Using Gradually Typed Symbolic Expressions for Embedding Domain-Specific Modeling Languages

NUS, Singapore
November 5, 2018

Financially supported by the Swedish Foundation for Strategic Research.

David Broman
KTH Royal Institute of Technology, Sweden

Joint work together with Jeremy G. Siek, Indiana University, USA

This research is in part financially supported by the Swedish Foundation for Strategic Research (FFL15-0032),

Agenda

Part I

Gradually typed symbolic expressions

modelyze

Part II

Formalization of the core language

$\lambda^{<\star>}$

Part III

Case study: EOO DSLs

Part I

Gradually typed symbolic expressions

modelyze

Embedded DSL: Strengths and Challenges

Embedded DSLs

Strengths
- Lower development costs
- Requires potentially less PL knowledge

Challenges
- To get good end-user experience (seamless integrated)
  - Leaking abstractions and reasonable error messages
- Expressive and easy to use for the domain expert

Statically or dynamically typed host language?

Statically Typed Approaches

- **Scala Virtualized** (Rompf et al. 2012)
- **LMS** (Rompf & Odersky, 2010)
- **Finally Tagless** (Carette et al., 2009)
- **Embedding in Haskell**

Pros:
- Potentially precise error messages
- Type safe transformations

Cons:
- Learning curve (Monads, GADTs etc.)
- Boilerplate code

Our approach:

Combine static and dynamic typing in the context of embedded DSLs.

Dynamically Typed Approaches

- **Racket**
- **LISP**
- **Python**
- **Julia**

Pros:
- "Easy" to get started with
- No expressiveness limitations

Cons:
- No static transformation guarantees
- Type errors discovered at runtime

Related Work

Implementing DSLs

Compiler construction
- JastAdd (Ekman & Hedin, 2007)
- MetaModelica (Pop &

Preprocessing and ten
- C++ Templates (Ve
- Template Haskell (
- Stratego/XP (Brave

Embedded DSLs
- Haskell DSELs, e.g Lava (Bjesse et al. Augustsson, 2008)
- FHM (Nilsson et al., 2003)
- ForSyDe (Sander & Jantsch, 2004)
- Lightweight modular staging (Rompf and Odersky, 2010)
- Shallow embedding and PE (Leißa et al., 2015)

Combining Dynamic and Static Typing

- Gradual Typing (Siek & Taha, 2007)
- Soft Typing (Cartwright & Fagan, 1991)
- h typecase (Abadi et al., 1991)
- acket (Tobin-Hochstadt,
- Wrigstad et al., 2010)

de and Data type

- es LISP, Mathematica
- a & Sheard, 2000)
- Jones et al.,2006; Xi et al., 2003; Cheney & Ralf, 2003)
- Open Data types (Löh & Hinze, 2006)
- Pattern Calculus (Jay, 2009)
- Syntactic library (Axelsson, 2012)

Contribution:
Gradually Typed Symbolic Expressions

Aim:
- To provide seamless integration and good error messages for the end user (static typing)
- Enable simple and flexible transformations (dynamic typing)

Our approach

(MODEL and analYZE)

Small, simple, host language for **embedding domain-specific languages** (DSL) of different models of computation (MoC)

Gradually typed functional language (call-by-value)

Novelty: Gradually typed symbolic expressions

Open source: www.modelyze.org

Interpreter implemented Ocaml

A DSL for mathematical modeling embedded in Modelyze

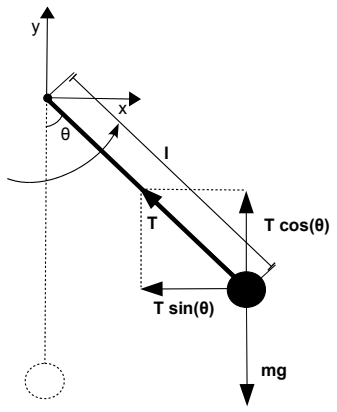
```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Equations and initial values are defined declaratively, just as the mathematical equations

Differential-Algebraic Equations (DAEs).

y'' means second order derivative



$$-T \cdot \frac{x}{l} = m\ddot{x}$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y}$$

$$x^2 + y^2 = l^2$$

$$x(0) = l\sin(\theta_s)$$

$$y(0) = -l\cos(\theta_s)$$

Declarative Mathematical Model

Which parts are part of the host language (Modelyze)?

Unknowns are internally represented as typed symbols

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

$s : \tau$

Fresh (unique) symbol

Tagged with a type

Variable **x** is bound to a fresh symbol of type **<Real>**

$<\tau>$ Symbolic type

Release the user from annotation burden

Symbols cannot be bound to values, so **x^2** would crash at runtime

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Use quasi-quoting to mix symbolic expressions and program code?

Using MetaML syntax < > for quotation and ~ for anti-quoting (escape)

