# Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages

*David Broman*
*Jeremy G. Siek*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-173
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-173.html

June 30, 2012

Acknowledgement

This research work was funded by the Swedish Research Council (VR), by the ITEA2 OPENPROD project, and by Linkoping University under the ELLIIT project. Additional funding was provided by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley. The second author's work was funded by NSF CAREER Award 0846121.

The authors would like to thank Peter Fritzson, Thomas Schon, Henrik Nilsson, Walid Taha, Sibylle Schupp, and Johan Akesson for comments and suggests on earlier drafts of this work. Our special thanks go to Michael Zimmer for his painstaking effort of reading the final draft.

# Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages

David Broman[a,*], Jeremy G. Siek[b]

[a] *University of California, Berkeley, USA, and Linköping University, Sweden*
[b] *University of Colorado at Boulder, USA*

**Abstract**

Equation-based modeling languages provide an effective means to simulate the physical part of a cyber-physical system. Such languages are complex domain-specific languages that enable model engineers, such as mechanical engineers, to declaratively model the dynamics of systems. However, these modeling languages do not support all modeling needs, which has lead to frequent revisions of state-of-the-art languages. In this article we explore a solution to this extensibility problem based on domain-specific embedded languages. We introduce a host language, named Modelyze, in which modeling languages may be easily embedded. The key features of Modelyze are first-class functions, which provide a mechanism to abstract components of a model, and symbolic expressions, to represent and manipulate equations. The type system for symbolic expressions supports model-level static error checking and provides an automatic lifting translation to provide seamless integration between the host language and the equations represented by symbolic expressions. The type system is based on gradual typing, enabling early static checking for model engineers while providing expressiveness for domain experts. We evaluate this approach by embedding a series of equation-based modeling languages in Modelyze and using them to develop models.

*Keywords:* domain-specific language, typed symbolic expression, gradual typing, cyber-physical system, modeling, simulation

## 1. Introduction

Cyber-physical systems (CPS) [1], such as automobiles, aircraft, and power plants, are expensive to develop because of the complexity and the strong need for safety and correctness. These systems are characterized by combining computation (embedded systems), computer networks, and physical dynamics (the plant). To master the complexity and correctness of these systems, *equation-based modeling languages* have emerged as an effective way to model and simulate systems before creating expensive

* Corresponding author. Tel.: +1 510 460 0280, Address: University of California at Berkeley, EECS Department, 545N Cory Hall, Berkeley CA 94720, USA

*Email addresses:* `broman@eecs.berkeley.edu` (David Broman), `jeremy.siek@colorado.edu` (Jeremy G. Siek)

physical prototypes. Examples of such languages include Modelica® [2, 3], Verilog-AMS [4], and VHDL-AMS [5, 6].

In equation-based modeling languages, the primary construct for describing the continuous-time behavior of the plant is a system of differential equations. For example, the model fragment

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

expresses a simple system of differential-algebraic equations (DAEs) [7] for a pendulum in cartesian coordinates, where variables `x` and `y` are the coordinates for the ball of the pendulum. As usual, an apostrophe signifies differentiation, so `x''` and `y''` are second order derivatives. From the modeler's point of view, one of the main strengths of these languages is that they are declarative, meaning that the system of equations describe *what* the behavior is, but not *how* the equations are solved. Symbolic manipulation [8, 9, 10, 11] and numerical approximation [12] techniques can be used to automatically solve such equation systems efficiently.

Another key characteristic of equation-based modeling languages is support for abstracting over systems of differential equations. Such abstractions can be further composed and connected together in hierarchical structures, which facilitates large scale reuse [13]. We call such a hierarchical abstraction an *equation-based object-oriented* (EOO) model[1]. The term "object-oriented" is used here to describe the direct correspondence between an EOO model and the physical object that it models. (It does not refer to objects as a language feature per se.) For example, an EOO model of a car may comprise other models, such as a gearbox, an engine, transmission shafts, and tires.

Even though several equation-based modeling languages have had considerable success in real-world industrial applications (see e.g., [14, 15]) their expressive power cannot handle all current and future modeling needs. For example, the Modelica language specification has been revised every 1-2 years over the last decade with new language constructs to satisfy the user community's growing modeling needs [3]. However, languages such as Modelica are large and complex. Consequently, extending such a language is very expensive; both from a social point of view (e.g., being involved in design committees and negotiating functionality) as well as technically expensive (e.g., implementing or extending compilers). We refer to this high cost of extending modeling languages as the *extensibility problem*.

As advocated by Steele [16] and recently emphasized by Tobin-Hochstadt *et al.* [17], an attractive alternative to building languages from scratch is to grow the language by pushing syntactic and semantic extensions into libraries. One such approach, pioneered by Hudak [18], is to create *domain-specific embedded languages* (DSELs). In this approach, the underlying host language provides enough syntactic and semantic flexibility to make libraries appear to be *domain-specific languages* (DSLs). DSELs have been successfully deployed in many domains [19, 20, 21, 22, 23],

[1]EOO models have different names in various languages. For example, in Modelica, they are simply referred to as *Modelica models*. In VHDL-AMS, the interface of a model is called an *entity* and its implementation an *architecture*.

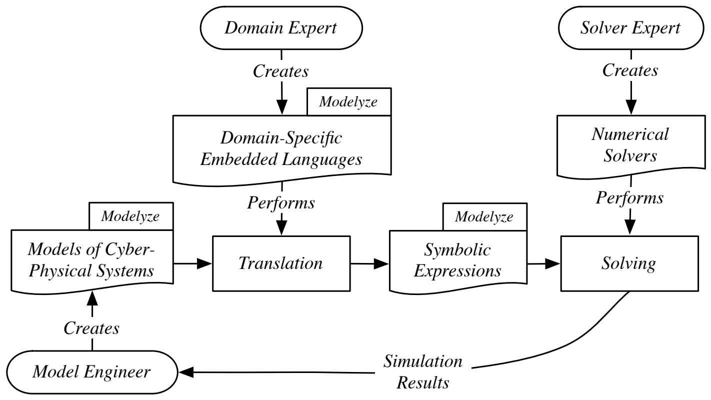

Figure 1: The roles, processes, and artifacts associated with the Modelyze approach to modeling and analyzing cyber-physical systems.

As pointed out by Mernik *et al.* [24], the main benefit of DSELs is the reduction in development effort that comes from borrowing large parts of infrastructure from the host language.

In this paper, we apply the DSEL approach to the modeling of cyber-physical systems. In particular, we develop a new host language, named *Modelyze*[2], that supports the development of modeling languages as DSELs. Figure 1 shows the roles (ovals), processes (rectangles), and artifacts (curvy rectangles) associated with the Modelyze approach to modeling cyber-physical systems. An expert in both the domain and in using Modelyze defines the domain-specific language. A model engineer then uses the domain-specific language to create models of cyber-physical systems. The DSLs are in fact Modelyze libraries that essentially translate the high-level semantics of the DSL into more primitive constructs within Modelyze, which in turn invoke symbolic and numeric solvers to compute the simulation results. The key language features of Modelyze are first-class functions, to enable abstraction over sub-models, and symbolic expressions à la LISP [26] extended with types, to represent equations. This typing of symbolic expressions is reminiscent of the typing of code in MetaML [27]. The type system is based on Siek and Taha's gradual typing [28, 29], extended with symbolic types. Gradual typing provides both static and dynamic typing within a single program. Static typing provides early error feedback to the model engineer. In addition, domain experts can mix static and dynamic typing when analyzing and transforming symbolic expressions, enabling explicit control of the trade-off between expressiveness and static checking.

In the suggested approach, there are several technical challenges/problems. Firstly, even if the basic syntax of Modelyze is chosen to suit model engineers, a quasi-quote

[2]The language Modelyze (MODEl and anaLYZE) is based on MKL, which is described in the first author's Ph.D. thesis [25]. The main extension of Modelyze compared to MKL is a new formal semantics that incorporates gradual typing.

notation (à la LISP [26] or MetaML [30]) for symbolic expressions would destroy the illusion that the DSEL is a domain-specific language. The model engineer wants to see equations, not symbolic expressions. Hence, the first challenge is to provide a *seamless integration* between the host language and symbolic expressions. Secondly, the DSEL developer is typically an advanced domain expert with limited compiler and language background. Language concepts, such as monads [31], type classes [32], and GADTs [33, 34, 35, 36], are powerful constructs for implementing DSELs, but they also have a steep learning curve. The challenge is to provide language mechanisms requiring only minimal training to learn how to pattern match, transform, and analyze symbolic expressions. Thirdly, when the model engineer makes a mistake in using an DSEL, the compiler error messages should be domain-specific and not leak details from the underlying host language (Modelyze). To solve the latter problem, Modelyze uses a type system that tracks precise types for symbolic expressions, inspired by the type system of MetaML. However, to support many different DSELs with symbolic expressions, the types for symbolic expressions must be extensible. Also, in contrast to LISP's macro system [37], C++ Templates [38], Template Haskell [39], and Stratego/XP [40], Modelyze perform transformation at runtime and not at compile time, a necessary criterion to model structurally dynamic systems.

