{aligned}
$$

Non-causal modelling using FHM Approach

$$
\begin{aligned}
simpleCircuit = \mathbf{sigrel}\ (\mathbf{flow}\ i, u)\ \mathbf{where}\quad & \\
& vSourceAC\ 1\ 1 \diamond ((acp_i, acp_v), (acn_i, acn_v)) \\
& resistor\ 1 \diamond ((rp_i, rp_v), (rn_i, rn_v)) \\
& inductor\ 1 \diamond ((lp_i, lp_v), (ln_i, ln_v)) \\
& capacitor\ 1 \diamond ((cp_i, cp_v), (cn_i, cn_v)) \\
& ground \diamond (gp_i, gp_v) \\
& acp_i + rp_i + lp_i = 0 \\
& acp_v = rp_v = lp_v \\
& rn_i + cp_i = 0 \\
& rn_v = cp_v \\
& acn_i + cn_i + ln_i + gp_i = 0 \\
& acn_v = cn_v = ln_v = gp_v \\
& i = acp_i \\
& u = acp_v - acn_v
\end{aligned}
$$

Functional Hybrid Modelling (FHM)

- FHM extends a purely functional language with a few key abstractions (e.g. signal relation) for supporting non-causal modelling [Nilsson et al. PADL2003]
- FHM approach is more **declarative** and **high-level**, i.e. focuses on **what** to model rather then **how** to simulate
  - Modular
  - Reusable

Functional Hybrid Modelling (FHM)

- FHM extends a purely functional language with a few key abstractions (e.g. signal relation) for supporting non-causal modelling [Nilsson et al. PADL2003]
- FHM approach is more **declarative** and **high-level**, i.e. focuses on **what** to model rather then **how** to simulate
  - Modular
  - Reusable
- Unfortunately, there was no publicly available implementation available.
- We investigate the implementation of FHM as a Haskell embedded domain-specific language ((EDSL)) in Haskell.

Domain Specific Embedding using Abstract Syntax Trees (ASTs)

$$
\begin{aligned}
&\textbf{data}\ \mathit{SigRel} = \mathit{SigRel}\ \mathit{Pattern}\ [\mathit{Equation}] \\
&\textbf{data}\ \mathit{Pattern} = \ldots \\
&\textbf{data}\ \mathit{Equation} = \\
&\quad\ \mathit{EquationEq}\ \mathit{Expr}\ \mathit{Expr} \\
&\quad | \mathit{EquationSigRelApp}\ \mathit{SigRel}\ \mathit{Expr} \\
&\textbf{data}\ \mathit{Expr} = \ldots
\end{aligned}
$$

Originally Proposed Syntax

$$
\begin{aligned}
&\mathit{twoPin} = \\
&\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v), u)\ \textbf{where} \\
&\qquad p_v - n_v = u \\
&\qquad p_i + n_i\ \ = 0 \\
&\mathit{resistor}\ r = \\
&\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
&\qquad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
&\qquad r * p_i = u
\end{aligned}
$$

Executable Haskell Code that Uses Quasiquoting

$$
\begin{array}{l}
\textbf{type}\ Pin = (\mathbb{R}, \mathbb{R}) \\
twoPin :: SR\ (Pin, Pin, \mathbb{R}) \\
twoPin = [\textbf{\$fhm}| \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v), u)\ \textbf{where} \\
\qquad p_v - n_v = u \\
\qquad p_i + n_i \ \ = 0 \\
|] \\
resistor :: \mathbb{R} \rightarrow SR\ (Pin, Pin) \\
resistor\ r = [\textbf{\$fhm}| \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
\qquad \$twoPin\$ \diamond ((p_i, p_v), (n_i, n_v), u) \\
\qquad \$r\$ * p_i = u \\
|]
\end{array}
$$