```
<~x^2. + ~y^2. = ~((fun t -> <t>)l^2.)>;
```

Heavy annotation burden for the end-user

Constructing typed symbolic expressions: Symbol Lifting Analysis

Symbol Lifting Analysis: During type checking, lift expressions that cannot be safely evaluated at runtime into symbolic expressions (data).

$$\Gamma \vdash_L e \rightsquigarrow e' : \tau$$

Rewritten to prefix curried form

```
(((/) x) l)
```

where

```
(/):Real-> Real -> Real
x:<Real>
l:Real
```

```
def Pendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y (-l*cos(angle));

    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2. + y^2. = l^2.;
}
```

Can be viewed as a form of type-based binding-time analysis.

Also, similar to the Rep type in LMS (Rompf & Odersky, 2010)

```
(((sval(/):Real-Real->Real) @ x) @ (sval l:Real))
```

Resulting type `<Real>`

Division cannot be performed, lift expression to type `<Real-> Real -> Real>`.

Term `sval e:T` wrapps `e` and results in type `<T>`

Term `e1@e2` is a symbolic application, represented as a tuple.

Deconstructing typed symbolic expressions: Pattern Matching using dynamic types

Defining a new symbolic data type

Symbols used as data constructors

```
type Equations
def (=) : <Real->Real->Equations>
def (;) : <Equations->Equations->Equations>
```

Query for all unknowns in a model instance

Dynamic symbolic type `<Dyn>`

Accumulator

