Principles of Equation-Based Object-Oriented Modeling and Languages

Module C: Modelyze – Defining Equation-Based DSLs

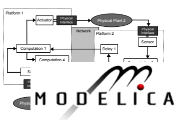
Mini-course, Scuola Superiore Sant'Anna, Pisa, Italy.
December 9-10, 2014

David Broman

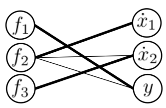
Associate Professor, KTH Royal Institute of Technology

Assistant Research Engineer, University of California, Berkeley

Course Structure



Module A
EOO Languages and Modelica Fundamentals



Module B
DAEs and Algorithms in EOO Languages

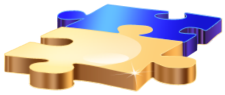
Module C
Modelyze – Defining Equation-Based DSLs

Module D
Co-simulation and the Functional Mock-up Interface

Agenda

Part I

Modelyze Overview

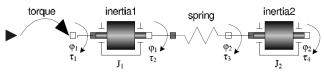


Part II

Modelyze Demo

Part III

Node-Based Connection Semantics



Part IV

Formal Semantics

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \langle\tau_{11} \to \tau_{12}\rangle \quad \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \quad \lceil e_2' : \tau_2 \rceil = e_2'' \quad \langle\tau_{11}\rangle \sim \lceil \tau_2 \rceil}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1' @ e_2'' : \langle\tau_{12}\rangle} \text{(L-APP5)}$$

Part I

Modelyze Overview

Problem: Expressiveness and Analyzability

Cannot express all modeling or analysis needs.
Limited to what the modeling language can provide.

Language versions: A, v1.0 → A, v1.1 → A, v2.0 → A, v2.2

Standard library versions: L, v1.0 → L, v1.1 → L, v2.0 → L, v2.2

Modelica: A new language definition approximately every second year

Uses

- Simulation
- Optimization
- Code generation for real-time
- Model export
- Grey-box system identification

etc.

C, v1.0

B, v1.0 → A, v1.1

gives many dialects and different languages (e.g. Mosilab, Optimica)

A, v1.0 → A, v1.1 → A, v2.0 → A, v2.2

What is Modelyze?

Modelyze
(MODEL and analYZE)

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

Key aspect: Both the DSL and models in the DSL are defined in Modelyze

Gradually typed functional language (call-by-value)

Novelty: Typed symbolic expressions

Formal semantics for a core of the language.
Proven type soundness for the core.

Prototype implementation (interpreter). Evaluated for series of equation-based DSLs.

Challenges and Contributions

1. Provide seamless integration between host language and embedded DSL

Performed together with type checking.

Symbol Lifting Analysis

2. Make it easy to transform and analyze equations (domain expert)

Pattern matching on symbolic expressions

Gradual Typing
+
Typed Symbolic Expressions

3. Provide domain specific errors (model engineer)

Static type checking

Mixing Static Typing vs. Dynamic Typing

Based on Gradual Typing (Siek & Taha, 2006)

Dynamic typing for meta-programming → Increased expressiveness for the domain expert

Tradeoff between simplicity to learn, safety, and expressiveness

Static typing for DSL constructs → Precise Error Reporting for DSL User

Related Work

Implementing DSLs

Compiler construction
- JastAdd (Ekman & Hedin, 2007)
- MetaModelica (Pop & Fritzson, 2006)

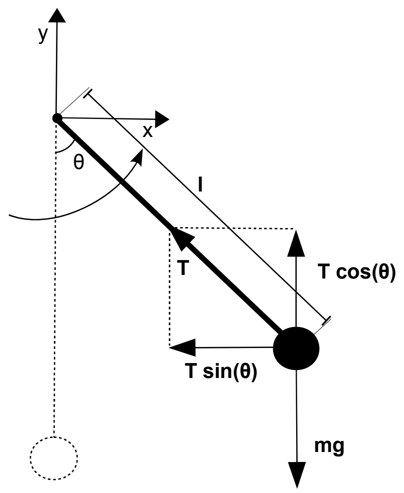
Preprocessing and template metaprogramming
- C++ Templates (Veldhuizen, 1995)
- Template Haskell (Sheard & Peyton Jones, 2002)
- Stratego/XP (Bravenboer et al., 2008)

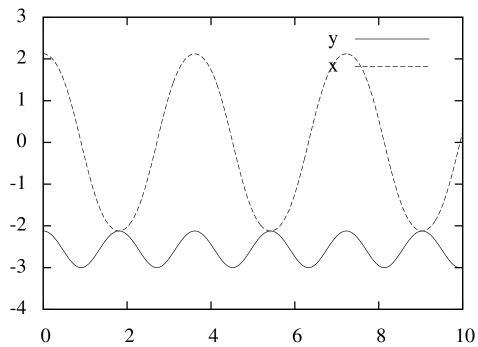
Embedded DSLs
- Haskell DSELs, e.g., Fran (Ellito & Hudak, 1997), Lava (Bjesse et al. 1998), and Paradise(Augustsson, 2008)
- FHM (Nilsson et al., 2003)
- ForSyDe (Sander & Jantsch, 2004)
- Pure embedding (Higher-order functions, polymorphism, lazy evaluation, type classes) (Hudak, 1998)

Combining Dynamic and Static Typing
- Gradual Typing (Siek & Taha, 2007)
- Soft Typing (Cartwright & Fagan, 1991)
- Dynamic type with typecase (Abadi et al., 1991)
- Typed Scheme, Racket (Tobin-Hochstadt, Felleisen, 2008)
- Thorn, like types (Wrigstad et al., 2010)

Representing Code and Data type
- Dynamic languages LISP, Mathematica
- MetaML <T> (Taha & Sheard, 2000)
- GADT (Peyton 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)

Pendulum Example





$$-T \cdot \frac{x}{l} = m\ddot{x} \qquad x(0) = l\sin(\theta_s)$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y} \qquad y(0) = -l\cos(\theta_s)$$

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

Differential-Algebraic equations

Algebraic constraint

Initial values

Declarative Mathematical Model

Using function abstraction to define the model

Unknowns are given types but not bound to values

```
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

$$-T \cdot \frac{x}{l} = m\ddot{x} \qquad x(0) = l\sin(\theta_s)$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y} \qquad y(0) = -l\cos(\theta_s)$$

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

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.;
}
```