The main novelty in Modelyze from a programming language perspective is the notion of *typed symbolic expressions*. More specifically, to overcome the above challenges, we present the following contributions:

- To provide a seamless integration between Modelyze and DSLs, we introduce a *symbolic lifting analysis* (SLA) (Section 2.2) that is inspired by binding-time analysis from partial evaluation [41].

- To enable pattern matching on symbolic expressions and to provide static error checking with extensible types, we develop a new type system based on *gradual typing* [28, 29]. Section 2.3 shows examples of using pattern matching and Section 2.4 demonstrates DSL-specific error reporting.

- We formalize the core of Modelyze and prove type safety (Section 3).

- We evaluate the expressiveness and extensibility of our approach by defining a series of equation-based DSLs. We implement models from several physical domains in these DSLs (Section 4).

Finally, in Section 5 we discuss related work and in Section 6 we conclude.

## 2. Typed Symbolic Expressions

This section describes and motivates the concept of *typed symbolic expressions*, the main novelty of Modelyze. This section is divided into four parts. First, we introduce a small modeling example, which is then used in the rest of the section. Second, we show how our approach relieves the end user from an annotation burden, by providing a seamless integration between the host language and the DSL. Third, we show how gradually typed functions with pattern matching constructs can be used to analyse and transform symbolic expressions. Forth, we motivate why, and show how, certain errors can be detected at the appropriate DSL level of abstraction.

## *2.1. An Example from the Modeling Engineer's Perspective*

We model a simple two-dimensional mechanical pendulum to illustrate a concrete and simple equation-based model. The model is described in a DSL we developed named M-DAE, which is Modelyze extended with support for modeling differential-algebraic equations. Figure 2 depicts the pendulum and the related simulation plot. The pendulum consists of a massless string of length $l$ together with an attached ball. Angle $\theta$ is the displacement from the equilibrium, force $T$ the tension in the string, and $m$ the mass of the ball.

Assuming no air drag, we model the forces in $x$ and $y$ directions together using Newton's second law of motion ($F = ma$)

$$-T \cdot \frac{x}{l} = m\ddot{x} \tag{1}$$

$$-T \cdot \frac{y}{l} - mg = m\ddot{y} \tag{2}$$

where accelerations in the $x$ and $y$ directions are expressed using second order derivatives $\ddot{x}$ and $\ddot{y}$. The example is given in cartesian coordinates, where the angle $\theta$ is eliminated by replacing expressions $\cos(\theta)$ and $\sin(\theta)$ with $\frac{y}{l}$ and $\frac{x}{l}$ respectively.

The equations based on Newton's law of motion are not enough to model the pendulum. Hence, the equation

$$x^2 + y^2 = l^2 \tag{3}$$

is needed to constrain the ball so that it follows a trajectory in which the string is taut. The initial value positions for $x$ and $y$ are defined by the start angle $\theta_s$ that specifies the initial displacement from equilibrium. The initial equations are

$$x(0) = l\sin(\theta_s) \tag{4}$$

$$y(0) = -l\cos(\theta_s) \tag{5}$$

The above mathematical model (differential equations together with initial conditions) is expressed in M-DAE as follows:

```
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;
}
```

The `Pendulum` model is defined using a function abstraction. Supplying concrete arguments to the pendulum, for the mass of the ball, length of the string, and initial angle,

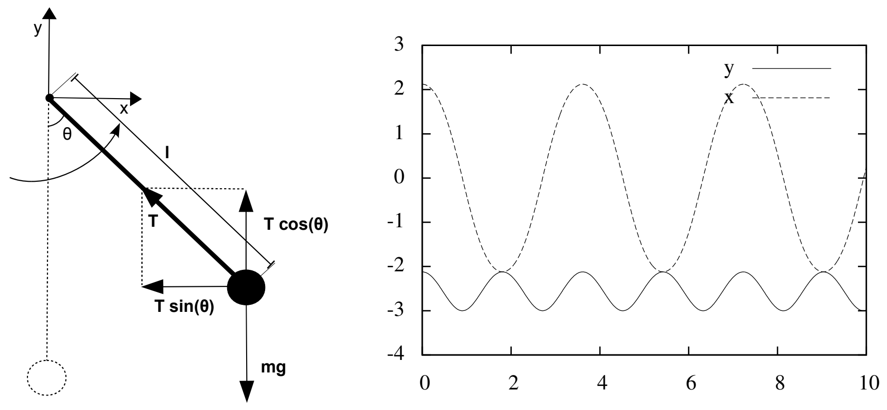

Figure 2: (a) Diagram of a simple pendulum. (b) Plot of the simulated pendulum.

creates an instance of the model. For example, expression `Pendulum(5,3,45*pi/180)` represents a mathematical model instance with the mass 5kg, string length 2m, and a start angle of $45$ degrees. Variable `pi` is bound outside the function to an approximation of $\pi$.

Line two in the code listing defines the new unknowns `x`, `y`, and `T`. We use the term *unknown* to describe a variable in an equation system. Internally, in the host language, these unknowns are represented as *typed symbols*. For example, three fresh symbols of symbolic type `Real` are created when line two is evaluated. As usual, we use the term *variable* for functional variables that can only be bound to a value once.

From a modeling point of view, the rest should be self explanatory. We pause to note the direct correspondence between the equations (1)-(3), the initial equations (4)-(5), and the Modelyze code (lines 3-8).

## 2.2. *Seamless Integration - Removing End User Annotation Burden*

In the `Pendulum` example, it is not obvious which parts of the syntax are from the host language and which are from the DSEL. This is intentional and is what we call *seamless integration* between the host language and the DSEL. In the `Pendulum` example, lines 1-2 are part of the host language, whereas lines 3-8 are defined by the DSL. Equations, derivatives, and initial values are not part of Modelyze, whereas function abstraction (line 1) and symbol creation (line 2) are part of the host language.

The notion of symbolic expression is an old concept, introduced in LISP by McCarty as *S-expressions* (symbolic expressions). *Quasi-quoting* is a classic way of mixing symbolic expressions with program code. For example, in Common Lisp [26], a quasi-quoted expression `‘(+ 1 ,a)` means that the expression should be treated as data (the quotation indicated by `‘`) together with an *unquote* (or anti-quote) `,a` forming a template so that variable `a` can be substituted at runtime. Other languages support quasi-quoting with different notation. For example, in MetaML [30], angle brackets (< >) are used for quotation and tilde (˜) is used for anti-quoting.

However, one problem with quasi-quoting is that it adds an extra level of *annotation burden* on the engineer to carefully add quotes at selected places in a program fragment. For example, if code line 8 of the `Pendulum` example is written using MetaML's quasi-quote notation, the resulting code line is

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

Note how the end-user must carefully consider the different sub-expressions. For example, on the right hand side of the equation, an extra lambda abstraction needs to be inserted so that expression `l^2` is directly computed to a value (variable $l$ is known when arguments to function `Pendulum` are supplied).

To relieve the end user from such annotation burden, the quotation of symbolic expressions is performed implicitly by the Modelyze compiler. We call this new process *symbolic lifting analysis* (SLA). In contrast to *binding time analysis* (BTA) [41] in partial evaluation [42], SLA determines which expressions *cannot* be evaluated at runtime, thus lifting these expressions into symbolic data structures. The SLA uses types to distinguish which expressions that should be lifted. This is the first motivation for why symbols are typed in Modelyze.

**Example 2.1** (Symbolic Lifting)**.** Consider again the `Pendulum` example, where three typed symbols are created on line 2. Each such symbol has a unique identifier and an associated (tagged) type. Similar to MetaML's notation of code types, our symbol types are expressed using enclosing angle brackets. For example, the type of a symbolic integer is `<Int>` and the type of a symbolic real is `<Real>`. Hence, in the example, variables `x`, `y`, and `T` are of type `<Real>`. Syntactically, typed symbols are created using the syntax

$$\texttt{def}\, x\texttt{:}T\texttt{;}e \tag{6}$$

which means that a new fresh symbol is created and tagged with type $T$, and then substituted for all free occurences of $x$ in $e$. Note that `x` itself is not the symbol, but a fresh symbol is substituted for `x`. This means that there can be many more symbols in an executing program than static occurences of `def`, which is a prerequisite for defining large reusable models.