```
def uk(e:<Dyn>,acc:UkSet) -> UkSet = {
  match e with
  | e1 e2 -> uk(e2,uk(e1,acc))
  | sym:<Real> -> Set.add e acc
  | _ -> acc
}
```

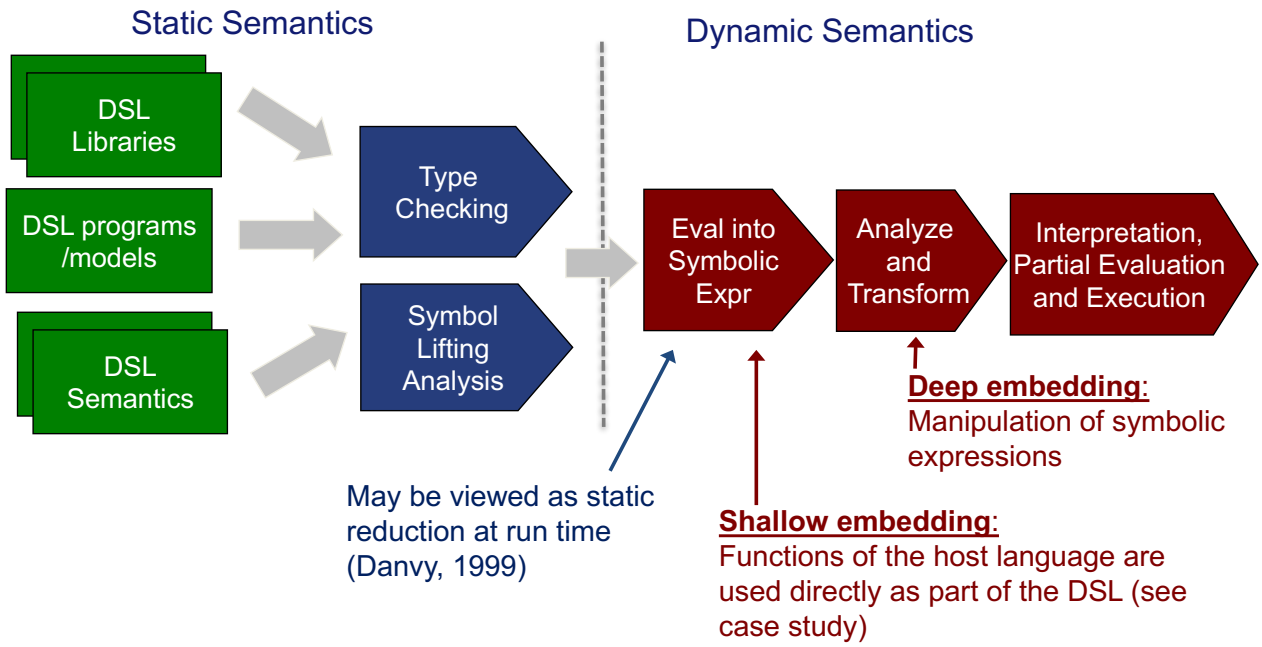
Matching a uniform data structure: no boilerplate code. **3 different forms.**

1. Matching **Symbolic applications** (Written without @)

2. Matching **typed symbols.** In this case of type `<Real>` i.e., unknowns in the model.

3. Lifted **symbol values**, `sval`(not shown in the example)

Embedding and Execution Process



Embedding and Execution Process

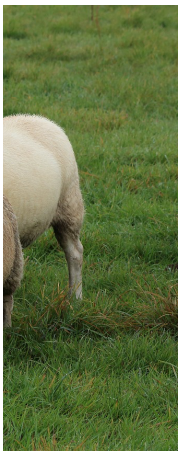
SHallow and d**EEP**

Let us mispronounce this a bit…

Deep embedding:
Manipulation of symbolic expressions

Shallow embedding:
Functions of the host language are used directly as part of the DSL (see case study)

Embedding and Execution Process



Cheap embedding
The aim of combining the convenience of shallow embedding with the power of deep embedding.

Other names for combining shallow and deep embedding:
neritic (Augustsson, 2012) and Yin-Yang in Scala (Jovanovic et al., 2014)

Deep embedding:
Manipulation of symbolic expressions

SHallow and d**EEP**

Let us mispronounce this a bit…

Shallow embedding:
Functions of the host language are used directly as part of the DSL (see case study)

Part II

Formalization of the core language

$\lambda^{<\star>}$

Formal Semantics

Three intermediate languages.