$\boxed{s : \tau}$

Fresh (unique) symbol

Tagged with a type

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

$\boxed{<\tau>}$

Symbolic type

Release the user from annotation burden

```
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.;
}
```

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

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

Symbol Lifting Analysis (SLA)

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

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

```
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.;
}
```

Rewritten to prefix curried form

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

where

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

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

Resulting type `<Real>`

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

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

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

Pattern Matching on Symbolic Expressions

Dynamic symbolic type <?>

Accumulator Sets of symbolic type <Real>

Query for all unknowns in a model instance

```
def getUnknowns(exp:<?>, acc:(Set <Real>)) -> (Set <Real>) = {
    match exp with
    | e1 e2 -> getUnknowns(e2,getUnknowns(e1,acc))
    | sym:Real -> Set.add exp acc
    | _ -> acc
}
```

Uniform data structure, no boilplate code (matching on symbolic applications)

Match all symbols of type <Real> i.e., unknowns in the model.

```
getUnknowns(Pendulum(5,3,45*pi/180),Set.empty)
```

Returns a set with 3 symbols (representing `x`, `y`, and `T`).

Static Error Checking at the DSL Level

Syntactically correct model (host syntax)

Static type error instead of dynamic error during translation/pattern matching.

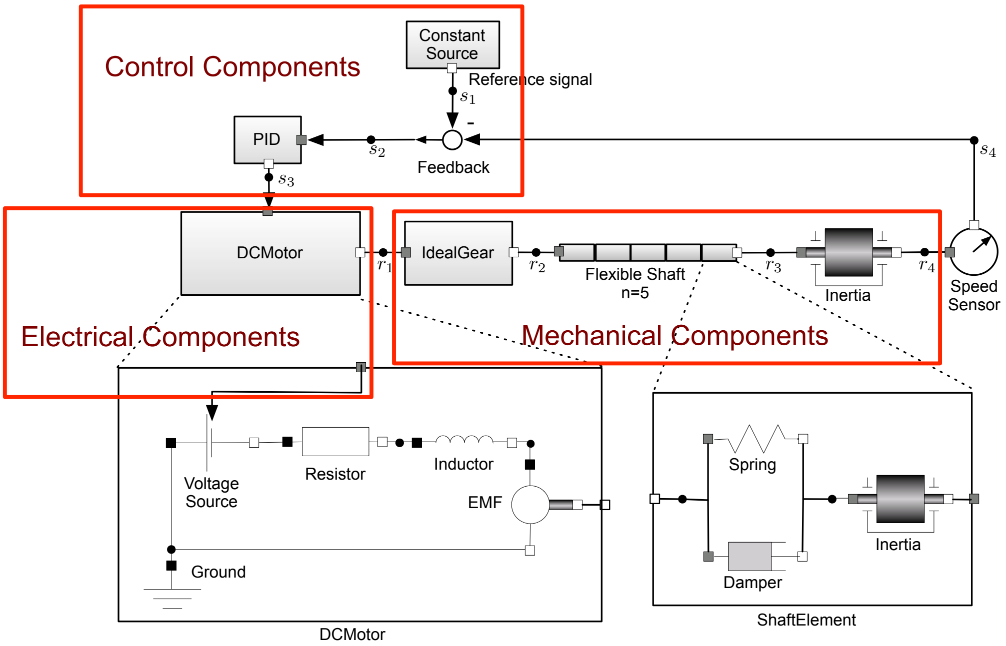
```
def ModifiedPendulum(m:Real,l:Real,angle:Real) = {
    def x,y,T:Real;
    init x (l*sin(angle));
    init y;                    //Error: Missing initial value

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

Quite intuitive error messages at the DSL level.

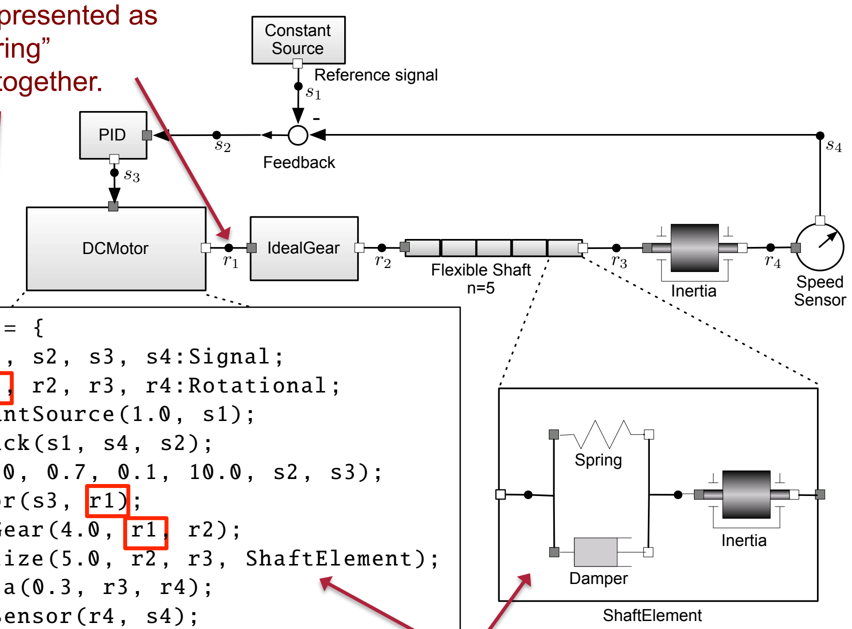
```
modifiedpendulum.moz 4:10-4:10 error: Missing argument
of type 'Real'.
```

Mechatronic Control Example (ModelyzeEOO)



Mechatronic Control Example

Nodes are represented as symbols. "Wiring" components together.

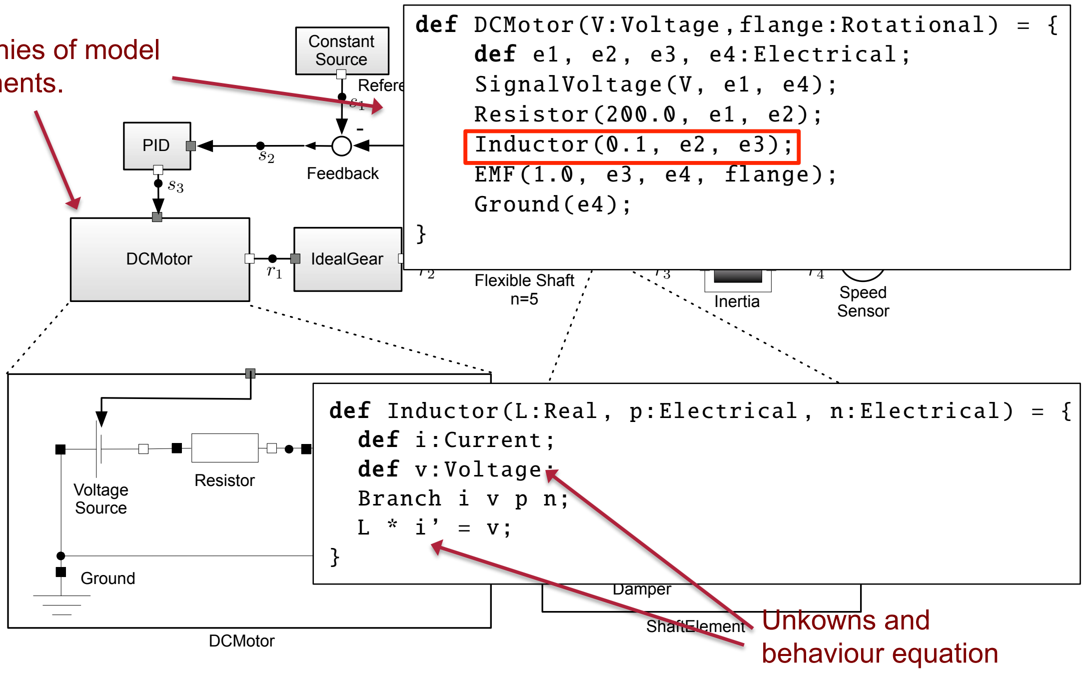