Let us zoom in on sub-expression `x/l` of the following equation

```
-T*x/l = m*x'';
```

on line 6 of the example. If we rewrite the expression in prefix curried form, we have `((/ x) l)`, where `/:Real->Real->Real`, `x:<Real>`, and `l:Real`. Clearly, this expression does not type check, because the parameters of the division operator are of type `Real`, but the first argument `x` is of the symbolic type `<Real>`. This is where symbolic lifting takes place. Because the division cannot be performed at runtime, the division operator is lifted to the symbolic type `<Real->Real->Real>`. Moreover, because the lifted version of the division operator now is of a symbolic type, the length `l` is also lifted to type `<Real>`. After lifting the separate parts, the expression `x/l` type checks and is of type `<Real>`.

To conclude this subsection, we have given some intuition regarding what happens during type checking and the symbolic lifting. The full details of the type system, including symbolic lifting and a proof of type soundness, are presented in Section 3.

## 2.3. Pattern Matching on Open Gradually Typed Symbolic Expressions

The result of evaluating an instance of a model, such as the `Pendulum` example, is a data structure consisting of symbolic expressions. To enable symbolic computation via transforming symbolic expressions, Modelyze provides pattern matching. In this section, we show an example of how to traverse a typed symbolic expression. We also motivate why the data constructors for symbolic expressions should be *open* as well as why *gradual typing* is useful in this context.

**Example 2.2** (Symbolic Data Type)**.** In the `Pendulum` example in Section 2.1, we stated explicit types for the function parameters, but not for the return type. If the function definition is non-recursive, the return type can be left out. If the return type is given in the Pendulum case, the definition is

```
def Pendulum(m:Real,l:Real,angle:Real) -> Equations = {
```

The type of the `Pendulum` is

```
Real -> Real -> Real -> Equations
```

meaning that the function takes three arguments and returns a value of type `Equations`. Note that functions are always curried in Modelyze, even though function definitions and applications can be syntactically expressed using parenthesis[3]. The return type `Equations` is a *symbolic data type*, defined as

```
type Equations
```

Expressions of a symbolic data type can be constructed by creating symbols of the type. In the pendulum example, equations of the `Equations` type are constructed using the infix equality symbol =, defined as follows:

```
def (=) : Real -> Real -> Equations
```

The notation of enclosing the operator in parenthesis means that it is an infix operator. In contrast to the symbols created in the `Pendulum` model, the symbol for = is a global symbol. The type of = is <`Real -> Real ->` `Equations`>, where the enclosing brackets have been implicitly added by the compiler, enforcing the invariant that the type of a symbol is always a symbolic type.

**Example 2.3** (Generic Traversal and Pattern Matching)**.** Assume that the following definitions, for creating equations, are defined in a DSL library called `equations.moz`:

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

Another library defines functions for solving linear algebraic equations. A fundamental function in the latter library, shown in Figure 3, collects all the unknowns of an

[3]In the current version it is possible to mix traditional functional syntax without parenthesis (similar to Haskell and OCaml) and syntax with parenthesis (similar to Java and C). Our design objective is to keep the useful functionality of currying and at the same time provide a syntax familiar to non functional programmers from the engineering communities.

```
1 def getUnknowns(exp:<?>, acc:(Set <Real>)) -> (Set <Real>) = {
2     match exp with
3     | e1 e2 -> getUnknowns(e2,getUnknowns(e1,acc))
4     | sym:Real -> Set.add exp acc
5     | _ -> acc
6 }
```

Figure 3: Example of a function that pattern matches over symbolic data.

equation system. This function recursively traverses a symbolic expression representing an equation system and returns all the typed symbols of type `Real`, representing unknowns. The function takes two parameters `exp` (the symbolic expression) and `acc` (an accumulator for a set of symbols) of types <?> and `Set <Real>`, respectively. The first parameter uses the *dynamic* type `?`, meaning that `exp` can be of any symbolic type.

The pattern matching construct `match` deconstructs symbolic expressions. For example, line 3 of Figure 3 matches a *symbolic application* and line 4 matches a symbol that is tagged with type `Real`. If it does not match any of the symbolic expressions (line 5), the accumulator is returned. Note how the dynamic symbolic type <?> enables the expression of generic traversals over symbolic expressions. This is an example where gradual typing is used to improve expressiveness by using dynamic checking for fragments of the program.

**Example 2.4** (Open Data Types)**.** Assume we develop a new DSL that can handle differential-algebraic equations. The syntactic extensions for expressing initial values and derivatives are described in a separate library:

```
include Equations
def der : Real -> Real
def (')  = der
def init : Real -> Real -> Equations
```

The first line shows simple file inclusion of the library `equations.moz`, defined in the previous example. Note that the symbolic data type is necessarily *open*, meaning that we can add new symbols later in the program (in separate libraries), and then use both the old and new symbols together in the same expression. In the above DAE extension, we first define the constructor `der` for representing derivatives that has the symbolic type <`Real->Real`>. Given an unknown `x` of type <`Real`>, the expression `der(x)` of type <`Real`> represents the derivative of `x`. We also define a postfix symbolic function `'` for representing derivatives.

The functionality of returning all unknowns of an equation system is still useful for a DAE. Because the static definitions of the DAE are based on the same definitions as for the linear equation system, we can still apply function `getUnknowns` on a DAE. For example, expression

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

returns, as expected, a set of three symbols.

Assume we need a function that returns the mapping between all unknowns (symbols) and their symbolic initial values. To collect this mapping, we use a built-in associative container type that maps keys to values, that is, $T_1$ => $T_2$ is a type that maps keys of type $T_1$ to values of type $T_2$. We define a type alias `InitVals` that maps from unknowns (symbols of type `Real`) to symbolic initial values.

```
type InitVals = <Real> => <Real>
```

The expression `Map.empty` creates an empty map container, whereas `Map.add k v m` creates a new container that is the same as `m` except that it maps `k` to `v`.

A generic function that traverses an equation system and collects this mapping can then be defined as follows:

```
def getInitValues(eqs:Equations, acc:InitVals) -> InitVals = {
    match eqs with
    | e1 ; e2 -> getInitValues(e2, getInitValues(e1,acc))
    | init x v -> Map.add x v acc
    | _ -> acc
}
```

By applying the `Pendulum` instance to `getInitValues`, we get the map

```
{x => 2.121271206, y => -2.12136947998}
```

Note that because the expressions `l*sin(angle)` and `-l*cos(angle)` only contain known values (no symbols), symbol lifting analysis did not lift these expressions.

In this subsection we showed how to pattern match and analyze symbolic expressions. Also, we briefly illustrated how open types make it possible to reuse DSL definitions. We showed how generic traversals can be achieved using dynamic types. However, the expressiveness of gradual typing also introduces limitations of static guarantees. For example, static type safety is not guaranteed during symbolic manipulation (but runtime type safety is guaranteed). The trade offs between expressiveness and the limitation of static guarantees will be discussed further in the evaluation in Section 4.

### 2.4. *Static Error Checking at the DSL Level*

When a model engineer makes mistakes when creating models in a specific DSL, it is important that the error messages directly reflect the abstraction level of that DSL. That is, the aim is to provide error message directly at the DSL level, pointing out the line of the model where the fault is located and mentioning types specific to the DSL. In this section we motivate and describe how this can be partially accomplished by performing static type checking of the typed symbolic expressions.

**Example 2.5** (Static Error Checking)**.** Consider again the `Pendulum` example but with an errors:

```
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;
}
```

Syntactically, this model is correct, i.e., neither the lexer nor the parser complains about the model. However, the inserted error prevents the model from being simulated. If there was no static type checking, the failure caused by this error would not have been detected until very late in the simulation process. The missing initial value would make the numerical solver to fail when trying to initialize the equation system. In such a case, the model engineer does not get any information of *where* in the actual model code the error is located.

However, by performing static type checking at the DSL level directly on the typed symbols, the user may receive error messages with significantly better fault localization. For example, the current Modelyze type checker reports the following error message for the example model with the missing initial value:

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

We should point out that this static type checking only rules out some of the potential errors that a user can make. Incorrectly specified equation systems that are either over or under-constrained are not detected. Improving such error detection involves further error detection mechanisms [43, 44, 45].

To summarize, typed symbolic expressions can be used in a host language to relieve the user from the quasi-quoting annotation burden, enable expressive transformation and pattern matching on symbolic expressions, and to provide good static error reporting at the DSL level. Section 4 further evaluates the strength and weaknesses of the suggested approach.

## 3. Semantics of Modelyze

Modelyze consists of a surface language and a core language. The translation steps from the surface to the core language are standard and include parsing, syntactic transformations, and pattern compilation [46]. In this section we present and formalize an essential subset of the core that includes typed symbolic expressions, symbol types, gradual typing, and symbol lifting analysis. Modelyze also includes other language constructs, such as lists, sets, maps, conditional expressions, and print statements, which are not essential for describing the main contributions of the language.