$$\lambda^{<\star>} \xrightarrow{\text{Symbol lifting}} \lambda_L^{<\star>} \xrightarrow{\text{Cast insertion}} \lambda_{LC}^{<\star>}$$

Core — Lifted — Cast inserted

$\lambda^{<\star>}$

Base Types	$B \in \mathbb{G}$	
Sym Data Types	$D \in \mathbb{D}$	
Types	τ	$::= B \mid \tau \rightarrow \tau \mid \star \mid <\tau> \mid D$
Variables	$x, y \in \mathbb{X}$	
Symbols	$s \in \mathbb{S}$	
Constants	$c \in \mathbb{C}$	
Expressions	e	$::= x \mid \lambda x{:}\tau.e \mid e\ e \mid c \mid \texttt{error} \mid v(\tau) \mid \texttt{case}(e, p, e, e)$
Patterns	p	$::= \texttt{sym}{:}\tau \mid x\ @\ x \mid \texttt{sval}\ x{:}\tau$

- **Small-step operational semantics**
- **Translation relations**
- **Type system**
- **Type soundness proof (progress and preservation theorems)**

$\lambda_L^{<\star>}$ (extends $\lambda^{<\star>}$)

Expressions	e	$+= e\ @\ e \mid \texttt{sval}\ e{:}\tau$

$\lambda_{LC}^{<\star>}$ (extends $\lambda_L^{<\star>}$)

Expressions	e	$+= \langle \tau \Leftarrow \tau \rangle e \mid s{:}\tau$

Part III

Case study: Equation-based DSLs



Examples of Application Areas

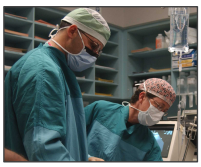
Cyber-Physical Systems (CPS)

Automotive
(systems of systems)

Industrial
Automation

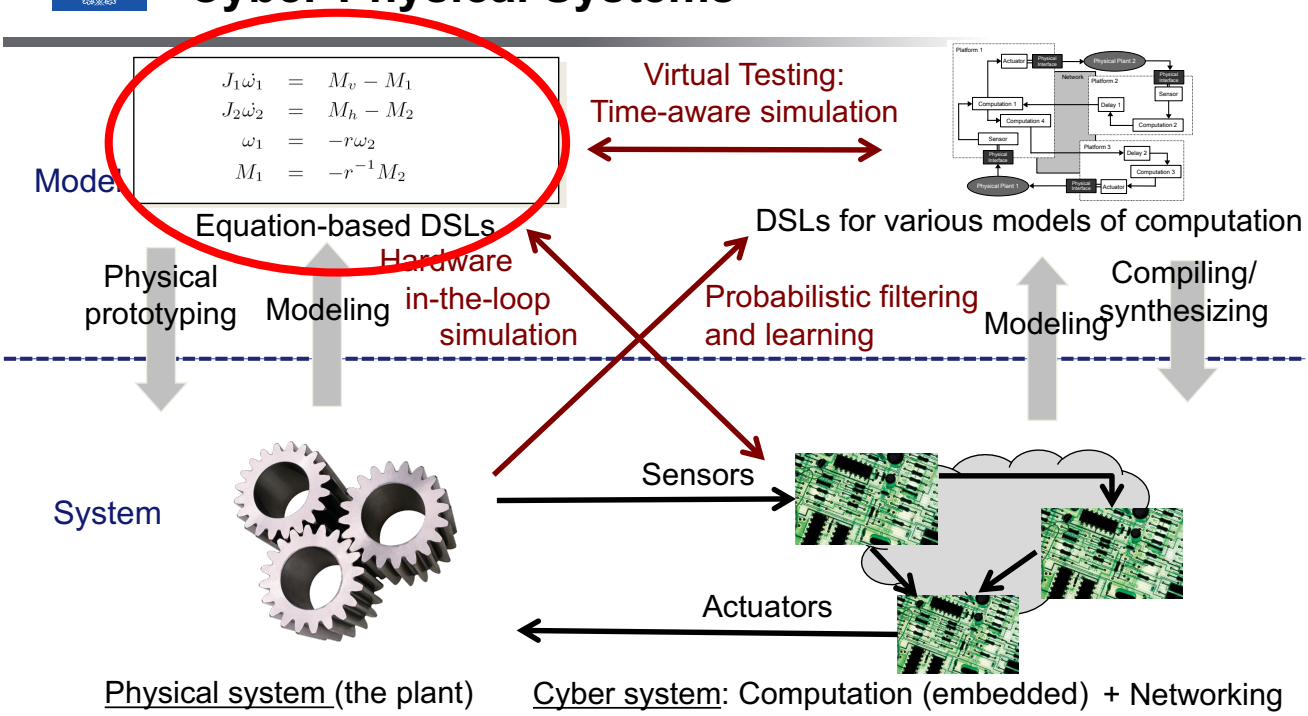
Aircraft
(traditional or
autonomous)

Satellites

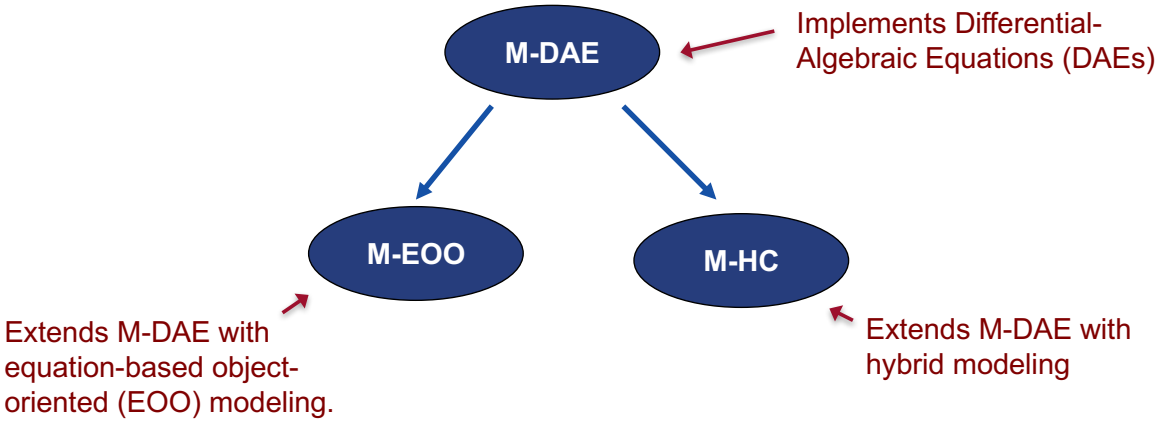


Medical
Equipment

Heterogeneous Modeling of Cyber-Physical Systems



Three Equation-Based DSLs



M-DAE

M-DAE

