Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL

George Giorgidze   Henrik Nilsson

Functional Programming Laboratory
School of Computer Science
University of Nottingham

19th International Workshop on Functional and (Constraint) Logic Programming (WFLP)
Madrid, Spain
2010 Jan 17

Outline

- Modelling and simulation from declarative programming point of view
- 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
- Modelling and simulation languages

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. Cf. Functional Programming.
- Causal modelling is the dominating modelling paradigm (e.g. Simulink).

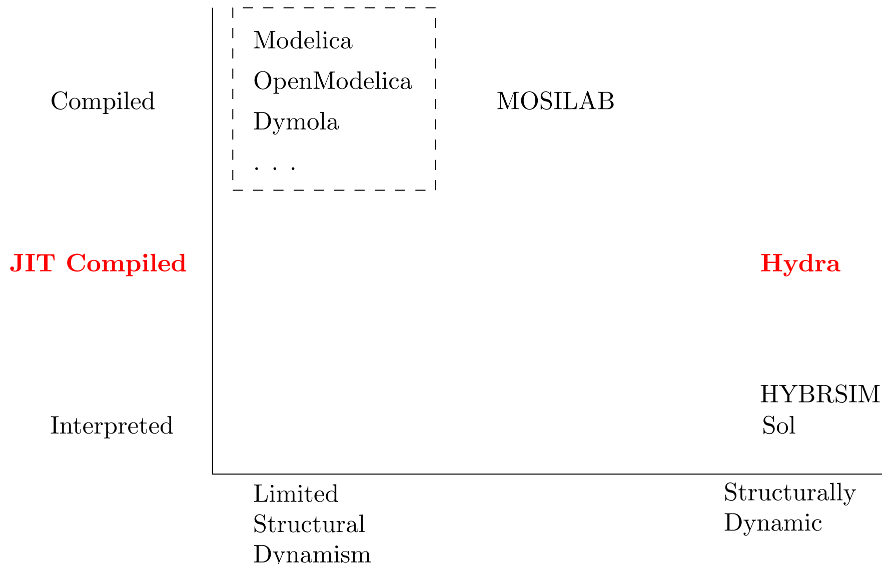
Non-Causal Modelling

- Differential Algebraic Equation (DAE) in implicit form:

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

- Causality inferred by simulation tool from usage context. Cf. Logic Programming.
- Non-causal modelling is a fairly recent development (e.g. Modelica)

Non-Causal Modelling Languages



Hydra

- Example of a combined functional and constraint programming language
- 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

- Example of a combined functional and constraint programming language
- 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

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

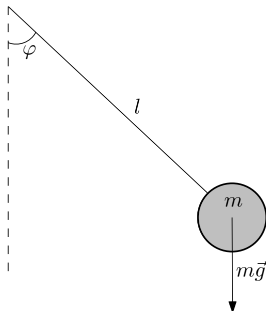
Mixed-level embedding combines the two approaches.

Iterative Staging and JIT Compilation

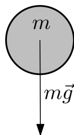
- Compilation: standard tool for EDSLs
  - Program generation, compilation and execution with a fixed number of stages
- Iterative staging: repeated program generation and execution
  - This work: efficient implementation approach for an iteratively staged EDSL using mixed-level embedding and JIT compilation

Breaking Pendulum Example

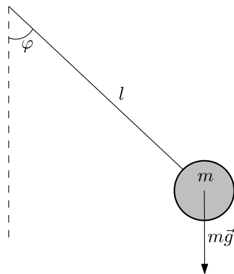
Mode 1 - Pendulum



Mode 2 - Free Fall



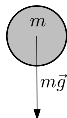
Pendulum Mode



$$
\begin{array}{l}
\textbf{type } \mathit{Coordinate} = (\mathbb{R}, \mathbb{R}) \\
\textbf{type } \mathit{Velocity} = (\mathbb{R}, \mathbb{R}) \\
\textbf{type } \mathit{Body} = (\mathit{Coordinate}, \mathit{Velocity})
\end{array}
$$