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

| | | | |
|---|---|---|---|
| Ground Types | $\gamma \in \mathbb{G}$ | | |
| Symbolic Data Types | $D \in \mathbb{D}$ | | |
| Types | $\tau$ | $::=$ | $\gamma \mid \tau \rightarrow \tau \mid \texttt{?} \mid \texttt{<}\tau\texttt{>} \mid D$ |
| 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 | $p$ | $::=$ | $\texttt{sym}\!:\!\tau \mid x \,@\, x \mid \texttt{lift}\ x\!:\!\tau$ |

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

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

Figure 4: Abstract syntax of $\lambda^{<?>}$ and $\lambda^{<?>}_{L}$.

To formalize the static and dynamic semantics, as well proving soundness property of the language, we present three different intermediate languages. The language $\lambda^{<?>}$ is the source language corresponding to the essential core of Modelyze. We define a translation from $\lambda^{<?>}$ to an intermediate language $\lambda^{<?>}_{L}$ that lifts selected expressions into symbolic expressions (Section 3.3). The reason for symbolic lifting is, as discussed in the previous chapter, to create data structures of equations that can later be inspected and analyzed. Both $\lambda^{<?>}$ and $\lambda^{<?>}_{L}$ are gradually typed languages, that is, they mix static and dynamic typing. The dynamic aspect is made explicit through a cast insertion translation into another intermediate language $\lambda^{<?>}_{LC}$ (Section 3.4). We present an operational semantics for $\lambda^{<?>}_{LC}$ and prove that the translations between the intermediate languages are type preserving. We prove the usual progress and preservation lemmas for $\lambda^{<?>}_{LC}$, from which we obtain type safety for $\lambda^{<?>}$ (Section 3.5).

## *3.1. Syntax*

The abstract syntax for $\lambda^{<?>}$ is defined in Figure. 4. The meta-variables $x$ and $y$ range over variables, taken from some countable set of names $\mathbb{X}$. The meta-variable $e$ ranges over the set of expressions $\mathit{Expr}$ and $\tau$ ranges over the set of types $\mathit{Types}$. We use subscripts to create different meta-variables, e.g., $e_1$ and $e_2$ are two different metavariables that range over expressions.

The first five expressions are standard, but to review, the expression $x$ is a free variable and lambda abstraction $\lambda x{:}\tau.e$ binds variable $x$ of type $\tau$ in $e$ and delays the execution of $e$ until the abstraction is applied to an argument. The expression $e_1\ e_2$ is an application and $c \in \mathbb{C}$ is a constant. The set of constants $\mathbb{C}$ is the union of the set of boolean values $\{\texttt{true}, \texttt{false}\}$, the set of integers, the set of floating-point values, the set of strings, and the set of primitive functions. The expression `error` is a simple form of exception used to signal cast and pattern match errors.

There are two new kinds of expressions in $\lambda^{<?>}$. Expression $\nu(\tau)$ (pronounced "new") creates a fresh symbol with type $\tau$. The expression $\texttt{case}(e, p, e_t, e_f)$ eliminates symbolic data. The value of $e$ is matched against the pattern $p$. In the core language, the patterns are non-recursive. Nested patterns within `match` constructs are compiled into `case` expressions in the core language. The value of $e_t$ is returned on a successful match and the value of $e_f$ is return on a unsuccessful match. Patterns can have three different shapes: $\texttt{sym} : \tau$ for symbols, $x \,@\, x$ for matching symbolic applications, and $\texttt{lift}\ x : \tau$ for values that have been lifted to become symbolic data. In the `lift` pattern form, the variable $x$ is a pattern variable and $\tau$ a type tag.

There are three standard types and two new types for this language. The meta-variable $\gamma$ ranges over all ground types $\mathbb{G}$ (e.g., booleans and integers), types of the form $\tau \rightarrow \tau$ are function types, and `?` is the dynamic type [28]. To categorize symbolic data of type $\tau$, we introduce the type $<\tau>$. Also, $D$ ranges over primitive symbolic data types. There is a finite set of such types in a program. An example of a $D$ type is the type `Equations` discussed in Section 2.3.

Modelyze's syntax for creating a symbol is defined by the following derived form

$$\texttt{def}\ x \texttt{:} \tau \texttt{;}\ e \equiv (\lambda x{:}<\tau>.e)\nu(\tau) \tag{7}$$

The symbolic lifting analysis translates from $\lambda^{<?>}$ to $\lambda_L^{<?>}$, making explicit whether an application is really a symbolic application and whether a value should be treated as just a value, or as a symoblic value. Thus, $\lambda_L^{<?>}$ has two new expressions: The first expression $e \,@\, e$ is called a *symbolic application*, which is typed as a function application, but is never applied. From a runtime perspective, a symbolic application can be seen as a tuple with two elements. The second new expression $\texttt{lift}\ e : \tau$ is a *lift expression* that injects the value of $e$ into the symoblic type $<\tau>$.

**Example 3.1** (Symbolic Lifting)**.** The following example shows a simple translation from the surface language to the core language, symbolic lifting analysis, and how the lifted expression is evaluated to a value. We assume the symbolic data type `Equation` and the symbol for equations `(=)`

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

are defined in the environment. Then the expression

```
def x:Real;
x+5 = 2*3
```

in the surface language is translated into

$$(\lambda x : <\texttt{Real}>.\texttt{(=)}(\texttt{(+)}\ x\ 5)(\texttt{(*)}\ 2\ 3))\ \nu(\texttt{Real})$$

in the core language. The symbol lifting analysis (formalized in Section 3.3) lifts the expressions into symbolic expressions, so that the new expression is well typed. For example, `(+)` is of type $\texttt{Real}\rightarrow\texttt{Real}\rightarrow\texttt{Real}$, but $x$ is of type $<\texttt{Real}>$. The SLA phase is then lifting `(+)` to $\texttt{lift (+)} : \texttt{Real}\rightarrow\texttt{Real}\rightarrow\texttt{Real}$ that now has symbolic type $<\texttt{Real}\rightarrow\texttt{Real}\rightarrow\texttt{Real}>$. The result after applying SLA on subterm $\texttt{(+)}\ x\ 5$ is

$$(\lambda x : <\texttt{Real}>.\texttt{(=)}(\texttt{lift (+)} : T_1 @\, x @\, (\texttt{lift}\ 5 : \texttt{Real}))(\texttt{(*)}\ 2\ 3))\ \nu(\texttt{Real})$$

where $T_1 = \texttt{Real}\to\texttt{Real}\to\texttt{Real}$. Note that also the real literal 5 is lifted to type `<Real>` and that function applications have been replaced by symbol applications ($e@\,e$). The expression `(*)` 2 3 does not contain any symbolic types, so its subterms should not be lifted. However, the resulting value of type `Real` is the second argument to symbol `(=)`, which is of symbol type. Hence, the whole expression needs to be lifted, resulting in the final lifted expression

$$(\lambda x\!:\!\texttt{<Real>}.\\ \quad \texttt{(=)}@\,(\texttt{lift (+)}\!:\!T_1@\,x@\,(\texttt{lift}\ 5\!:\!\texttt{Real}))@\,(\texttt{lift}\ (\texttt{(*)}\ 2\ 3)\!:\!\texttt{Real})\\ )\nu(\texttt{Real})$$

The expression is now well typed and can be evaluated. Expression $\nu(\texttt{Real})$ is reduced to the symbolic expression $s_1 : \texttt{Real}$ where $s_1$ is a new fresh (unique) symbol. After beta reduction of the outer most application and reduction of expression `(*)` 2 3, expression

$$\texttt{(=)}@\,(\texttt{lift (+)}\!:\!T_1@\,(s_1\!:\!\texttt{Real})@\,(\texttt{lift}\ 5\!:\!\texttt{Real}))@\,(\texttt{lift}\ 6\!:\!\texttt{Real}))$$

is the final value of type `Equations`.

**Example 3.2** (Pattern Compilation and Case Analysis)**.** Assuming that a symbol `der` is defined at the top-level scope

```
def der : Real -> Real
```

such that a fresh symbol $s_{der}$ : `Real->Real` exists in the environment, the match expression

```
match e with
| der x -> x
| sym:Real -> e
```

in the surface language is translated to the expression

$$\texttt{case}(e, y_1\,@\,x,\\ \quad \texttt{if}\ y_1 = (s_{der}\!:\!\texttt{Real->Real})\ \texttt{then}\ x\ \texttt{else}\ \texttt{error}\ ,\\ \quad \texttt{case}(e, \texttt{sym}\!:\!\texttt{Real}, e, \texttt{error}\ ))$$

in the core language (extended with an equality operator and `if`-expressions). Variable $y_1$ is a temporary variable created by the pattern compilation algorithm. Note that the front end translation step keeps track of symbol definitions defined at the top-level scope to differentiate between pattern variables (`x` in this case) and symbols that should be compared for equality (for example `der`).

