

# Functional Hybrid Modelling and Simulation in Haskell

George Giorgidze    Henrik Nilsson

Functional Programming Laboratory  
School of Computer Science  
University of Nottingham

2010 Feb 10

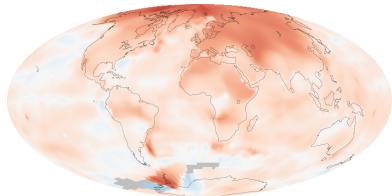
# Outline

- ▶ Declarative languages for modelling and simulation
- ▶ Hybrid and structurally dynamic systems
- ▶ New language for modelling and simulation embedded in Haskell
- ▶ New EDSL implementation approach

# Modelling and Simulation

Developing models and studying their properties and behaviour are of immense theoretical and practical importance.

- ▶ Science
  - ▶ Understanding
  - ▶ Prediction



- ▶ Engineering
  - ▶ Development
  - ▶ Optimisation
  - ▶ Safety



Computers do not have to be involved ...



## But usually they are

Modelling and simulation has been a main application of digital computers from the start:

- ▶ Monte Carlo simulation of nuclear detonation (Manhattan project, Los Alamos)

Recent examples:

- ▶ Los Alamos molecular ribosome model: 2.64 million atoms.
- ▶ The Blue Brain Project: Simulation of 10000 neurons on 8192-processor IBM Blue Gene super computer.

# DSLs for Modelling and Simulation

The need for **domain-specific languages** (DSLs) to allow scientists and engineers to develop models is evident:

- ▶ Domain-experts are usually not programmers
- ▶ The scale of the problems is such that high-level, domain-specific notation and tools are essential to get the work done

# DSLs for Modelling and Simulation

Some examples:

- ▶ Spice (analogue circuits)
- ▶ VHDL-AMS (mixed digital/analogue circuits)
- ▶ NEURON (neuron modelling)
- ▶ gPROMS (process industries)
- ▶ Simulink (domain-neutral)
- ▶ Stateflow (event-driven simulation)
- ▶ Modelica (domain-neutral)
- ▶ ...

# Causal Modelling

- ▶ Ordinary Differential Equation (ODE) in **explicit** form:

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

- ▶ **Causality** (cause-effect relationship) given by the modeller.
- ▶ Causal modelling is the dominating modelling paradigm (e.g. Simulink from MathWorks).

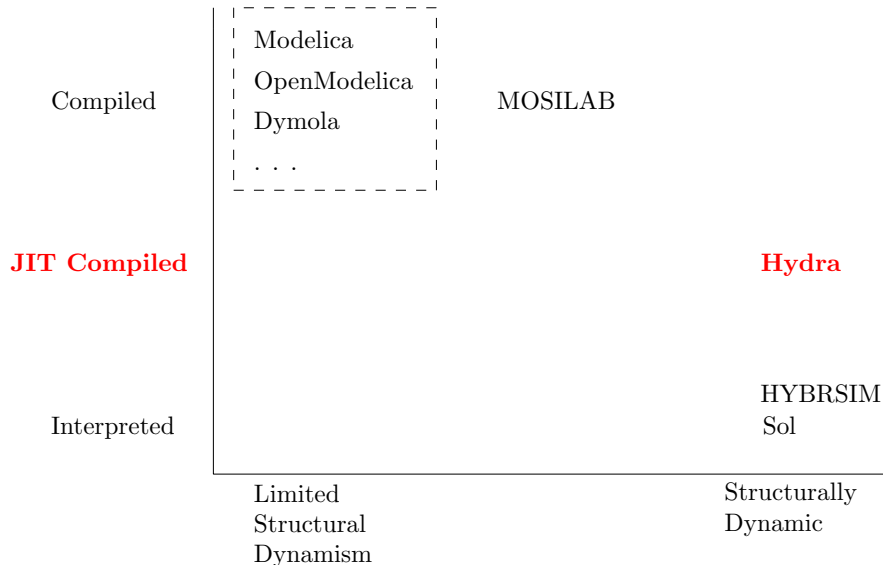
# Non-Causal Modelling

- ▶ Differential Algebraic Equation (DAE) in **implicit** form:

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

- ▶ More **declarative** approach, i.e. focuses on **what** to model rather than **how** to simulate
- ▶ Causality inferred by simulation tool from usage context.
- ▶ Non-causal modelling is a fairly recent development (e.g. Modelica, Dymola, MapleSim)