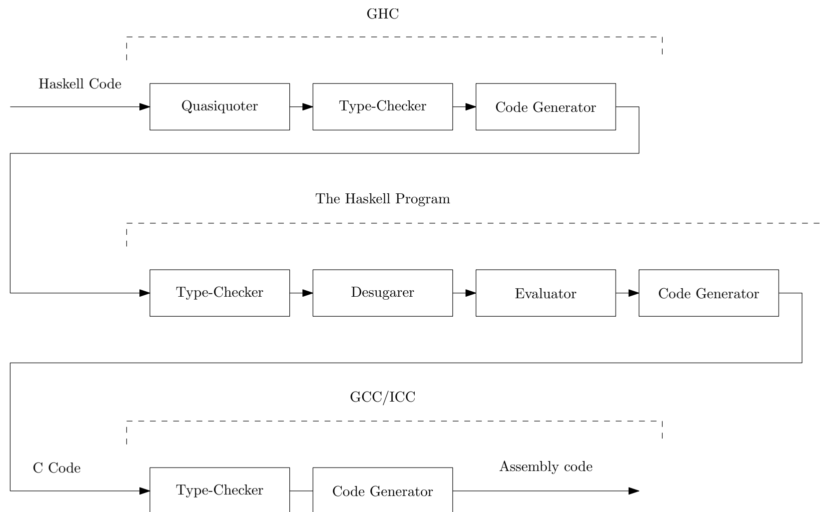
Complete Model in Haskell

$$
\begin{array}{l}
\mathit{simpleCircuit} :: \mathit{SR}\ (\mathbb{R}, \mathbb{R}) \\
\mathit{simpleCircuit} = [\$\mathbf{fhm}| \\
\quad \mathbf{sigrel}\ (\mathbf{flow}\ i, u)\ \mathbf{where} \\
\qquad \$\mathit{vSourceAC}\ 1\ 1\$ \diamond ((\mathit{acp}_i, \mathit{acp}_v), (\mathit{acn}_i, \mathit{acn}_v)) \\
\qquad \$\mathit{resistor}\ 1\$ \diamond ((\mathit{rp}_i, \mathit{rp}_v), (\mathit{rn}_i, \mathit{rn}_v)) \\
\qquad \$\mathit{inductor}\ 1\$ \diamond ((\mathit{lp}_i, \mathit{lp}_v), (\mathit{ln}_i, \mathit{ln}_v)) \\
\qquad \$\mathit{capacitor}\ 1\$ \diamond ((\mathit{cp}_i, \mathit{cp}_v), (\mathit{cn}_i, \mathit{cn}_v)) \\
\qquad \$\mathit{ground}\$ \diamond (\mathit{gp}_i, \mathit{gp}_v) \\
\qquad \mathbf{connect}\ \mathit{acp}_i\ \mathit{rp}_i\ \mathit{lp}_i \\
\qquad \mathbf{connect}\ \mathit{acp}_v\ \mathit{rp}_v\ \mathit{lp}_v \\
\qquad \mathbf{connect}\ \mathit{rn}_i\ \mathit{cp}_i \\
\qquad \mathbf{connect}\ \mathit{rn}_v\ \mathit{cp}_v \\
\qquad \mathbf{connect}\ \mathit{acn}_i\ \mathit{cn}_i\ \mathit{ln}_i\ \mathit{gp}_i \\
\qquad \mathbf{connect}\ \mathit{acn}_v\ \mathit{cn}_v\ \mathit{ln}_v\ \mathit{gp}_v \\
\qquad i = \mathit{acp}_i \\
\qquad u = \mathit{acp}_v - \mathit{acn}_v \\
|]
\end{array}
$$

Meta-modelling and Higher-order Models

$$
\begin{array}{l}
\mathit{serialise} :: [\mathit{SR}\ (\mathit{Pin}, \mathit{Pin})] \rightarrow \mathit{SR}\ (\mathit{Pin}, \mathit{Pin}) \\
\mathit{serialise}\ [\mathit{sr}] = \mathit{sr} \\
\mathit{serialise}\ (\mathit{sr} : \mathit{srs}) = [\mathbf{\$fhm}| \\
\quad \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v))\ \mathbf{where} \\
\qquad \$\mathit{sr}\$ \qquad\qquad\quad\ \diamond ((p_i, p_v), (n1_i, n1_v)) \\
\qquad \$\mathit{serialise}\ \mathit{srs}\$ \diamond ((p2_i, p2_v), (n_i, n_v)) \\
\qquad \mathbf{connect}\ n1_i\ p2_i \\
\qquad \mathbf{connect}\ n1_v\ p2_v \\
|]
\end{array}
$$

Code Flow in the Implementation



Summary of Implementation

- The method of embedding was inspired by [Mainland, ICFP 2008]
- It employs new features introduced in the Glasgow Haskell Compiler (GHC) version 6.10
- In particular, our embedding uses quasiquoting that allowed us to embed the language using the domain specific syntax
- Generates fast simulation code in C

Future Work and Conclusions

-
  - **F**unctional +
  - **H**ybrid -
  - **M**odelling +

Future Work and Conclusions

- - **F**unctional +
  - **H**ybrid -
  - **M**odelling +
- The work is supported by the first publicly available prototype implementation of a FHM language.
- The source code (under the open source BSD license) and paper is available online (`http://www.cs.nott.ac.uk/~ggg/`).

Thank You!