## *3.2. Gradual typing*

To provide gradual typing, we adopt the idea of replacing type equality in the type checking rules with the type consistency relation $\sim$ [28, 29]. The definition of type consistency is given in Figure 5. To review, the type consistency relation was created to resolve problems that were encountered in prior attempts to integrate static and dynamic typing [47]. These prior attempts were based on subtyping, and placed the dynamic type ? at both the top and the bottom of the subtype hierarchy, to allow both implicit casts from any type to ? and from ? to any other type. The problem with this approach is that subtyping is transitive, and thus, the hierarchy collapses and every type becomes a subtype of every other type. The idea of the type consistency relation is to have a separate relation from subtyping that handles the implicit casts to and from ?, and that is not transitive.

**Example 3.3** (Consistency Relation)**.**
Example of consistency relations that do and do not hold.

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

$$\texttt{<?>} \sim \texttt{<Bool} \to \texttt{Real>} \qquad \texttt{<Real>} \to \texttt{Real} \sim \texttt{<?>} \to \texttt{Real} \qquad \texttt{Int} \sim \texttt{?}$$

$$\texttt{Int} \not\sim \texttt{<?>} \qquad \texttt{Real} \to \texttt{<?>} \not\sim \texttt{<?>} \to \texttt{Real} \qquad \texttt{Real} \not\sim \texttt{Bool}$$

The consistency relation is closely related to two other relations, naive subtyping and a meet operator. We employ the meet operator in the cast insertion relation and the naive subtyping relation is helpful in understanding the meet operator. In particular,

$$\boxed{\tau <:_n \tau}$$

$$\tau <:_n ? \qquad \gamma <:_n \gamma \qquad D <:_n D$$

$$\frac{\tau_1 <:_n \tau_3 \qquad \tau_2 <:_n \tau_4}{\tau_1 \to \tau_2 <:_n \tau_3 \to \tau_4} \qquad \frac{\tau_1 <:_n \tau_2}{\texttt{<}\tau_1\texttt{>} <:_n \texttt{<}\tau_2\texttt{>}}$$

$$\boxed{\tau \sim \tau}$$

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

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

$$\boxed{\tau \sqcap \tau}$$

$$\begin{gathered}
\tau \sqcap ? = \tau \\
? \sqcap \tau = \tau \\
\gamma \sqcap \gamma = \gamma \\
D \sqcap D = D \\
(\tau_1 \to \tau_2) \sqcap (\tau_3 \to \tau_4) = (\tau_1 \sqcap \tau_3) \to (\tau_2 \sqcap \tau_4) \\
\texttt{<}\tau_1\texttt{>} \sqcap \texttt{<}\tau_2\texttt{>} = \texttt{<}\tau_1 \sqcap \tau_2\texttt{>}
\end{gathered}$$

Figure 5: Naive Subtyping, Consistency, and Meet.

the meet operator computes the greatest lower bound (if it exists) with respect to naive subtyping. The consistency relation holds between two types exactly when there exists a greatest lower bound of the two types. So one could define the consistency relation in terms of the meet operator, and vice versa.

**Example 3.4** (Relation between Naive Substyping, Consistency, and Meet)**.**
The naive subtyping relation forms the following partial order for the types ?, <?>, ? $\to$ ?, `Real`, <`Real`>, <`Real`> $\to$ ?, ? $\to$ `Real`, and <`Real`> $\to$ `Real`:

$$\begin{tikzcd}
 & & \texttt{?} \arrow[dl, dash] \arrow[dr, dash] \arrow[ddr, dash] & & \\
 & \texttt{?} \to \texttt{?} \arrow[dl, dash] \arrow[dr, dash] & & \texttt{<?>} \arrow[dr, dash] & \\
\texttt{<Real>} \to \texttt{?} \arrow[dr, dash] & & \texttt{?} \to \texttt{Real} \arrow[dl, dash] & \texttt{Real} & \texttt{<Real>} \\
 & \texttt{<Real>} \to \texttt{Real} & & &
\end{tikzcd}$$

The top element ? is the supertype of all other types, but for brevity, we omit some of these lines in the figure. The following table shows examples of how the consistency and meet relations relates to the partial order.

| Consistency | Meet |
|---|---|
| `<Real>` $\to$ ? $\sim$ ? $\to$ `Real` | `<Real>` $\to$ ? $\sqcap$ ? $\to$ `Real` $=$ `<Real>` $\to$ `Real` |
| `<Real>` $\to$ ? $\sim$ ? | `<Real>` $\to$ ? $\sqcap$ ? $=$ `<Real>` $\to$ ? |
| <?> $\sim$ `<Real>` | <?> $\sqcap$ `<Real>` $=$ `<Real>` |
| `<Real>` $\not\sim$ `Real` | `<Real>` $\sqcap$ `Real` $=$ *undefined* |
| `Real` $\not\sim$ ? $\to$ ? | `Real` $\sqcap$ ? $\to$ ? $=$ *undefined* |

**Proposition 1.**

1. *If* $\tau_1 <:_n \tau_2$*, then* $\tau_1 \sim \tau_2$.
2. *The meet of two types is consistent with those two types. That is, if* $\tau_3 = \tau_1 \sqcap \tau_2$*, then* $\tau_3 \sim \tau_1$ *and* $\tau_3 \sim \tau_2$.

*Proof.*

1. The proof a straightforward induction on the derivation of $\tau_1 <:_n \tau_2$.
2. Because $\tau_3$ is a lower bound of $\tau_1$ and $\tau_2$, we have $\tau_1 <:_n \tau_3$ and $\tau_2 <:_n \tau_3$. Then by part 1 of this proposition and the symmetry of $\sim$, we have $\tau_3 \sim \tau_1$ and $\tau_3 \sim \tau_2$.

□

## *3.3. Type system and symbolic lifting analysis*

As usual, expressions are assigned types in the context of a *typing environment*, which is a partial function from variables to types. Sometimes we use set notation when dealing with a type environment, e.g., $x : \tau \in \Gamma$ is equivalent to $\Gamma(x) = \tau$. We use the notation $\Gamma, x{:}\tau$ to extend environment $\Gamma$ with a new binding $x{:}\tau$. If a binding of $x$ exists in $\Gamma$, the new binding replaces the old one. We define the subset relation between typing environments as follows.

**Definition 1.** $\Gamma \subseteq \Gamma' \equiv \forall x\tau.\ \Gamma(x) = \tau$ *implies* $\Gamma'(x) = \tau$.

The type system for $\lambda^{\text{<?>}}$ is defined by a four-place *symbolic lifting relation*

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

where $e$ is an expression in $\lambda^{\text{<?>}}$, $e'$ an expression in $\lambda_L^{\text{<?>}}$, $\tau$ the type of the resulting value, and $\Gamma$ the typing environment. The symbolic lifting relation is inductively defined by the inference rules in Figure 6, which we discuss shortly.

**Definition 2** (Well-typed expression in $\lambda^{\text{<?>}}$)**.** *An expression $e$ of $\lambda^{\text{<?>}}$ is well typed (typable) in typing environment $\Gamma$ at type $\tau$ if there exists $e'$ such that $\Gamma \vdash_L e \leadsto e' : \tau$.*

Language $\lambda^{\text{<?>}}$ is an explicitly typed language and the rules for symbolic lifting are syntax directed, so it is straightforward to implement the type system with a recursive function. The input to such a function would be an empty typing environment and expression $e_1$; the output would be an expression $e_2$ and its type $\tau$ (which is also the type of $e_1$).

We now give an overview of the type and translation rules for the symbolic lifting relation, shown in Figure 6. The rules for variables and for lambda abstractions are standard and similar to the simply-typed lambda calculus. As usual, the rule (L-CONST) assumes a function $\Delta : \mathbb{C} \to \mathit{Types}$ that when applied to a constant returns the constant's type. We assume that the $\Delta$-function cannot return a symbolic type and therefore give the following assumption:

**Assumption 1** ($\Delta$-types)**.**
*If $\Delta(c) = \tau$ then $\tau \in \mathbb{G}$ or there exists $\tau_1$ and $\tau_2$ such that $\tau = \tau_1 \to \tau_2$.*

We define the lifting operator $\lceil e : \tau \rceil$ to check whether an expression has symbolic type, and if not, wrap it in a `lift` expression. Similarly, we define a lifting operator $\lceil \tau \rceil$ on types.

$$\lceil e : \tau \rceil = \begin{cases} e & \text{if } \tau \sim \text{<?>} \\ \texttt{lift}\ e{:}\tau & \text{otherwise} \end{cases} \qquad \lceil \tau \rceil = \begin{cases} \tau & \text{if } \tau \sim \text{<?>} \\ \text{<}\tau\text{>} & \text{otherwise} \end{cases}$$

**Proposition 2.**

1. *If $\Gamma \vdash_L e \leadsto e' : \tau$, then $\Gamma \vdash_L \lceil e : \tau \rceil \leadsto \lceil e' : \tau \rceil : \lceil \tau \rceil$.*
2. $\lceil \tau \rceil \sim \text{<?>}$