# Non-Causal Modelling Languages



# Hydra

- ▶ **Two-level language:**
  - ▶ Functional level (time-invariant values)
  - ▶ Signal level (constrains on time-varying values)
- ▶ Functional language primarily serving as a meta language that generates updated constrains given as non-causal DAEs

# Hydra

- ▶ **Two-level language:**
  - ▶ Functional level (time-invariant values)
  - ▶ Signal level (constrains on time-varying values)
- ▶ Functional language primarily serving as a meta language that generates updated constrains given as non-causal DAEs
- ▶ **First class** equational constraints allowing for an evolving model structure by **repeated generation** of updated constraints
- ▶ Instance of Functional Hybrid Modelling (FHM)

# Embedded Domain Specific Languages (EDSLs)

**Embedding** is a powerful and popular way to implement **domain-specific languages** (DSLs). There are two basic approaches to language embeddings:

- ▶ **Shallow:**
  - ▶ Domain-specific notions expressed directly in host language terms
- ▶ **Deep:**
  - ▶ Building representation of domain-specific programs as data
  - ▶ Providing interpreter or compiler

# Embedded Domain Specific Languages (EDSLs)

**Embedding** is a powerful and popular way to implement **domain-specific languages** (DSLs). There are two basic approaches to language embeddings:

- ▶ **Shallow:**
  - ▶ Domain-specific notions expressed directly in host language terms
- ▶ **Deep:**
  - ▶ Building representation of domain-specific programs as data
  - ▶ Providing interpreter or compiler

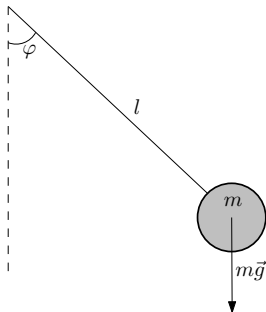
**Mixed-level** embedding combines the two approaches.

# Iterative Staging and JIT Compilation

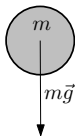
- ▶ **Compilation**: standard tool for EDSLs
  - ▶ Program generation, compilation and execution with a **fixed number of stages**
- ▶ **Iterative staging**: repeated program generation and execution
  - ▶ This work: efficient implementation approach for an iteratively staged EDSL using **mixed-level embedding** and **JIT compilation**

# Breaking Pendulum Example

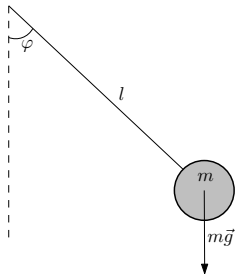
Mode 1 - Pendulum



Mode 2 - Free Fall



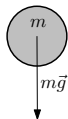
# Pendulum Mode



```
type Coordinate = ( $\mathbb{R}$ ,  $\mathbb{R}$ )  
type Velocity = ( $\mathbb{R}$ ,  $\mathbb{R}$ )  
type Body = (Coordinate, Velocity)  
  
pendulum ::  $\mathbb{R} \rightarrow \mathbb{R} \rightarrow SR$  Body  
pendulum l  $\varphi_0$  = [Hydra]  
  sigrel ((x, y), (vx, vy)) where  
    init  $\varphi$       =  $\varphi_0$   
    init der  $\varphi$  = 0  
    init vx     = 0  
    init vy     = 0  
    x          =  $l * \sin \varphi$   
    y          =  $-l * \cos \varphi$   
    (vx, vy)   = (der x, der y)  
    der (der  $\varphi$ ) + ( $g / l$ ) *  $\sin \varphi$  = 0
```

||

# Free Fall Mode



```
freeFall :: Body → SR Body
freeFall ((x0, y0), (vx0, vy0)) = [Hydra|
  sigrel ((x, y), (vx, vy)) where
    init (x, y)      = (x0, y0)
    init (vx, vy)  = (vx0, vy0)
    (der x, der y)   = (vx, vy)
    (der vx, der vy) = (0, -g)
```

||

# Combining the Two Modes

$pendulumBE :: \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow SR (Body, E Body)$