```
def CPS() = {
    def s1, s2, s3, s4:Signal;
    def r1, r2, r3, r4:Rotational;
    ConstantSource(1.0, s1);
    Feedback(s1, s4, s2);
    PID(3.0, 0.7, 0.1, 10.0, s2, s3);
    DCMotor(s3, r1);
    IdealGear(4.0, r1, r2);
    serialize(5.0, r2, r3, ShaftElement);
    Inertia(0.3, r3, r4);
    SpeedSensor(r4,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

Higher-order model (higher-order function)

Mechatronic Control Example

Hierarchies of model components.




```
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;
}
```

Unkowns and behaviour equation

Part II
Modelyze Demo

Experimental DSLs

Extensible DSLs for physical modeling

Differential-Algebraic Equations (DAE) → ModelyzeDAE

Acausal connections (Electrical and Mechanical domain) → ModelyzeEOO

HybridCharts (DAE with modes) → ModelyzeHC

EOO + HC = HEOO → ModelyzeHEOO

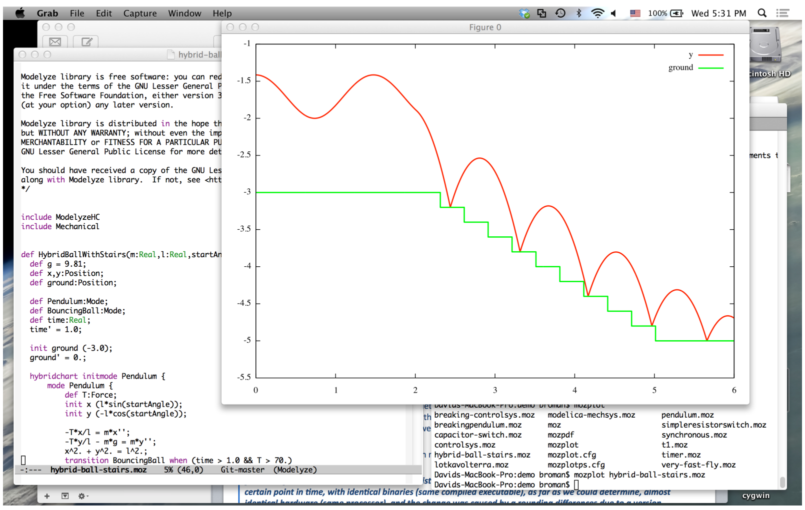
ModelyzeDAE → ModelyzeEOO

ModelyzeDAE → ModelyzeHC

ModelyzeEOO → ModelyzeHEOO

ModelyzeHC → ModelyzeHEOO

Demo…



Part II

Node-Based Connection Semantics

Functional EOO languages

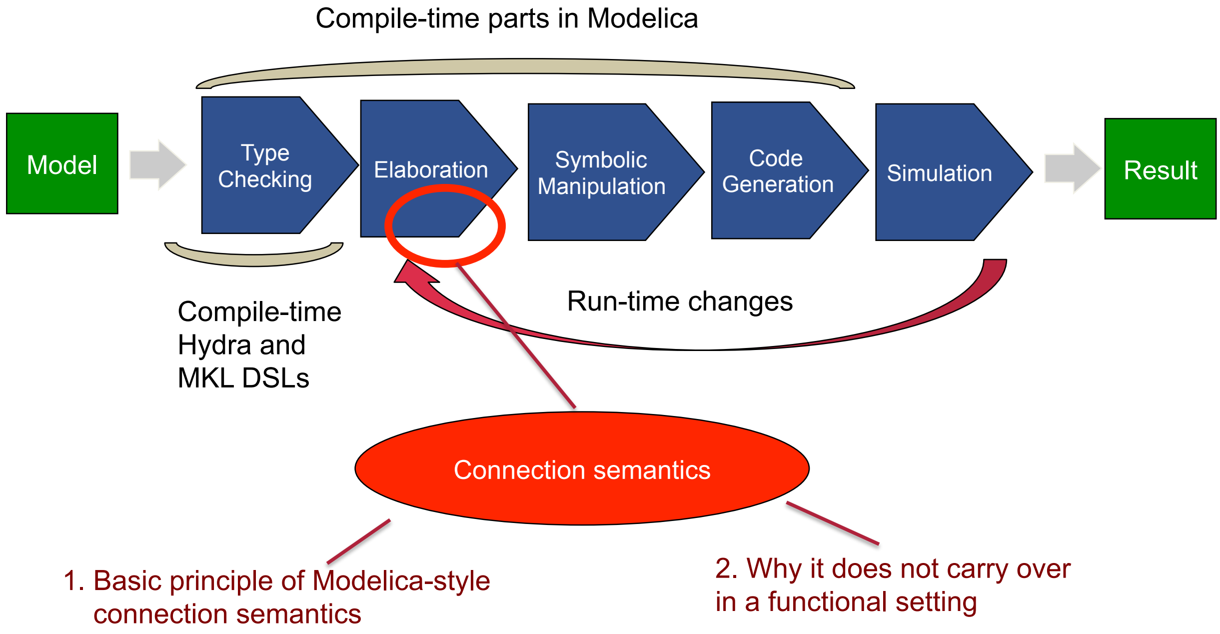
The state-of-the-art EOO language Modelica has a hierarchical structure with a large complex semantics

Functional EOO - why?