Because $\lambda^{\text{<?>}}$ is gradually typed, it does not require the argument of a function to be equal to the parameter type, but instead it may be consistent, as specified in rule (L-APP1). Also, the function expression may have type ?, in which case any argument type is allowed, as specified in rule (L-APP2). Next, to implement symbolic lifting, if

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash_L x \leadsto x : \tau_1}\ \text{(L-VAR)} \qquad \frac{\Gamma, x : \tau_1 \vdash_L e_1 \leadsto e_1' : \tau_2}{\Gamma \vdash_L \lambda x{:}\tau_1 . e_1 \leadsto \lambda x{:}\tau_1 . e_1' : \tau_1 \to \tau_2}\ \text{(L-ABS)}$$

$$\frac{\Delta(c) = \tau_1}{\Gamma \vdash_L c \leadsto c : \tau_1}\ \text{(L-CONST)} \qquad \Gamma \vdash_L \nu(\tau_1) \leadsto \nu(\tau_1) : \mathtt{<}\tau_1\mathtt{>} \quad \text{(L-NEW)}$$

$$\frac{}{\Gamma \vdash_L \mathtt{error} \leadsto \mathtt{error} : \tau}\ \text{(L-ERROR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \tau_{11} \to \tau_{12} \\ \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2 \qquad \tau_{11} \sim \tau_2\end{array}}{\Gamma \vdash_L e_1\ e_2 \leadsto e_1'\ e_2' : \tau_{12}}\ \text{(L-APP1)} \qquad \frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \mathtt{?} \\ \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2\end{array}}{\Gamma \vdash_L e_1\ e_2 \leadsto e_1'\ e_2' : \mathtt{?}}\ \text{(L-APP2)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e_1' : \mathtt{<}\tau_{11}\mathtt{>} \to \tau_{12} \qquad \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2 \qquad \mathtt{<}\tau_{11}\mathtt{>} \not\sim \tau_2 \qquad \tau_{11} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e_1'\ (\mathtt{lift}\ e_2' : \tau_2) : \tau_{12}}\ \text{(L-APP3)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e_1' : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash_L e_2 \leadsto e_2' : \mathtt{<}\tau_2\mathtt{>} \qquad \tau_{11} \not\sim \mathtt{<}\tau_2\mathtt{>} \qquad \tau_{11} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto (\mathtt{lift}\ e_1' : \tau_{11} \to \tau_{12}) @\, e_2' : \mathtt{<}\tau_{12}\mathtt{>}}\ \text{(L-APP4)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \mathtt{<}\tau_{11} \to \tau_{12}\mathtt{>} \\ \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2'' \\ \mathtt{<}\tau_{11}\mathtt{>} \sim \lceil \tau_2 \rceil\end{array}}{\Gamma \vdash_L e_1\ e_2 \leadsto e_1' @\, e_2'' : \mathtt{<}\tau_{12}\mathtt{>}}\ \text{(L-APP5)} \qquad \frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \mathtt{<?>} \\ \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2''\end{array}}{\Gamma \vdash_L e_1\ e_2 \leadsto e_1' @\, e_2'' : \mathtt{<?>}}\ \text{(L-APP6)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \tau_1 \quad \Gamma \vdash_L e_2 \leadsto e_2' : \tau_2 \quad \Gamma \vdash_L e_3 \leadsto e_3' : \tau_3 \\ \mathtt{<?>} \sim \tau_1 \quad \lceil \tau_2 \rceil \sim \lceil \tau_3 \rceil \quad \lceil e_2' : \tau_2, e_3' : \tau_3 \rceil = (\tau_5, e_2'', e_3'')\end{array}}{\Gamma \vdash_L \mathtt{case}(e_1, \mathtt{sym}{:}\tau_4, e_2, e_3) \leadsto \mathtt{case}(e_1', \mathtt{sym}{:}\tau_4, e_2'', e_3'') : \tau_5}\ \text{(L-CSYM)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \tau_1 \quad \Gamma, x_1 : \mathtt{<?>}, x_2 : \mathtt{<?>} \vdash_L e_2 \leadsto e_2' : \tau_2 \\ \Gamma \vdash_L e_3 \leadsto e_3' : \tau_3 \quad \mathtt{<?>} \sim \tau_1 \quad \lceil \tau_2 \rceil \sim \lceil \tau_3 \rceil \quad \lceil e_2' : \tau_2, e_3' : \tau_3 \rceil = (\tau_4, e_2'', e_3'')\end{array}}{\Gamma \vdash_L \mathtt{case}(e_1, x_1 @\, x_2, e_2, e_3) \leadsto \mathtt{case}(e_1', x_1 @\, x_2, e_2'', e_3'') : \tau_4}\ \text{(L-CAPP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e_1' : \tau_1 \quad \Gamma, x : \tau_4 \vdash_L e_2 \leadsto e_2' : \tau_2 \\ \Gamma \vdash_L e_3 \leadsto e_3' : \tau_3 \quad \mathtt{<?>} \sim \tau_1 \quad \lceil \tau_2 \rceil \sim \lceil \tau_3 \rceil \quad \lceil e_2' : \tau_2, e_3' : \tau_3 \rceil = (\tau_5, e_2'', e_3'')\end{array}}{\Gamma \vdash_L \mathtt{case}(e_1, \mathtt{lift}\ x : \tau_4, e_2, e_3) \leadsto \mathtt{case}(e_1', \mathtt{lift}\ x : \tau_4, e_2'', e_3'') : \tau_5}\ \text{(L-CLIFT)}$$

Figure 6: Type System and Symbolic Lifting for $\lambda^{\mathtt{<?>}}$.

the parameter type is symbolic but the argument type is not, then we lift the argument as specified in rule (L-APP3). In the following example, a function with a symbolic parameter type is applied to an integer, so the integer is lifted but the the application remains a normal function application.

$$\vdash (\lambda x{:}\texttt{<?>}.x)\ 5 \rightsquigarrow (\lambda x{:}\texttt{<?>}.x)\ (\texttt{lift}\ 5 : \texttt{Int})$$

On the other hand, if the argument type is symbolic but the parameter type is not, then we lift the function and change from normal application to symbolic application, as specified in rule (L-APP4). In the next example, we have a function applied to a symbol, so the function is lifted.

$$\vdash (\lambda x{:}\texttt{Int}.x)\ \nu(\texttt{Int}) \rightsquigarrow (\texttt{lift}\ (\lambda x{:}\texttt{Int}.x) : \texttt{Int} \to \texttt{Int})@\,\nu(\texttt{Int})$$

Next we consider the cases in which the function is already symbolic. The two rules (L-APP5) and (L-APP6) are analogous to the rules (L-APP1) and (L-APP2). The first handles the case when the function has symbolic function type and the second handles the case when the function has symbolic dynamic type. In both rules, the argument is lifted if it is not already symbolic. The following is an example of applying a symbolic function, so the application becomes a symbolic application and the argument is lifted.

$$\vdash \nu(\texttt{Int}{\to}\texttt{Int})\ 5 \rightsquigarrow \nu(\texttt{Int}{\to}\texttt{Int})@\,(\texttt{lift}\ 5 : \texttt{Int})$$

The next example demonstrates gradual typing for symbolic expressions.

$$\vdash \nu(\texttt{?})\ 5 \rightsquigarrow \nu(\texttt{?})@\,(\texttt{lift}\ 5 : \texttt{Int})$$

The function in this case is both symbolic and dynamic. Again, we change to a symbolic application and lift the argument.

To conclude our discussion of the lifting relation, we turn to the `case` expression, which decomposes symbolic data. There are three rules, corresponding to the three kinds of patterns: symbols, applications, and lifted values. In each case, we require $e_1$ to either have symbolic type or dynamic type, which is expressed by requiring that $\texttt{<?>} \sim \tau_1$. In the rule for application (L-CAPP), the branch $e_2$ is typed in a context that contains variables $x_1$ and $x_2$, both assigned the type <?>, which gives a dynamic flavor to decomposing symbolic data. To reconcile the types and terms of the two branches, we define the following operator that lifts a branch if necessary.

$$\lceil e_2 : \tau_2, e_3 : \tau_3 \rceil = \begin{cases} (\tau_2 \sqcap \tau_3, e_2, e_3) & \text{if } \tau_2 \sim \tau_3 \\ (\lceil \tau_2 \rceil \sqcap \lceil \tau_3 \rceil, \lceil e_2 : \tau_2 \rceil, \lceil e_3 : \tau_3 \rceil) & \text{otherwise} \end{cases}$$

## *3.4. Cast insertion*

The standard approach to defining the semantics of a gradually-typed language is to translate to an intermediate language that replaces the implicit injections and projections allowed by the consistency relation with explicit casts [28]. The explicit casts make it easier to reason about when errors should occur and better reflects the runtime representations that would be used in a compiled implementation.