```
1: def Pendulum(m:Real,l:Real,a:Real)={
2:     def x,y,T:Real;
3:     init x (l*sin(a));
4:     init y (-l*cos(a));
5:     -T*x/l = m*x'';
6:     -T*y/l - m*g = m*y'';
7:     x^2. + y^2. = l^2.;
8:     probe "x" = x;
9:     probe "y" = y;
10:}
```

Evaluation to normal form

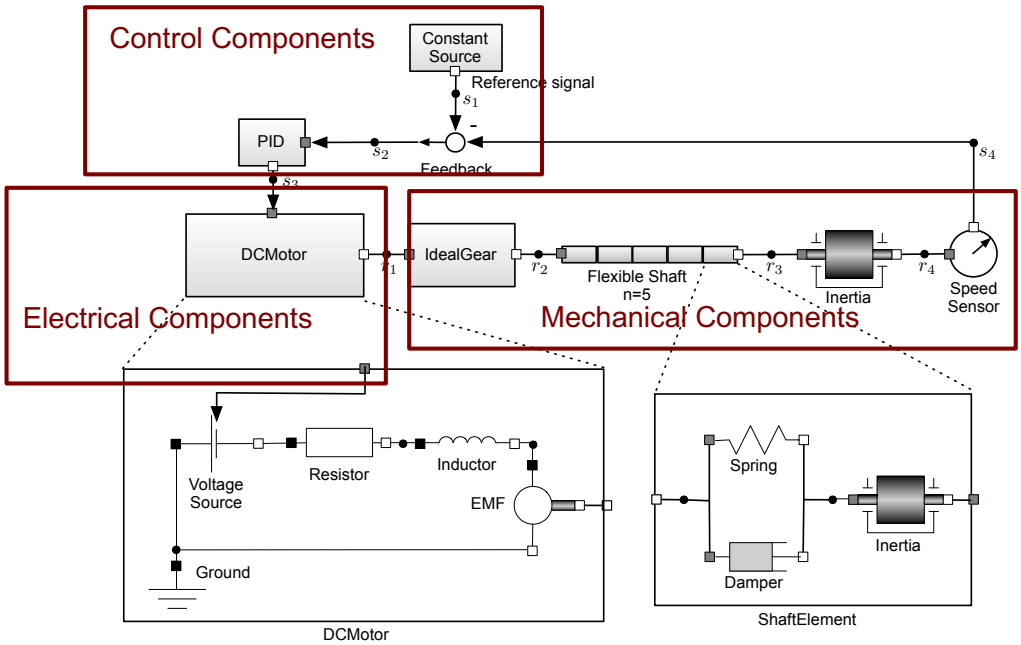
Phase I: `daeInit()`