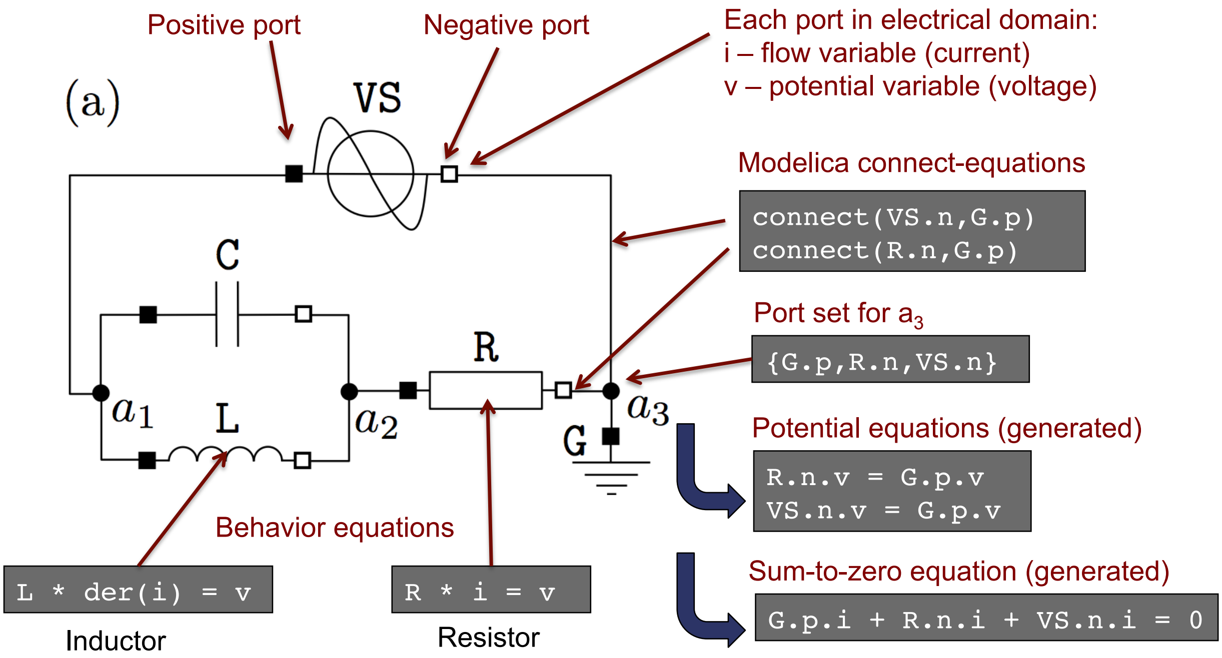
- Utilize expressive power of *higher-order functions* in an EOO setting
- Explore the concepts higher-order acusal models
- Gain a cleaner and simpler semantics rooted in the lambda calculus
  - Formal reasoning
  - Formal specifications
- Explore expressiveness beyond state-of-the-art, e.g., to model *structurally dynamic systems*

David Broman
dbro@kth.se
Part I Modelyze Overview
Part II Modelyze Demoi
Part III Node-Based Connection Semantics
Part IV Formal Semantics

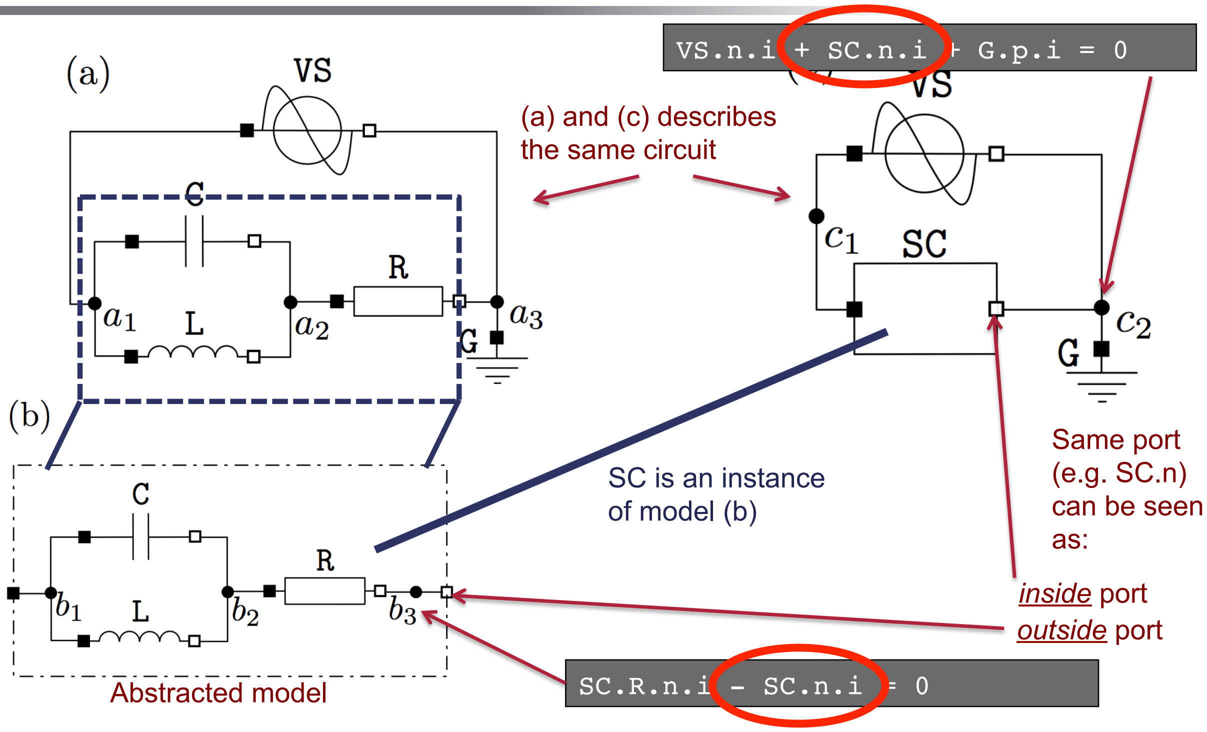
Overview of the Compilation and Simulation Process



Models and Equation Generation



Abstraction and Composition



Why does it not carry over in a functional setting?

