Embedding a Functional Hybrid Modelling Language in Haskell

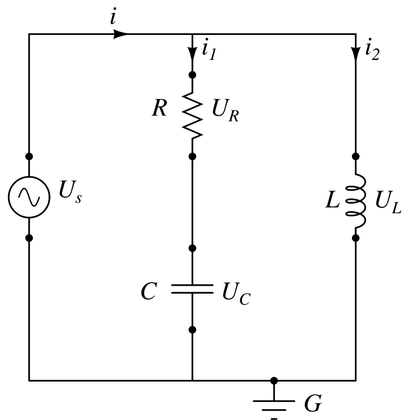
George Giorgidze Henrik Nilsson

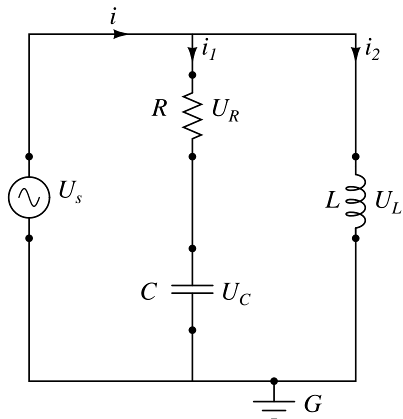
Functional Programming Laboratory
School of Computer Science
University of Nottingham

20th International Symposium on the Implementation and Application of Functional Languages
University of Hertfordshire, United Kingdom
September 12, 2008

Outline

1. Modelling Languages for Physical Systems
   - Causal Modelling Languages
   - Non-Causal Modelling Languages
2. Functional Hybrid Modelling (FHM)
3. The Haskell Embedding
4. Future Work
5. Conclusions


i
i1
i2
R
UR
Us
L
UL
C
UC
G

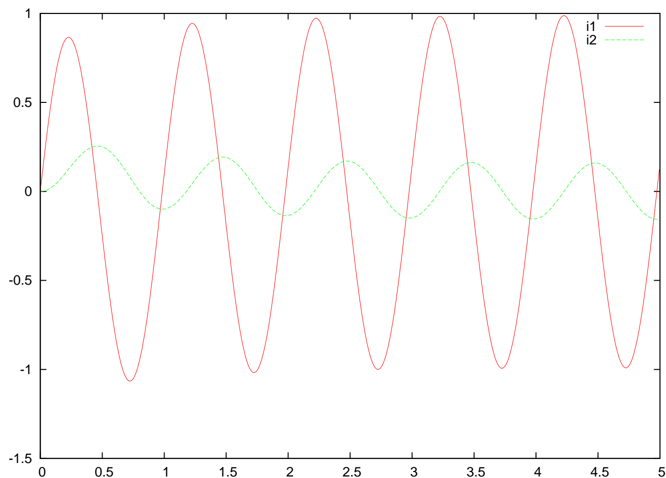


$$
\begin{aligned}
U_S &= sin(2\pi t) \\
U_R &= R i_1 \\
i_1 &= C \frac{dU_C}{dt} \\
U_L &= L \frac{di_2}{dt} \\
i_1 + i_2 &= i \\
U_R + U_C &= U_s \\
U_S &= U_L
\end{aligned}
$$

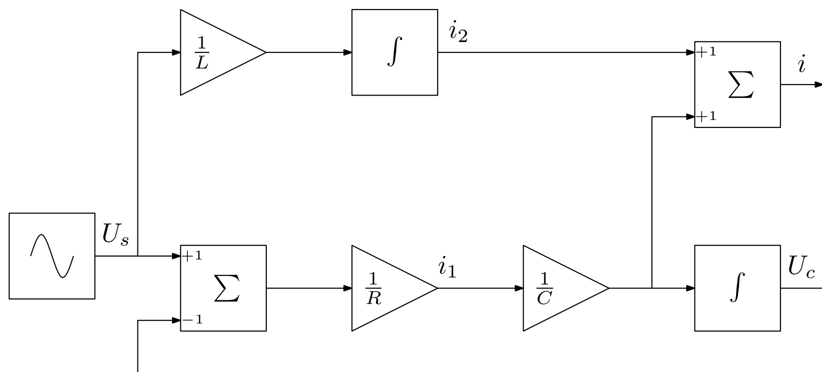
$$
\begin{aligned}
U_S &= sin(2\pi t) \\
U_L &= U_S \\
di_2 &= \frac{U_L}{L} dt \\
U_R &= U_S - U_C \\
i_1 &= \frac{U_R}{R} \\
dU_C &= \frac{i1}{C} dt \\
i &= i_1 + i_2
\end{aligned}
$$