$pendulumBE \ t \ l \ \varphi_0 = [\$hydra|$

**sigrel** ((( $x, y$ ), ( $v_x, v_y$ )), **event**  $e$ ) **where**

$\$ pendulum \ l \ \varphi_0 \ \$ \diamond ((x, y), (v_x, v_y))$

**event**  $e = ((x, y), (v_x, v_y))$  **when**  $time = \$t \ \$$

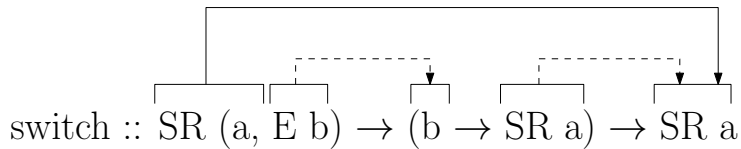
||

$breakingPendulum :: SR Body$

$breakingPendulum = switch (pendulumBE \ 10 \ 1 \ (pi / 4)) \ freeFall$

$switch :: SR (a, E b) \rightarrow (b \rightarrow SR a) \rightarrow SR a$

# The Switch Combinator



# Internal (Untyped) Representation

```
data SigRel =  
    SigRel      Pattern [Equation]  
  | SigRelSwitch SigRel (Expr → SigRel)
```

```
data Equation =  
    EquationInit Expr Expr  
  | EquationEq Expr Expr  
  | EquationEvent String Expr Expr  
  | EquationSigRelApp SigRel Expr
```

# Typed Combinators

- ▶ Phantom types
- ▶ Typing EDSL in Haskell
- ▶ Haskell type checking (GHC)

**data** *SR* *a* = *SR* *SigRel*

**data** *PatternT* *a* = *PatternT* *Pattern*

**data** *ExprT* *a* = *ExprT* *Expr*

**data** *E* *a*

*sigrel* :: *PatternT* *a* → [*Equation*] → *SR* *a*

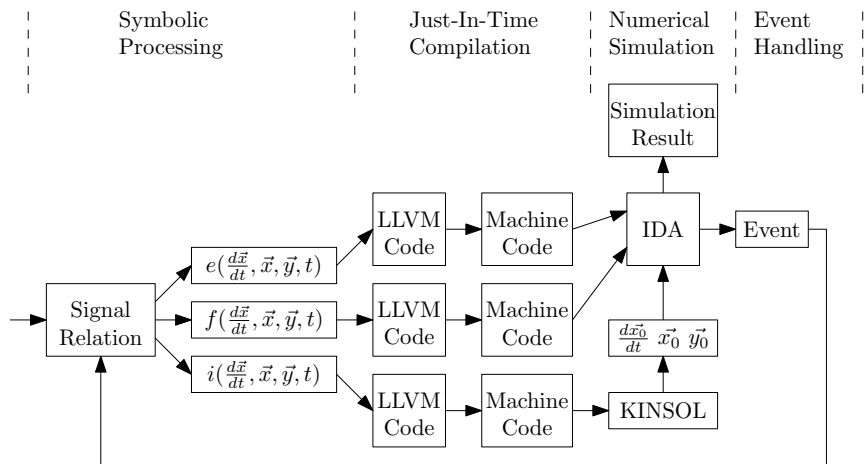
*sigrel* (*PatternT* *p*) *eqs* = *SR* (*SigRel* *p* *eqs*)

*switch* :: *SR* (*a*, *E* *b*) → (*b* → *SR* *a*) → *SR* *a*

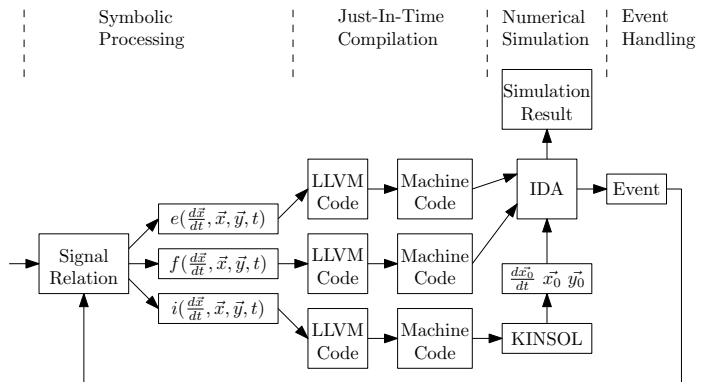
*equationEq* :: *ExprT* *a* → *ExprT* *a* → *Equation*