Modelica-Style Approach

- Ports contain instance variables (e.g., `v` and `i`)
- Ports are part of the model hierarchy
- Ports can be classified to be *inside* or *outside* with respect to the model hierarchy context

Functional EOO

- Using *functional abstraction* for expressing model abstraction
- Ports becomes formal parameters, that is, ports are no longer direct *parts* of the model
- After functional application (beta-reduction), we have collapsed the hierarchy
- Information about what is inside and outside is no longer available

Approaches in a functional setting

Lava (Bjesse et al., 1998) and Wired (Axelsson et al., 2005)
Functional languages for hardware design, embedded in Haskell

Extend language with special abstraction and application for models

Flow lambda calculus (Broman, 2007)
Extended with a *new model abstraction* and *new model application* for generating equations. Complicated semantics.

FHM/Hydra (Giorgidze and H. Nilsson, 2008)
Connect statements generate sum-to-zero. Special *signal relation application* for generating signs on flow variables.

Our approach: Use lambda abstraction both for function and model abstraction

=> get higher-order models for free

Phase 1

Collapsing the model hierarchy

Directly from the host language (Modelyze)

Phase 2

Connection Semantics

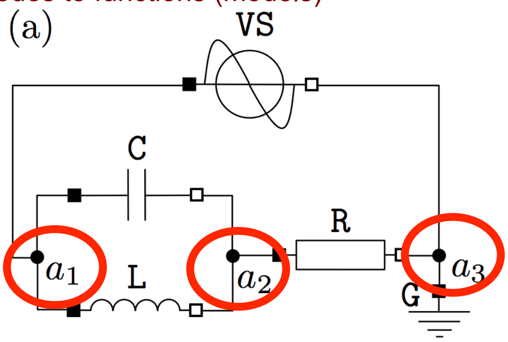
Implemented as a Modelyze library

Phase 1: An circuit model in a Modelyze DSL

Defines nodes of type `Electrical` (fresh typed symbols)

Connections (wiring) by supplying nodes to functions (models)

```
def CircuitA() = {
    def a1,a2,a3:Electrical;
    SineVoltage(220,50,a1,a3);
    Capacitor(0.02,a1,a2);
    Inductor(0.1,a1,a2);
    Resistor(200,a2,a3);
    Ground(a3);
}
```



Components (model instances) are created using function application.

For example, parallel connection (Capacitor and Inductor)

Can also be defined using a higher-order function, e.g.
`parallel(Capacitor(0.02),Inductor(0.1));`

Phase 1: Peeking into a model

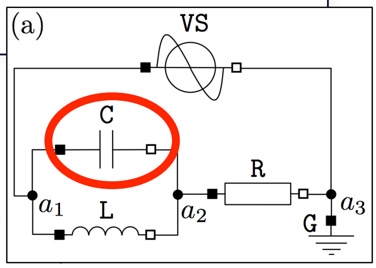
```
def Capacitor(C:Real,p:Electrical,n:Electrical) = {
    def i:Current;
    def v:Voltage;
    Branch(i,v,p,n);
    C * der(v) = i;
}
```

Capacitance

Node parameters

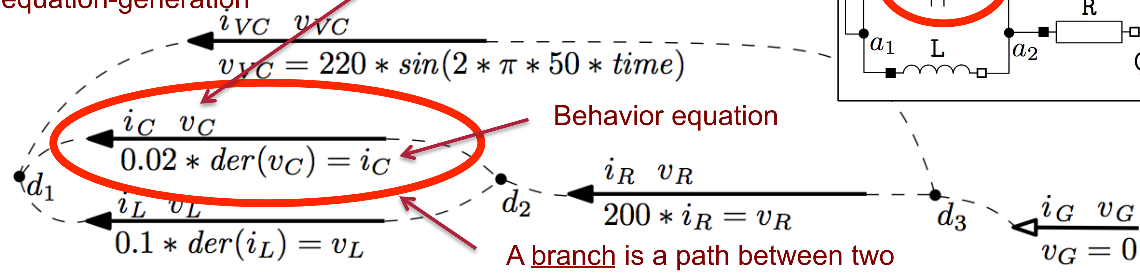
Two variables (signals)

Differential equation (behavior)



Branch: DSL construct that in phase 2 enables correct equation-generation

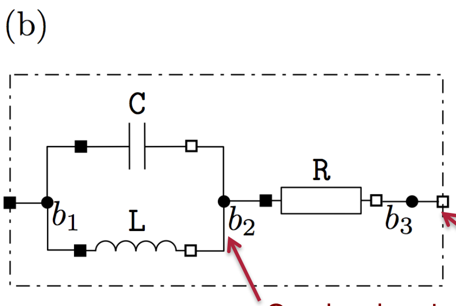
i = current through
v = voltage drop across



Behavior equation

A branch is a path between two nodes through a component

Phase 1: SubCircuit



One local node

Ports of the model

```
def SubCircuit(p:Electrical,n:Electrical) = {
    def b2:Electrical;
    Capacitor(0.02,p,b2);
    Inductor(0.1,p,b2);
    Resistor(200,b2,n);
}
```