$\boxed{\lambda^{<?>}_{LC}}$ (extends $\lambda^{<?>}_{L}$)

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

Figure 7: Abstract syntax of $\lambda^{<?>}_{LC}$.

The abstract syntax for $\lambda^{<?>}_{LC}$ is defined in Figure 7. A new expression $\langle \tau_2 \Leftarrow \tau_1 \rangle e$ for casts is defined, where the expression $e$ is cast from source type $\tau_1$ to target type $\tau_2$. Also we add an expression for the runtime representation of a symbol ($s\!:\!\tau$).

Cast insertion is defined by a four-place *cast insertion relation*

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

where $e$ is an expression in $\lambda^{<?>}_{L}$, $e'$ an expression in $\lambda^{<?>}_{LC}$, $\tau$ the resulting type, and $\Gamma$ the typing environment. The cast insertion relation is inductively defined by the inference rules in Figure 8. The rules are, for the most part, a straightforward extension to the standard cast insertion relation for gradual typing [28]. One interesting thing to note is that, in rules (C-SAPP1) and (C-SAPP2), the function and argument are cast to <?> because that is the type expected when a `case` expression decomposes a symbolic application. The notion of well-typed expression for $\lambda^{<?>}_{L}$ is defined in terms of the cast insertion relation.

**Definition 3** (Well-typed expression in $\lambda^{<?>}_{L}$)**.** *An expression $e$ of $\lambda^{<?>}_{L}$ is well typed (typable) in typing environment $\Gamma$ at type $\tau$ if there exists $e'$ such that $\Gamma \vdash_C e \rightsquigarrow e' : \tau$.*

The symoblic lifting translation, defined in the previous section, preserves types. That is, it translates well-typed expressions to well-typed expressions.

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

*Proof.* By induction on a derivation of $\Gamma \vdash_L e \rightsquigarrow e' : \tau$. All the cases are straightforward. □

Next we define the type system for $\lambda^{<?>}_{LC}$ by a three-place *typing relation*

$$\Gamma \vdash e : \tau$$

where $e$ is an expression in $\lambda^{<?>}_{LC}$, $\tau$ its type, and $\Gamma$ the typing environment. The typing relation is inductively defined in Figure 9. It is a *simple* type system in the sense of the simply-typed lambda calculus.

The cast insertion relation translates well-typed expressions to well-typed expressions.

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

*Proof.* The proof is a straightforward induction on the derivation of $\Gamma \vdash_C e \rightsquigarrow e' : \tau$. The cases for (C-CSYM), (C-CAPP), and (C-CLIFT) use Proposition 1. □

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash_C x \rightsquigarrow x : \tau_1}\ \text{(C-VAR)}$$

$$\frac{\Gamma, x : \tau_1 \vdash_C e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash_C \lambda x{:}\tau_1 . e_2 \rightsquigarrow \lambda x{:}\tau_1 . e_2' : \tau_1 \rightarrow \tau_2}\ \text{(C-ABS)} \qquad \frac{}{\Gamma \vdash_C \texttt{error} \rightsquigarrow \texttt{error} : \tau}\ \text{(C-ERROR)}$$

$$\frac{\Delta(c) = \tau_1}{\Gamma \vdash_C c \rightsquigarrow c : \tau_1}\ \text{(C-CONST)} \qquad \frac{}{\Gamma \vdash_C \nu(\tau_1) \rightsquigarrow \nu(\tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(C-NEWSYM)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1}{\Gamma \vdash_C \texttt{lift}\, e_1 : \tau_1 \rightsquigarrow (\texttt{lift}\, e_1' : \tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(C-LIFT)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow e_1'\ (\langle \tau_{11} \Leftarrow \tau_2 \rangle e_2') : \tau_{12}}\ \text{(C-APP1)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \texttt{?} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow (\langle \tau_2 \rightarrow \texttt{?} \Leftarrow \texttt{?} \rangle e_1')\ e_2' : \texttt{?}}\ \text{(C-APP2)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \texttt{<?>} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \texttt{<?>} \sim \tau_2}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow e_1' @ \langle \texttt{<?>} \Leftarrow \tau_2 \rangle e_2' : \texttt{<?>}}\ \text{(C-SAPP1)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_C e_1 \rightsquigarrow e_1' : \texttt{<}\tau_{11} \rightarrow \tau_{12}\texttt{>} \\ \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \sim \tau_2 \\ e_1'' = (\langle \texttt{<?>} \Leftarrow \texttt{<}\tau_{11} \rightarrow \tau_{12}\texttt{>} \rangle e_1') \qquad e_2'' = \langle \texttt{<?>} \Leftarrow \tau_2 \rangle e_2' \end{array}}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow \langle \texttt{<}\tau_{12}\texttt{>} \Leftarrow \texttt{<?>} \rangle (e_1'' @ e_2'') : \texttt{<}\tau_{12}\texttt{>}}\ \text{(C-SAPP2)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \\ \texttt{<?>} \sim \tau_1 \qquad \tau_2 \sim \tau_3 \qquad \tau_5 = \tau_2 \sqcap \tau_3 \\ e_1'' = \langle \texttt{<?>} \Leftarrow \tau_1 \rangle e_1' \qquad e_2'' = \langle \tau_5 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_5 \Leftarrow \tau_3 \rangle e_3' \end{array}}{\begin{array}{l} \Gamma \vdash_C \ \texttt{case}(e_1, \texttt{sym}{:}\tau_4, e_2, e_3) \rightsquigarrow \\ \qquad\qquad \texttt{case}(e_1'', \texttt{sym}{:}\tau_4, e_2'', e_3'') : \tau_5 \end{array}}\ \text{(C-CSYM)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, x_1 : \texttt{<?>}, x_2 : \texttt{<?>} \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \quad \texttt{<?>} \sim \tau_1 \quad \tau_2 \sim \tau_3 \quad \tau_4 = \tau_2 \sqcap \tau_3 \\ e_1'' = \langle \texttt{<?>} \Leftarrow \tau_1 \rangle e_1' \qquad e_2'' = \langle \tau_4 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_4 \Leftarrow \tau_3 \rangle e_3' \end{array}}{\begin{array}{l} \Gamma \vdash_C \ \texttt{case}(e_1, x_1 @ x_2, e_2, e_3) \rightsquigarrow \\ \qquad\qquad \texttt{case}(e_1'', x_1 @ x_2, e_2'', e_3'') : \tau_4 \end{array}}\ \text{(C-CAPP)}$$

$$\frac{\begin{array}{c} \Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, x : \tau_4 \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \quad \texttt{<?>} \sim \tau_1 \quad \tau_2 \sim \tau_3 \quad \tau_5 = \tau_2 \sqcap \tau_3 \\ e_1'' = \langle \texttt{<?>} \Leftarrow \tau_1 \rangle e_1' \qquad e_2'' = \langle \tau_5 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_5 \Leftarrow \tau_3 \rangle e_3' \end{array}}{\begin{array}{l} \Gamma \vdash_C \ \texttt{case}(e_1, \texttt{lift}\, x : \tau_4, e_2, e_3) \rightsquigarrow \\ \qquad\qquad \texttt{case}(e_1'', \texttt{lift}\, x : \tau_4, e_2'', e_3'') : \tau_5 \end{array}}\ \text{(C-CLIFT)}$$

Figure 8: The Cast Insertion Relation.

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash x : \tau_1}\ \text{(T-VAR)} \qquad \frac{\Delta(c) = \tau_1}{\Gamma \vdash c : \tau_1}\ \text{(T-CONST)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 . e_2 : \tau_1 \rightarrow \tau_2}\ \text{(T-ABS)} \qquad \frac{\Gamma \vdash e_1 : \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 : \tau_{11}}{\Gamma \vdash e_1\ e_2 : \tau_{12}}\ \text{(T-APP)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \sim \tau_2}{\Gamma \vdash \langle \tau_2 \Leftarrow \tau_1 \rangle e_1 : \tau_2}\ \text{(T-CAST)} \qquad \frac{}{\Gamma \vdash \texttt{error} : \tau}\ \text{(T-ERROR)}$$

$$\frac{}{\Gamma \vdash (s : \tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(T-SYM)} \qquad \frac{}{\Gamma \vdash \nu(\tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(T-NEWSYM)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (\texttt{lift}\ e_1 : \tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(T-LIFT)} \qquad \frac{\Gamma \vdash e_1 : \texttt{<?>} \qquad \Gamma \vdash e_2 : \texttt{<?>}}{\Gamma \vdash e_1 @ e_2 : \texttt{<?>}}\ \text{(T-SAPP)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2}{\Gamma \vdash \texttt{case}(e_1, \texttt{sym} : \tau_4, e_2, e_3) : \tau_2}\ \text{(T-CASE-SYM)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>} \qquad \Gamma, x_1 : \texttt{<?>}, x_2 : \texttt{<?>} \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2}{\Gamma \vdash \texttt{case}(e_1, x_1 @ x_2, e_2, e_3) : \tau_2}\ \text{(T-CASE-APP)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>} \qquad \Gamma, x : \tau_4 \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2}{\Gamma \vdash \texttt{case}(e_1, \texttt{lift}\ x : \tau_4, e_2, e_3) : \tau_2}\ \text{(T-CASE-LIFT)}$$

Figure 9: Type System for $\lambda_{LC}^{\texttt{<?>}}$.