*equationSigRelApp* :: *SR* *a* → *ExprT* *a* → *Equation*

# Execution model of Hydra



# Future Work



- ▶ Code reuse
- ▶ Mode switching overhead reduction
- ▶ (Soft) real-time simulation

# Related Work

- ▶ **Flask**: EDSL for programming sensor networks [Mainland et al.]
- ▶ **Functional Reactive Programming** (FRP), particularly Yampa [Nilsson et al.]
- ▶ DSLs embedded in **multi-stage** host language (e.g. MetaOCaml) [Taha et al.]
- ▶ Modelling and simulation languages:
  - ▶ MKL [Broman et al.]
  - ▶ MOSILAB [Nytsch-Geusen et al.]
  - ▶ Sol [Zimmer et al.]

# Conclusions

- ▶ Novel approach to the implementation of non-causal modelling languages supporting structural dynamism
  - ▶ Modelling systems that main-stream non-causal language can not handle
  - ▶ First compiled implementation of a non-causal modelling language that supports highly structurally dynamic systems
- ▶ EDSL approach
  - ▶ Mixed-level embedding
  - ▶ Compilation of iteratively staged EDSL
  - ▶ JIT compilation through LLVM framework
- ▶ Papers and implementation:  
<http://cs.nott.ac.uk/~ggg/>

Thank You

Questions?

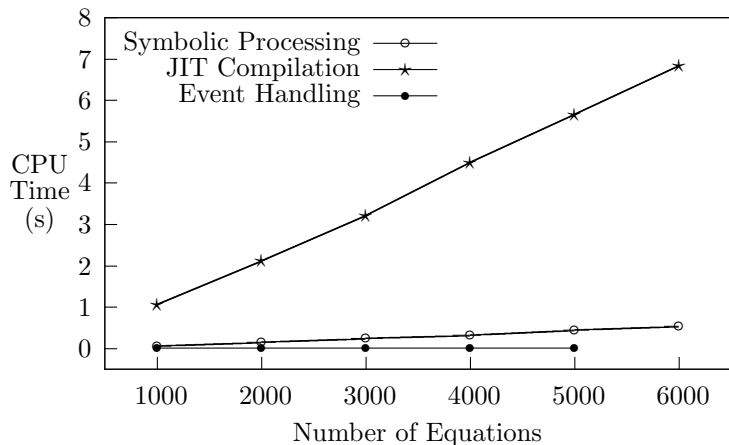
# Bonus Slides

# Performance

	Pendulum $t \in [0, 10)$		Free Fall $t \in [10, 20]$	
	CPU Time		CPU Time	
	s	%	s	%
Symbolic Processing	0.0001	0.2	0.0000	0.0
JIT Compilation	0.0110	18.0	0.0077	9.1
Numerical Simulation	0.0500	81.8	0.0767	90.9
Event Handling	0.0000	0.0	-	-
Total	0.0611	100.0	0.0844	100.0

Table: Time profile of the breaking pendulum simulation

# Performance



**Figure:** Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation

## Shallow

**type** *Region* =  $(\mathbb{R}, \mathbb{R}) \rightarrow \text{Bool}$

*circle* ::  $\mathbb{R} \rightarrow \text{Region}$

*circle* *r* =  $\lambda(x, y) \rightarrow \text{sqrt } (x \uparrow 2 + y \uparrow 2) \leq r$

*rectangle* ::  $\mathbb{R} \rightarrow \mathbb{R} \rightarrow \text{Region}$

*union* :: *Region*  $\rightarrow$  *Region*  $\rightarrow$  *Region*

...

## Deep

**data** *Region* =

*Circle*  $\mathbb{R}$

    | *Rectangle*  $\mathbb{R} \mathbb{R}$

    | *Union* *Region* *Region*

...

Given a parametrised signal relation:

$$sr1 :: \mathbb{R} \rightarrow SR ((\mathbb{R}, \mathbb{R}), E \mathbb{R})$$

we can recursively define a signal relation  $sr$  that describes an overall behaviour by “stringing together” the behaviours described by  $sr1$ :

$$\begin{aligned} sr &:: \mathbb{R} \rightarrow SR (\mathbb{R}, \mathbb{R}) \\ sr \ x &= switch (sr1 \ x) \ sr \end{aligned}$$