Three components

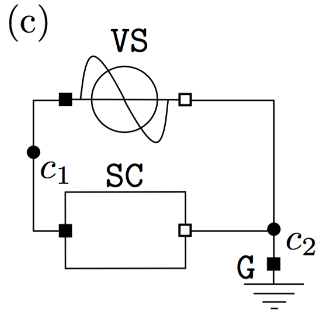
Phase 1: Composition

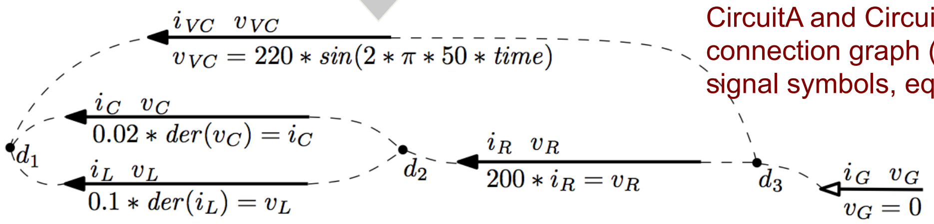
```
type TwoPin = Electrical -> Electrical -> Equations
```

Takes a higher-order model (acausal) as input

```
def CircuitC(SC:TwoPin) = {
    def c1,c2:Electrical;
    SineVoltage(220,50,c1,c2);
    SC(c1,c2);
    Ground(c2);
}
```

Instance of a higher-order model

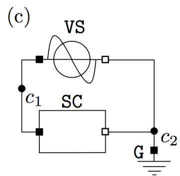




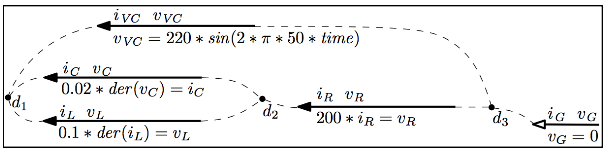
CircuitA and CircuitC result in the same connection graph (containing nodes, signal symbols, equations, and branches)

Our approach

```
def CircuitC(SC:TwoPin) = {
    def c1,c2:Electrical;
    SineVoltage(220,50,c1,c2);
    SC(c1,c2);
    Ground(c2);
}
```



High-level model



Graph-level

$$
\begin{aligned}
J_1\dot{\omega}_1 &= M_v - M_1 \\
J_2\dot{\omega}_2 &= M_h - M_2 \\
\omega_1 &= -r\omega_2 \\
M_1 &= -r^{-1}M_2
\end{aligned}
$$

Differential-Algebraic Equations

Phase 1

Collapsing the model hierarchy

Directly from the host language (MKL)

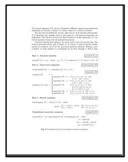
Phase 2

Connection Semantics

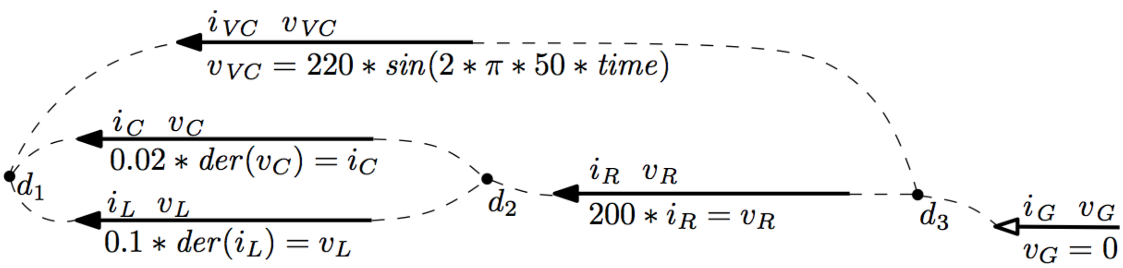
Implemented as a MKL library

Phase 2: Connection Semantics

<u>Input:</u>
All information for handling outside/inside is encoded in the generated graph – regardless how it was constructed