```
elaborateProbes()
elaborateDerivatives()
indexReductionPantelides()
    - pantelides algorithm
    - symbolic differentiation
makeResidual()
    - online partial evaluation
makeInitValues()
```

Phase II: `simLoop()`

```
def eval(e:<Dyn>,yy:Vars,yp:Vars) -> Dyn = {
    match e with
    | der x -> ...
    | sym:Real -> eval(yy(e),yy,yp)
    | f e -> (eval(f,yy,yp)) (eval(e,yy,yp))
    | sval v:Dyn -> v
    | _ ->  error "Unsupported construct"
```

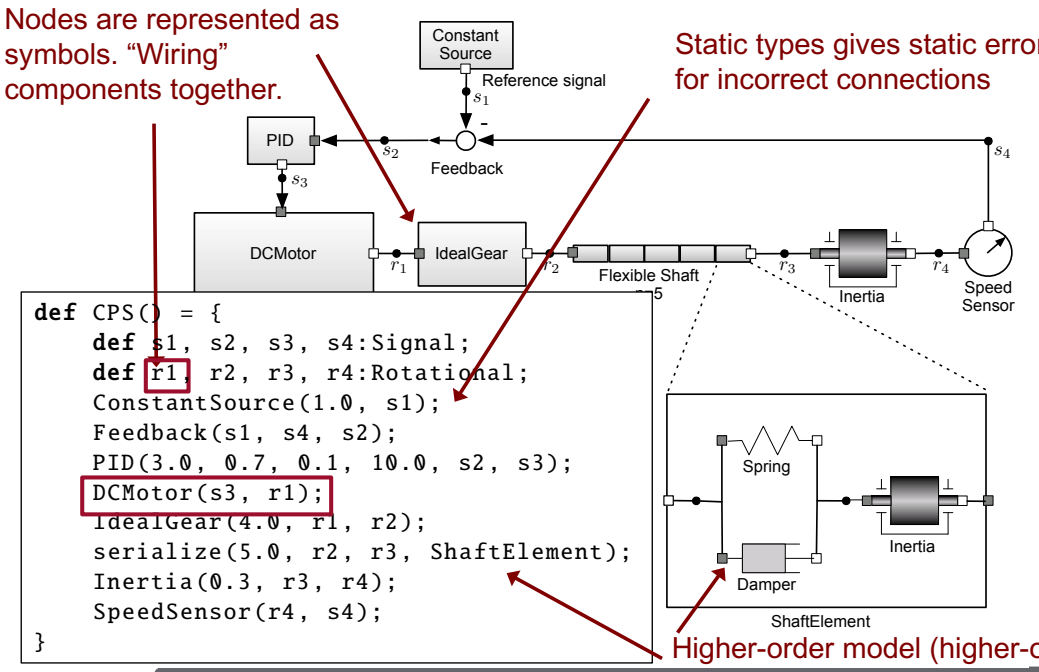
M-EOO



M-EOO

Nodes are represented as symbols. “Wiring” components together.

Static types gives static error for incorrect connections



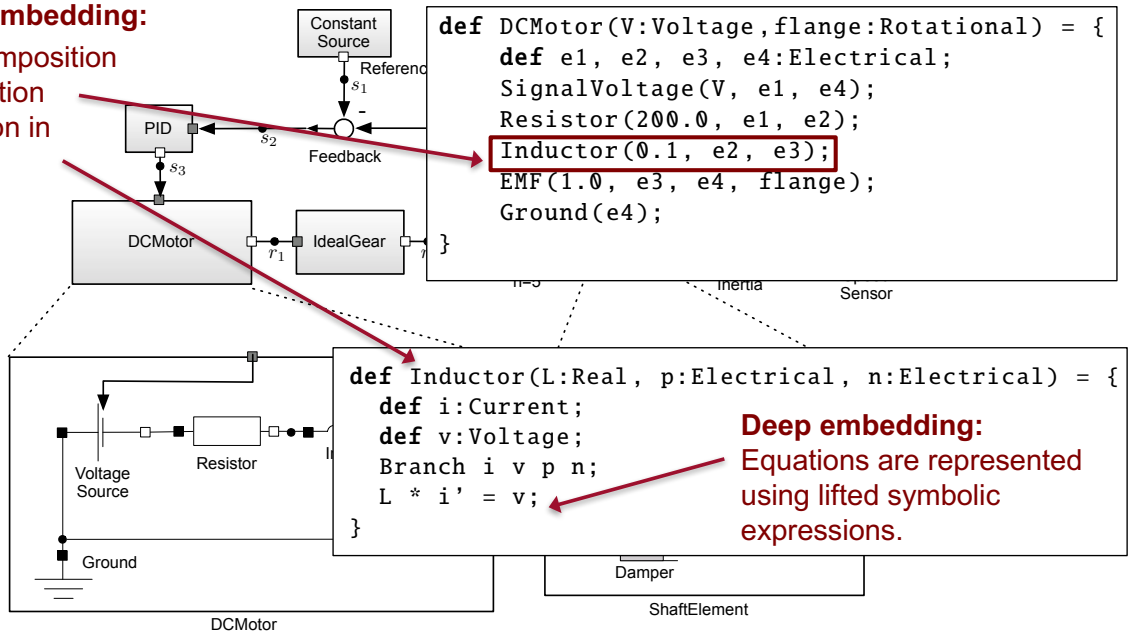

```
def CPS() = {
    def s1, s2, s3, s4:Signal;
    def r1, r2, r3, r4:Rotational;
    ConstantSource(1.0, s1);
    Feedback(s1, s4, s2);
    PID(3.0, 0.7, 0.1, 10.0, s2, s3);
    DCMotor(s3, r1);
    IdealGear(4.0, r1, r2);
    serialize(5.0, r2, r3, ShaftElement);
    Inertia(0.3, r3, r4);
    SpeedSensor(r4, s4);
}
```

Higher-order model (higher-order function)

M-EOO

Shallow embedding:

-Model composition using function composition in the host language.