$$
\begin{array}{l}
\mathit{pendulum} :: \mathbb{R} \to \mathbb{R} \to \mathit{SR}\ \mathit{Body} \\
\mathit{pendulum}\ l\ \varphi_0 = [\$\mathit{hydra}| \\
\quad \textbf{sigrel}\ ((x, y), (v_x, v_y))\ \textbf{where} \\
\qquad \textbf{init}\ \varphi \qquad\quad = \ \$\, \varphi_0\, \$ \\
\qquad \textbf{init}\ \mathit{der}\ \varphi = 0 \\
\qquad \textbf{init}\ v_x \qquad = 0 \\
\qquad \textbf{init}\ v_y \qquad = 0 \\
\qquad x \qquad\qquad\quad = \qquad \$\, l\, \$ * \mathit{sin}\ \varphi \\
\qquad y \qquad\qquad\quad = \ - \ \$\, l\, \$ * \mathit{cos}\ \varphi \\
\qquad (v_x, v_y) \quad\ = (\mathit{der}\ x, \mathit{der}\ y) \\
\qquad \mathit{der}\ (\mathit{der}\ \varphi) + (\$g\ /\ l\$) * \mathit{sin}\ \varphi = 0 \\
|]
\end{array}
$$

Free Fall Mode



$$
\begin{aligned}
& \mathit{freeFall} :: \mathit{Body} \rightarrow \mathit{SR}\ \mathit{Body} \\
& \mathit{freeFall}\ ((x_0, y_0), (v_{x_0}, v_{y_0})) = [\$\mathit{hydra}| \\
& \quad \mathbf{sigrel}\ ((x, y), (v_x, v_y))\ \mathbf{where} \\
& \qquad \mathbf{init}\ (x, y) \qquad\quad\;\; = (\$x_0\$, \$y_0\$) \\
& \qquad \mathbf{init}\ (v_x, v_y) \qquad\;\; = (\$v_{x_0}\$, \$v_{y_0}\$) \\
& \qquad (\mathit{der}\ x, \mathit{der}\ y) \quad\;\; = (v_x, v_y) \\
& \qquad (\mathit{der}\ v_x, \mathit{der}\ v_y) = (0, -\$g\$) \\
& |]
\end{aligned}
$$

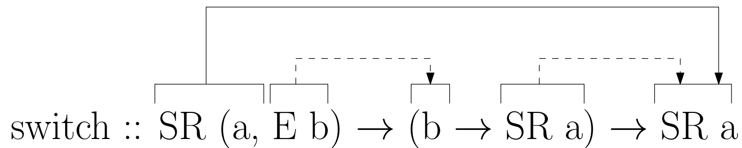
Combining the Two Modes

$$
\begin{array}{l}
\mathit{pendulumBE} :: \mathbb{R} \to \mathbb{R} \to \mathbb{R} \to \mathit{SR}\ (\mathit{Body}, E\ \mathit{Body}) \\
\mathit{pendulumBE}\ t\ l\ \varphi_0 = [\$\mathit{hydra}| \\
\quad \textbf{sigrel}\ (((x, y), (v_x, v_y)), \textbf{event}\ e)\ \textbf{where} \\
\qquad \$\ \mathit{pendulum}\ l\ \varphi_0\ \$\ \diamond ((x, y), (v_x, v_y)) \\
\qquad \textbf{event}\ e = ((x, y), (v_x, v_y))\ \textbf{when}\ \mathit{time} = \$t\ \$ \\
|]
\end{array}
$$

$$
\begin{array}{l}
\mathit{breakingPendulum} :: \mathit{SR}\ \mathit{Body} \\
\mathit{breakingPendulum} = \mathit{switch}\ (\mathit{pendulumBE}\ 10\ 1\ (\mathit{pi}\ /\ 4))\ \mathit{freeFall}
\end{array}
$$

$$
\mathit{switch} :: \mathit{SR}\ (a, E\ b) \to (b \to \mathit{SR}\ a) \to \mathit{SR}\ a
$$

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)

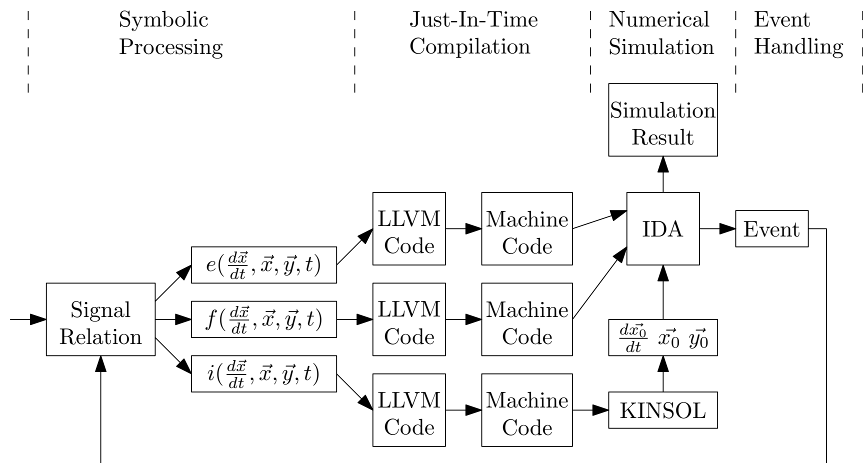
$$
\begin{aligned}
&\mathbf{data}\ SR\ a && = SR && SigRel\\
&\mathbf{data}\ PatternT\ a && = PatternT && Pattern\\
&\mathbf{data}\ ExprT\ a && = ExprT && Expr\\
&\mathbf{data}\ E\ a
\end{aligned}
$$