Semantics formalized in the paper

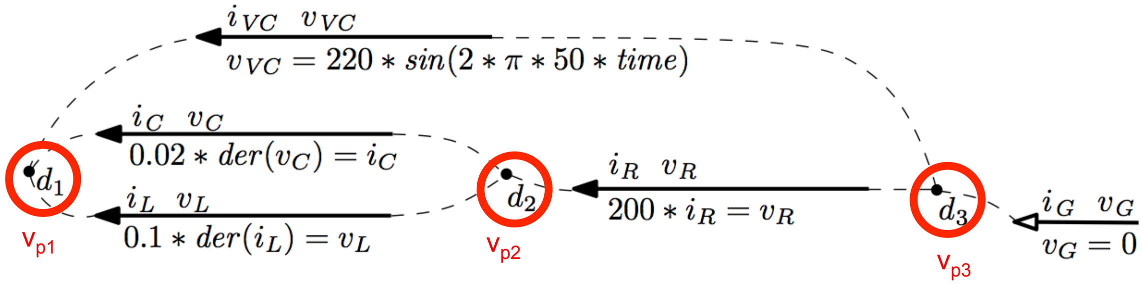


Phase 2: Connection Semantics

3 potential variables

<u>Rule 1</u>
Potential variables

Associate a distinct variable (the potential) with each node



Phase 2: Connection Semantics

3 potential variables

Rule 1
Potential variables

3 sum-to-zero equations

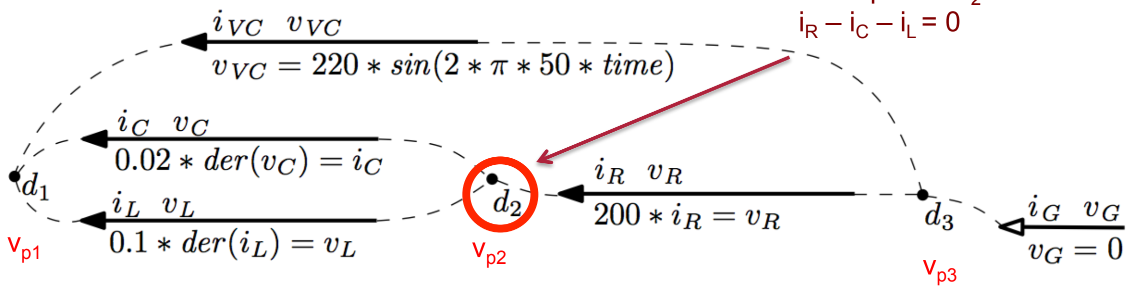
Rule 2
Sum-to-zero equations

For each node, create a sum-to-zero equation

Positive sign if branch points towards the node

Negative sign if branch points away from the node

Example for d_2
$i_R - i_C - i_L = 0$



Phase 2: Connection Semantics

3 potential variables

Rule 1
Potential variables

3 sum-to-zero equations

Rule 2
Sum-to-zero equations

5 branch equations

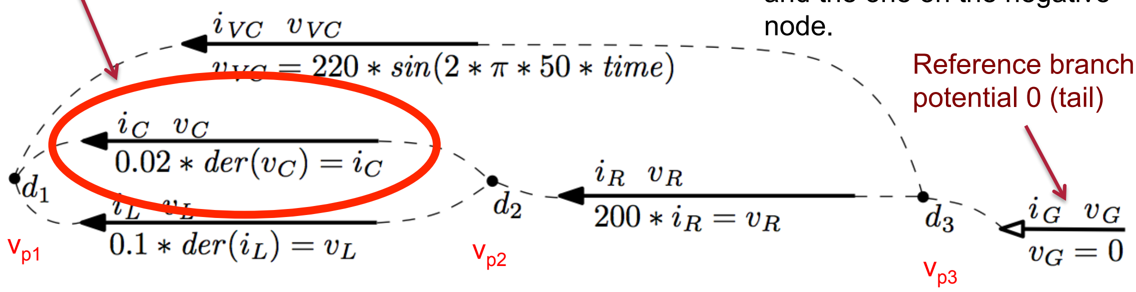
Rule 3
Branch equations

Example, conductor:
$d_1 - d_2 = v_C$

For each branch, create a branch equation

The relative potential is the difference of the potential variable on the positive node and the one on the negative node.

Reference branch, potential 0 (tail)



Phase 2: Connection Semantics

3 potential variables

3 sum-to-zero equations

5 branch equations

Rule 1
Potential variables

Rule 2
Sum-to-zero equations

Rule 3
Branch equations

Explicit in the models
5 relative potential (across) variables and
5 flow variables (through)

13 variables

13 equations

5 behavior equations
(e.g., Ohm's law)

i_{VC} v_{VC}

$v_{VC} = 220 * sin(2 * \pi * 50 * time)$

i_C v_C

$0.02 * der(v_C) = i_C$

d_1

i_L v_L

$0.1 * der(i_L) = v_L$

d_2

i_R v_R

$200 * i_R = v_R$

d_3

i_G v_G

$v_G = 0$

v_{p1} v_{p2} v_{p3}

Clear Invariants

3 potential variables

3 sum-to-zero equations

5 branch equations

Rule 1
Potential variables

Rule 2
Sum-to-zero equations

Rule 3
Branch equations

Explicit in the models
5 relative potential (across) variables and
5 flow variables (through)

Dependent on the
number of nodes
(3 in this case)

5 behavior equations
(e.g., Ohm's law)

i_{VC} v_{VC}

$v_{VC} = 220 * sin(2 * \pi * 50 * time)$

i_C v_C

$0.02 * der(v_C) = i_C$

d_1

i_L v_L

$0.1 * der(i_L) = v_L$

d_2

i_R v_R

$200 * i_R = v_R$

d_3

i_G v_G

$v_G = 0$

v_{p1} v_{p2} v_{p3}

Clear Invariants

3 potential variables

Rule 1
Potential variables

Explicit in the models
5 relative potential (across) variables and
5 flow variables (through)

3 sum-to-zero equations

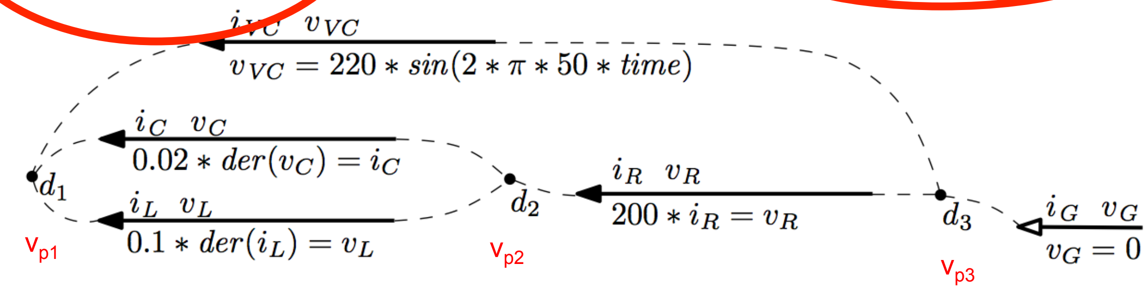
Rule 2
Sum-to-zero equations

Dependent on the number of branches (5 in this case)

5 branch equations

Rule 3
Branch equations

5 behavior equations (e.g., Ohm's law)



Prototype Implementation and Evaluation

Prototype Implementation

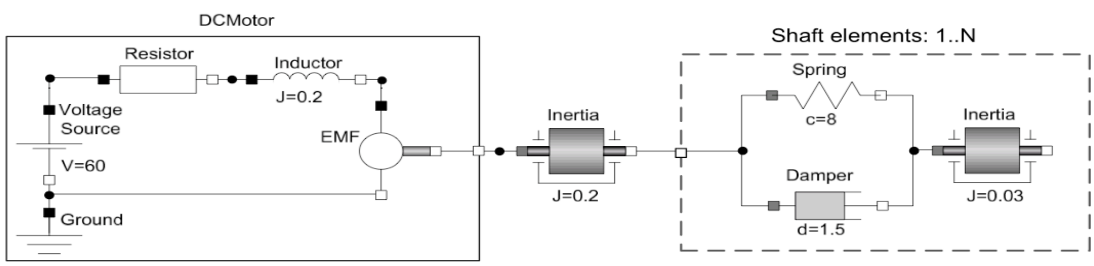
As an embedded DSL in Modelyze (a Modelyze library)

Model libraries in electrical, mechanical, and control domain (also as Modelyze libraries)

Evaluation

Correctness compared to Modelica

Generate flat Modelica (a DAE) and compare simulation results



Part IV
Formal Semantics

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \texttt{<}\tau_{11} \rightarrow \tau_{12}\texttt{>} \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2'' \\ \texttt{<}\tau_{11}\texttt{>} \sim \lceil \tau_2 \rceil\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1' @ e_2'' : \texttt{<}\tau_{12}\texttt{>}} \text{(L-APP5)}$$

Intermediate Languages

To enable formalization and proving type soundness, we define three intermediate languages.