$$
\begin{aligned}
U_S &= sin(2\pi t) \\
U_L &= U_S \\
di_2 &= \frac{U_L}{L} dt \\
U_R &= U_S - U_C \\
i_1 &= \frac{U_R}{R} \\
dU_C &= \frac{i1}{C} dt \\
i &= i_1 + i_2
\end{aligned}
$$

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

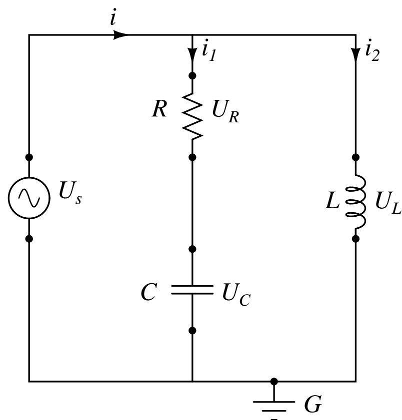

i1
i2
1
0.5
0
-0.5
-1
-1.5
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

Simulink block-diagrams



Modelica

```
connetor Pin
  flow Real i;
  Real v;
end Pin;

model TwoPin
  Pin p, n;
  Real u, i;
equation
  u = p.v − n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

model Resistor
  extends TwoPin;
  parameter Real R = 1;
equation
  R ∗ i = u;
end Resistor;

model Capacitor
  extends TwoPin;
  parameter Real C = 1;
equation
  C ∗ der (u) = i;
end Capacitor;

model Inductor
  extends TwoPin;
  parameter Real L = 1;
equation
  u = L ∗ der (i);
end Inductor;
```

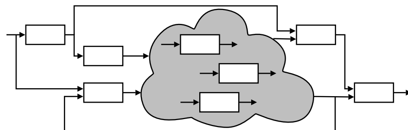


model *SimpleCircuit*
  Resistor R;
  Capacitor C;
  Inductor L;
  VSourceAC AC;
  Ground G;
equation
  connect $(AC.p, R.p)$;
  connect $(AC.p, L.p)$;
  connect $(R.n, C.p)$;
  connect $(AC.n, C.n)$;
  connect $(AC.n, L.n)$;
  connect $(AC.n, G.p)$;
end *SimpleCircuit*;

Functional Reactive Programming (FRP)

- Functional Reactive Programming (FRP) [Elliott, Hudak, Nilsson, Peterson, ...]
- Yampa implements FRP concepts as a domain specific language embedded in Haskell.

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



Functional Hybrid Modelling (FHM)

- FHM [Nilsson, Peterson, Hudak, PADL2003] generalises Yampa's notion of *causal signal function* to *non-causal signal relation*. A signal relation has following conceptual type:

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

  Intuitively, a value of type $SR\ \alpha$ defines the relation on a value of type $Signal\ \alpha$ and on a value of type $Time$. In the FHM setting, *solving* of signal relation $sr :: SR\ \alpha$ involves finding of signal $s :: Signal\ \alpha$ where

  $$\forall\ t :: Time, sr\ s\ t \equiv True$$

Functional Hybrid Modelling (FHM)

- FHM [Nilsson, Peterson, Hudak, PADL2003] generalises Yampa's notion of *causal signal function* to *non-causal signal relation*. A signal relation has following conceptual type:

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

  Intuitively, a value of type $SR\ \alpha$ defines the relation on a value of type $Signal\ \alpha$ and on a value of type $Time$. In the FHM setting, *solving* of signal relation $sr :: SR\ \alpha$ involves finding of signal $s :: Signal\ \alpha$ where

  $$\forall\ t :: Time, sr\ s\ t \equiv True$$

- For example, equality can be seen as a signal relation:

  $$\begin{array}{l} (=) \quad\quad :: SR\ (a, a) \\ (=)\ s\ t \approx fst\ (s\ t) \equiv snd\ (s\ t) \end{array}$$

$$
\begin{aligned}
\mathit{twoPin} = \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v), u)\ \textbf{where} \\
u = p_v - n_v \\
p_i + n_i = 0
\end{aligned}
$$

$$
\begin{aligned}
\mathit{twoPin} = \mathbf{sigrel}\ ((\mathbf{flow}\ p_i, p_v), (\mathbf{flow}\ n_i, n_v), u)\ \mathbf{where} \\
u = p_v - n_v \\
p_i + n_i = 0
\end{aligned}
$$

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

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

- Unfortunately, there was no publicly available implementation available.

- Unfortunately, there was no publicly available implementation available.
- We investigate the implementation of a Functional Hybrid Modelling (FHM) language for non-causal modelling and simulation of physical systems in the form of a domain-specific language embedded in Haskell.

- Unfortunately, there was no publicly available implementation available.
- We investigate the implementation of a Functional Hybrid Modelling (FHM) language for non-causal modelling and simulation of physical systems in the form of a domain-specific language embedded in Haskell.
- The work is supported by the first publicly available prototype implementation of a FHM language.
- The cabalized Haskell source code of a working prototype is available from `http://www.cs.nott.ac.uk/~ggg/`

$$
\begin{array}{l}
\mathit{twoPin} = \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v), u)\ \textbf{where} \\
\quad\quad u = p_v - n_v \\
\quad\quad p_i + n_i = 0
\end{array}
$$

$$
\begin{array}{l}
\mathit{resistor}\ r = \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
\quad\quad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
\quad\quad r * p_i = u
\end{array}
$$

$$
\begin{array}{l}
\mathit{inductor}\ l = \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
\quad\quad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
\quad\quad l * \mathit{der}\ p_i = u
\end{array}
$$

$$
\begin{array}{l}
\mathit{capacitor}\ c = \\
\quad \textbf{sigrel}\ ((\textbf{flow}\ p_i, p_v), (\textbf{flow}\ n_i, n_v))\ \textbf{where} \\
\quad\quad \mathit{twoPin} \diamond ((p_i, p_v), (n_i, n_v), u) \\
\quad\quad c * \mathit{der}\ u = p_i
\end{array}
$$