$$
\begin{aligned}
&sigrel :: PatternT\ a \rightarrow [Equation] \rightarrow SR\ a\\
&sigrel\ (PatternT\ p)\ eqs = SR\ (SigRel\ p\ eqs)
\end{aligned}
$$

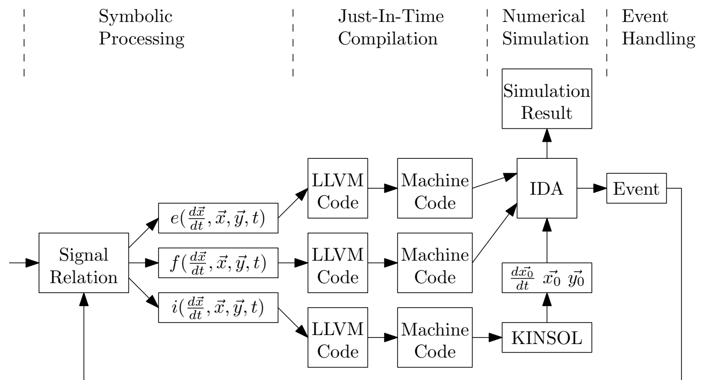
$$
switch :: SR\ (a, E\ b) \rightarrow (b \rightarrow SR\ a) \rightarrow SR\ a
$$

$$
\begin{aligned}
&equationEq :: ExprT\ a \rightarrow ExprT\ a \rightarrow Equation\\
&equationSigRelApp :: SR\ a \rightarrow ExprT\ a \rightarrow Equation
\end{aligned}
$$

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.]
- Accelerate: EDSL for data-parallel array computations on GPUs [Chakravarty 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
- Paper 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

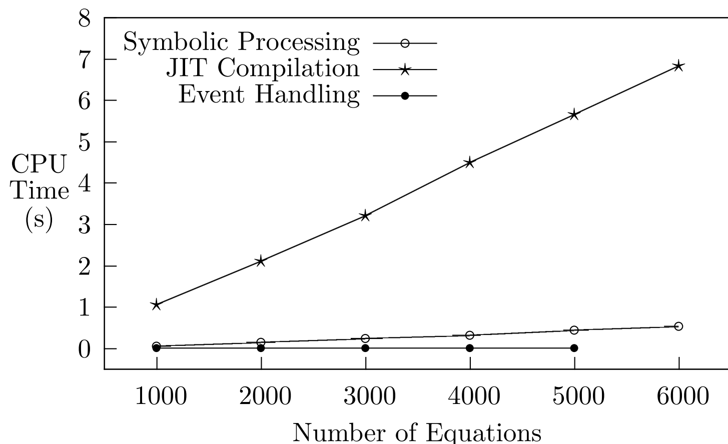


Figure: Plot demonstrating how CPU time spent on mode switches grows as number of equations increase in structurally dynamic RLC circuit simulation

Shallow

$$\textbf{type}\ \mathit{Region} = (\mathbb{R}, \mathbb{R}) \rightarrow \mathit{Bool}$$

$$\mathit{circle} :: \mathbb{R} \rightarrow \mathit{Region}$$
$$\mathit{circle}\ r = \lambda(x, y) \rightarrow \mathit{sqrt}\ (x \uparrow 2 + y \uparrow 2) \leqslant r$$

$$\mathit{rectangle} :: \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathit{Region}$$

$$\mathit{union} :: \mathit{Region} \rightarrow \mathit{Region} \rightarrow \mathit{Region}$$

$$\ldots$$

Deep

$$\begin{array}{ll}\textbf{data}\ \mathit{Region} = & \\ & \mathit{Circle}\ \mathbb{R} \\ \mid & \mathit{Rectangle}\ \mathbb{R}\ \mathbb{R} \\ \mid & \mathit{Union}\ \mathit{Region}\ \mathit{Region} \\ \ldots & \end{array}$$

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 $sr1$:

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