Both are gradually typed
(Mixing static and dynamic typing).

$$\lambda^{\texttt{<?>}} \longrightarrow \lambda_L^{\texttt{<?>}} \longrightarrow \lambda_{LC}^{\texttt{<?>}}$$

Core — Lifted — Cast inserted

Translation step from *surface language* to core language. Includes: parsing, syntatic transformation, pattern compilation etc.

Translating from the core to Lifted language. Includes: *symbolic lifting analysis (SLA).*

Dynamic aspects made explicit by *cast insertion*. Also vital for proving type safety.

Abstract Syntax

Core

$\boxed{\lambda^{<?>}}$

Function and dynamic types.

Symbolic type.

Symbolic data type (e.g., the equation above).

Ground Types	$\gamma \in \mathbb{G}$		
Symbolic Data Types	$D \in \mathbb{D}$		
Types	τ	$::=$	$\gamma \mid \tau \to \tau \mid ? \mid <\tau> \mid D$
Variables	$x, y \in \mathbb{X}$		
Symbols	$s \in \mathbb{S}$		
Constants	$c \in \mathbb{C}$		
Expressions	e	$::=$	$x \mid \lambda x\!:\!\tau.e \mid e\ e \mid c \mid \texttt{error} \mid$
			$\nu(\tau) \mid \texttt{case}(e, p, e, e)$
Patterns	p	$::=$	$\texttt{sym}\!:\!\tau \mid x\,@\,x \mid \texttt{lift}\ x\!:\!\tau$

Standard expressions (var, lambda, application, constant, error).

Case expression for eliminating symbolic data.

Three forms of patterns.

"new" creates a new fresh symbol of type tau.

Note: Recursive patterns compiled away during pattern compilation.

Lifted

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

Extended with symbolic application and lift expression.

Expressions	e	$+=$	$e\,@\,e \mid \texttt{lift}\ e\!:\!\tau$

Consistency Relation

We adopt the idea of gradual typing by Siek and Taha (2006) by replacing type equality by a type consistency relation ~

Dynamic type consistent with all types.

Ground and symbolic data types must be equal.

$$\tau \sim ? \qquad ? \sim \tau \qquad \gamma \sim \gamma \qquad D \sim D$$

$$\frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 \to \tau_2 \sim \tau_3 \to \tau_4} \qquad \frac{\tau_1 \sim \tau_2}{<\tau_1> \sim <\tau_2>}$$

Nested types must match.

Examples

$\texttt{<Int>} \sim \texttt{<?>}$

$\texttt{Int} \not\sim \texttt{<?>}$

$\texttt{<Real>} \to \texttt{Real} \sim \texttt{<?>} \to \texttt{Real}$

Symbolic Lifting Relation

The type system for the core language is defined by a four-place *symbol lifting relation*

$$\boxed{\Gamma \vdash_L e \rightsquigarrow e' : \tau}$$

$\lambda^{<?>}$ Core

$\lambda_L^{<?>}$ Lifted

Type System and Symbolic Lifting for Core

Selected rules (out of 13 rules) $\Gamma \vdash_L e \rightsquigarrow e' : \tau$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e'_1 : \tau_{11} \to \tau_{12} \\ \Gamma \vdash_L e_2 \rightsquigarrow e'_2 : \tau_2 \qquad \tau_{11} \sim \tau_2\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e'_1\ e'_2 : \tau_{12}}\ \text{(L-APP1)}$$

Type equality replaced with type consistency

Argument is of symbolic type

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e'_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash_L e_2 \rightsquigarrow e'_2 : <\tau_2> \qquad \tau_{11} \not\sim <\tau_2> \qquad \tau_{11} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow (\texttt{lift}\ e'_1 : \tau_{11} \to \tau_{12}) @ e'_2 : <\tau_{12}>}\ \text{(L-APP4)}$$

Lift function term

Change to symbolic application

Cast Insertion

Cast insertion defined by a *symbol lifting relation*

$$\boxed{\Gamma \vdash_C e \rightsquigarrow e' : \tau}$$

$\lambda_L^{<?>}$ Lifted

$\lambda_{LC}^{<?>}$ Cast inserted

Type system and Dynamic Semantics for the Lifted (Runtime) Language

Type relation

$$\boxed{\Gamma \vdash e : \tau}$$

$\lambda_{LC}^{<?>}$ Cast inserted

Dynamic semantics (small-step operational)

$$\boxed{e \mid S \longrightarrow e' \mid S'}$$

Set of symbols (computational effect to generate fresh symbols)

Some reduction rules

$$(\lambda x{:}\tau_1.e_1)v_1 \mid S \longrightarrow [x \mapsto v_1]e_1 \mid S \quad \text{(E-BETA)}$$

$$\nu(\tau_1) \mid S \longrightarrow s{:}\tau_1 \mid S \cup \{s\} \quad \text{if } s \notin S \quad \text{(E-NEWSYM)}$$

Creates fresh symbols

Type Soundness

Proposition 3 (Symbolic Lifting Preserves Types)**.** *If* $\Gamma \vdash_L e \rightsquigarrow e' : \tau$ *then* e' *is well typed in* Γ *at type* τ*.*

Proposition 4 (Cast Insertion Preserves Types)**.** *If* $\Gamma \vdash_C e \rightsquigarrow e' : \tau$ *then* $\Gamma \vdash e' : \tau$*.*

Lemma 3 (Progress)**.** *If* $\vdash e : \tau$ *then* $e \in$ *Values, or for all* S *there exists* S' *and* e' *such that* $e \mid S \longrightarrow e' \mid S'$*, or* $e = \texttt{error}$*.*

Lemma 7 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \mid S \longrightarrow e' \mid S'$ *then* $\Gamma \vdash e' : \tau$*.*

Summary and Conclusions

Summary and Conclusions

Some key take away points:

- **Modelyze** is a host language for embedding domain-specific languages (DSLs).
- In particular, it is designed for **embedding equation-based languages.**
- Some of the special **semantic aspects** are: typed symbols, gradual typing, and symbolic lifting analysis.
- **Node-based connection semantics** are especially useful in a functional setting for encoding EOO languages.

Thanks for listening!