```
def DCMotor(V:Voltage,flange:Rotational) = {
    def e1, e2, e3, e4:Electrical;
    SignalVoltage(V, e1, e4);
    Resistor(200.0, e1, e2);
    Inductor(0.1, e2, e3);
    EMF(1.0, e3, e4, flange);
    Ground(e4);
}
```

```
def Inductor(L:Real, p:Electrical, n:Electrical) = {
  def i:Current;
  def v:Voltage;
  Branch i v p n;
  L * i' = v;
}
```

Deep embedding:
Equations are represented using lifted symbolic expressions.

M-EOO

Normal evaluation results in the collapsed hierarchy

Reuses M-DAE
Translational DSL reuse

Evaluation to normal form

Phase I: `elaborateConnections()`

`potentials()`
Add potential equations to the equation system. E.g. the voltage potential is the same at each connect node in the electrical domain. Collect connect nodes and remove branches.

`sumzero()`
Generate and add sum-to-zero equations, following Kirchhoff's current law.

Phase II: `mdae()`
Reuse everything from the DSL m-dae

`daeInit()`
Reuse all parts from DSL m-dae.

`simLoop()`
Reuse all parts from DSL m-dae.

`pprintSimulation()`
Pretty print simulation for plotting.

Additional equations are added

M-HC


```
def BreakingPendulum(m:Real, l:Real, angle:Real) = {
    def x,y:Position;
    def time:Real;
    def Pendulum, BouncingBall:Mode;
    init x (l*sin(angle));
    init y (-l*cos(angle));
    time' = 1.0;
    probe("y") = y;

    hybridchart initmode Pendulum {
        mode Pendulum {
            def T:Force;
            -T*x/l = m*x'';
            -T*y/l - m*g = m*y'';
            x^2. + y^2. = l^2.;
            transition BouncingBall
                when (time >= 3.5 && T >= 4.0) action nothing;
        };
        mode BouncingBall {
            x'' = 0.;
            -g = y'';
            transition BouncingBall
                when (y <= -4.0) action (y' <-  y' * -0.7);
        };
    };
}
```

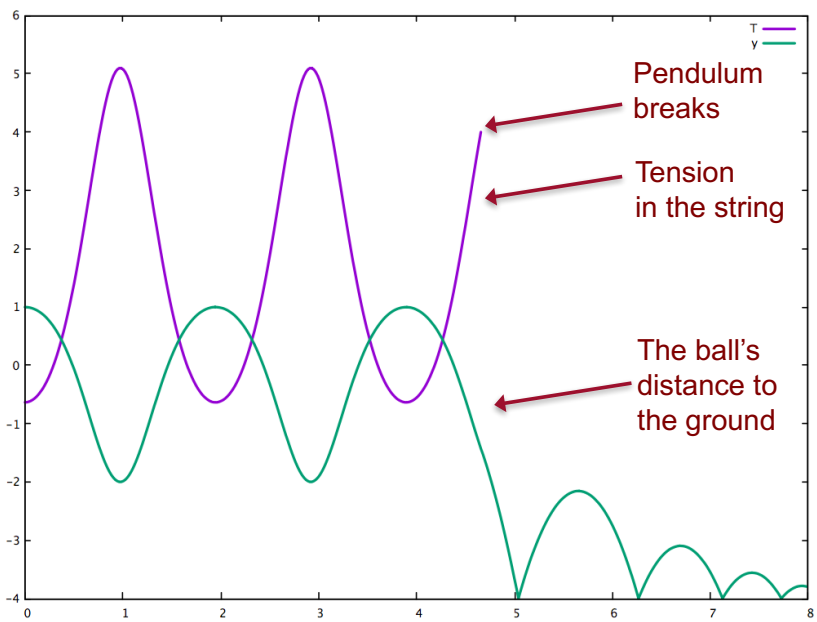
Hybrid DSL with both discrete and continuous simulation

Different modes where the equation-system can be switched over time

Note: all these constructs are typed symbols

Self-transitions, make a ball bounce.

M-HC



Conclusions

Some key take away points:

- Combining static and dynamic typing in a embedded DSL setting give new possibilities.
- Static typing is highly important for the end-user experience (including error reporting)
- Dynamic typing makes expressive transformations simple, but removes typing guarantees.

David Broman and Jeremy G. Siek. **Gradually Typed Symbolic Expressions.** In Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2018), Los Angeles, ACM, 2018.

Thanks for listening!