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

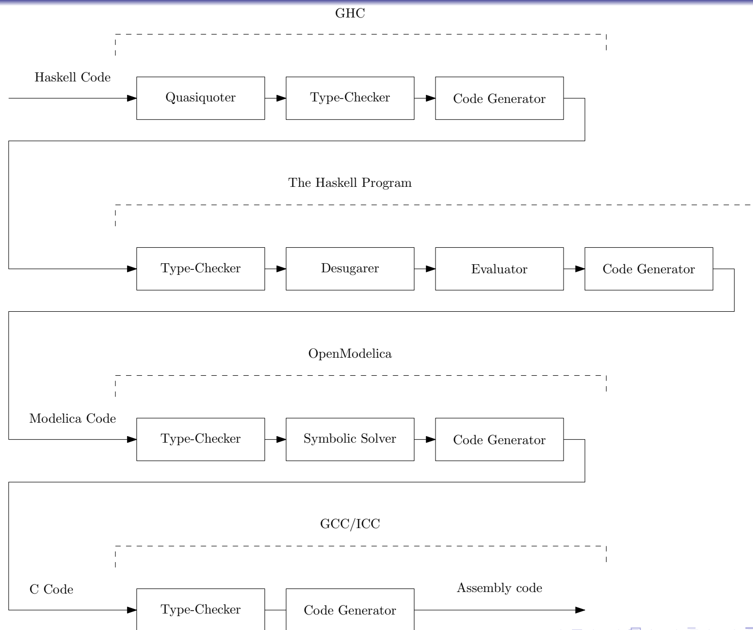
```
import FHM

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

resistor r = [$fhm|
  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
|]

inductor l = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $l$ ∗ der p_i = u
|]

capacitor c = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $c$ ∗ der u = p_i
|]
```

```
simpleCircuit = [$fhm|
  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
|]
```

```
import FHM

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

resistor :: Double → SigRel
resistor r = [$fhm|
  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
|]

inductor :: Double → SigRel
inductor l = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $l$ ∗ der p_i = u
|]

capacitor :: Double → SigRel
capacitor c = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $c$ ∗ der u = p_i
|]

simpleCircuit :: SigRel
simpleCircuit = [$fhm|
  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
|]
```

```
import FHM

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

resistor :: Double → SigRel
resistor r = [$fhm|
  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
|]

inductor :: Double → SigRel
inductor l = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $l$ ∗ der p_i = u
|]

capacitor :: Double → SigRel
capacitor c = [$fhm|
  sigrel ((flow p_i, p_v), (flow n_i, n_v)) where
    $twoPin$ ◇ ((p_i, p_v), (n_i, n_v), u)
    $c$ ∗ der u = p_i
|]
```

```
simpleCircuit :: SigRel
simpleCircuit = [$fhm|
  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)

    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
|]
```


GHC
Haskell Code
Quasiquoter
Type-Checker
Code Generator
The Haskell Program
Type-Checker
Desugarer
Evaluator
Code Generator
OpenModelica
Modelica Code
Type-Checker
Symbolic Solver
Code Generator
GCC/ICC
C Code
Type-Checker
Code Generator
Assembly code

- The method of embedding was inspired by [Mainland, ICFP 2008]
- It employs new features introduced in the development version of the Glasgow Haskell Compiler (GHC) version 6.9.
- In particular, our embedding uses Template Haskell (TH) and quasiquoting.

Future Work

- The next major step is to design and implement switching combinators capable of switching between signal relations during the simulation. This will enable modelling of highly structurally dynamic systems.
- There are number of challenges that needs to be overcome. For examples, state transfer during switches and simulation code generation for highly structurally dynamic systems.

Future Work

- The next major step is to design and implement switching combinators capable of switching between signal relations during the simulation. This will enable modelling of highly structurally dynamic systems.
- There are number of challenges that needs to be overcome. For examples, state transfer during switches and simulation code generation for highly structurally dynamic systems.
- 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.

Conclusions

- We showed how to realize the basic FHM notion of signal relation and language constructs for composing signal relations into complete models as a domain-specific embedding in Haskell. We used Template Haskell and quasiquoting, as pioneered by Mainland to achieve this.

Conclusions

- We showed how to realize the basic FHM notion of signal relation and language constructs for composing signal relations into complete models as a domain-specific embedding in Haskell. We used Template Haskell and quasiquoting, as pioneered by Mainland to achieve this.
- We think this is a promising approach. In addition to the usual benefits of embedded language implementations it allows:
  - Standard compilation technology to be applied for analysis and code generation
  - To handle aspects of the embedded language that are far removed from the host language

Conclusions

- We showed how to realize the basic FHM notion of signal relation and language constructs for composing signal relations into complete models as a domain-specific embedding in Haskell. We used Template Haskell and quasiquoting, as pioneered by Mainland to achieve this.
- We think this is a promising approach. In addition to the usual benefits of embedded language implementations it allows:
  - Standard compilation technology to be applied for analysis and code generation
  - To handle aspects of the embedded language that are far removed from the host language
- The work is supported by the first publicly available prototype implementation of a FHM language. The cabalized Haskell source code of a working prototype is available from `http://www.cs.nott.ac.uk/~ggg/`

Questions?