## *3.5. Dynamic semantics*

We define the dynamic semantics of $\lambda^{<?>}$ in Figure 10 by defining a partial function *eval* from well-typed $\lambda^{<?>}$ expressions to observations.[4] A valid implementation of $\lambda^{<?>}$ must produce the same observation as specified by *eval* for a given expression. The *eval* function is defined in terms of the lifting and cast insertion translations as well as an operational semantics for $\lambda^{<?>}_{LC}$ in *small-step* style [49]. The shape of the single-step reduction relation is

$$e \mid S \longrightarrow e' \mid S'$$

where expression $e$ is reduced to $e'$ in one step, and $S$ and $S'$ are sets of symbols. The meta-variable $S \subseteq \mathbb{S}$ ranges over a (potentially empty) set of symbols. Hence, the operational semantics includes computational effects in terms of new symbols that are created during evaluation.

The reduction relation determines a notion of *value*, which constitutes the set of well-typed, closed expressions that cannot be further reduced. (All other expressions that cannot be reduced are called "stuck" expressions.) In Figure 10 we present an equivalent definition for values in terms of a grammar. This equivalence is a corollary of the Progress Lemma that is proved in Section 3.6. As usual, values include constants and functions. In addition, because $\lambda^{<?>}_{LC}$ has casts, there are several value forms for casted values. Lastly, there are three values forms for the three kinds of symbolic data.

The axiom

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

is the standard function application rule ($\beta$-reduction). The value $v_1$ is substituted for $x$ in $e_1$. The notation $[x \mapsto v_1]e_1$ stands for standard capture-avoiding substitution, where $v_1$ is substituted for all $x$ that appear free in $e_1$. For completeness the definitions of substitution and the free variable function $FV(e)$ are given in the Appendix. As usual, we identify expressions up to exchanging the names of bound variables.

The second rule handles the application of built-in functions, and is also standard:

$$c_1\ v_1 \mid S \longrightarrow \delta(c_1, v_1) \mid S \quad \text{(E-DELTA)}$$

The $\delta$ function abstracts the computation of built-in operators.

The rule

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

creates new symbols. The side condition $s \notin S$ means that we pick a fresh symbol $s$ that is not in the set $S$. The new state is augmented with the new symbol. Note that the resulting symbolic expression $s{:}\tau_1$ is tagged with the type $\tau_1$ from the $\nu$-expression.

[4]Many authors omit the definition of such an *eval* function, but as Felleisen, et al. [48] correctly point out, every language definition should be explicit about what is ultimately considered observable.

Runtime Structures

$$\begin{array}{llll}
\text{Static Types} & \sigma & ::= & \gamma \mid \tau{\rightarrow}\tau \mid {<}\tau{>} \mid D \\
\text{Values} & v & ::= & \lambda x{:}\tau.e \mid c \mid \\
 & & & \langle ? \Leftarrow \sigma\rangle v \mid \langle \tau_3 \rightarrow \tau_4 \Leftarrow \tau_1 \rightarrow \tau_2\rangle v \mid \langle {<}\tau_2{>} \Leftarrow {<}\tau_1{>}\rangle v \mid \\
 & & & s\,{:}\,\tau \mid v\,@\,v \mid \mathtt{lift}\;v\,{:}\,\tau \\
\text{Frames} & F & ::= & \Box\; e_2 \mid v\;\Box \mid \mathtt{case}(\Box, p, e_2, e_3) \mid \langle \tau_1 \Leftarrow \tau_2\rangle \Box \mid \Box @\, e_2 \mid \\
 & & & v @ \Box \mid \mathtt{lift}\;\Box\,{:}\,\tau_1
\end{array}$$

$\boxed{e \longrightarrow e}$

$$(\lambda x{:}\tau_1.e_1)\; v_1 \mid S \longrightarrow [x \mapsto v_1]e_1 \mid S \qquad \text{(E-BETA)}$$
$$c_1\; v_1 \mid S \longrightarrow \delta(c_1, v_1) \mid S \qquad \text{(E-DELTA)}$$

$$(\langle \tau_1{\rightarrow}\tau_2 \Leftarrow \tau_3{\rightarrow}\tau_4\rangle v_1)\; v_2 \mid S \longrightarrow \langle \tau_2 \Leftarrow \tau_4\rangle (v_1\; \langle \tau_3 \Leftarrow \tau_1\rangle v_2) \mid S \qquad \text{(E-CAST1)}$$
$$\langle \gamma \Leftarrow \gamma\rangle v_1 \mid S \longrightarrow v_1 \mid S \qquad \text{(E-CAST2)}$$
$$\langle ? \Leftarrow ?\rangle v_1 \mid S \longrightarrow v_1 \mid S \qquad \text{(E-CAST3)}$$
$$\langle T_2 \Leftarrow ?\rangle\langle ? \Leftarrow T_1\rangle v \mid S \longrightarrow \langle T_2 \Leftarrow T_1\rangle v \qquad \text{if } T_1 \sim T_2 \qquad \text{(E-CAST4)}$$
$$\langle T_2 \Leftarrow ?\rangle\langle ? \Leftarrow T_1\rangle v \mid S \longrightarrow \mathtt{error} \qquad \text{if } T_1 \not\sim T_2 \qquad \text{(E-CAST5)}$$

$$\nu(\tau_1) \mid S \longrightarrow s\,{:}\,\tau_1 \mid S \cup \{s\} \qquad \text{if } s \notin S \qquad \text{(E-NEWSYM)}$$
$$\mathtt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_2' \mid S \qquad \text{if } \mathit{match}(v_1, p, e_2, e_2') \qquad \text{(E-CASE-T)}$$
$$\mathtt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_3 \mid S \qquad \text{if } \neg\mathit{match}(v_1, p, e_2, e_2') \qquad \text{(E-CASE-F)}$$
$$\mathtt{case}(\langle {<}\tau_2{>} \Leftarrow {<}\tau_1{>}\rangle v_1, p, e_2, e_3) \mid S \longrightarrow \mathtt{case}(v_1, p, e_2, e_3) \mid S \qquad \text{(E-CASE-C)}$$

$$\frac{e \mid S \longrightarrow e' \mid S'}{F[e] \mid S \longrightarrow F[e'] \mid S'}\text{(E-CONG)} \qquad \frac{}{F[\mathtt{error}\,] \longrightarrow \mathtt{error}}\text{(E-ERROR)}$$

Observations

$$\begin{aligned}
\mathit{observe}(\lambda x{:}\tau.e) &= \mathit{function} \\
\mathit{observe}(c) &= c \\
\mathit{observe}(\langle ? \Leftarrow \sigma\rangle v) &= \mathit{dynamic} \\
\mathit{observe}(\langle \tau_3 \rightarrow \tau_4 \Leftarrow \tau_1 \rightarrow \tau_2\rangle v) &= \mathit{function} \\
\mathit{observe}(\langle {<}\tau_2{>} \Leftarrow {<}\tau_1{>}\rangle v) &= \mathit{symbolic} \\
\mathit{observe}(s\,{:}\,\tau) &= \mathit{symbolic} \\
\mathit{observe}(v_1\,@\,v_2) &= \mathit{symbolic} \\
\mathit{observe}(\mathtt{lift}\;v\,{:}\,\tau) &= \mathit{symoblic}
\end{aligned}$$

The Dynamic Semantics of $\lambda^{<?>}$, the *eval* function.

$$\mathit{eval}(e) = \begin{cases} \mathit{observe}(v) & \text{if } \emptyset \vdash_L e \rightsquigarrow e' : \tau, \emptyset \vdash_C e' \rightsquigarrow e'' : \tau, \text{ and } e'' \longrightarrow^* v \\ \mathtt{error} & \text{if } \emptyset \vdash_L e \rightsquigarrow e' : \tau, \emptyset \vdash_C e' \rightsquigarrow e'' : \tau, \text{ and } e'' \longrightarrow^* \mathtt{error} \\ \bot & \text{otherwise} \end{cases}$$

Figure 10: Dynamic Semantics of $\lambda^{<?>}$.

$$\mathit{match}(s : \tau_1, \texttt{sym} : \tau_1, e_1, e_1) \quad \text{(M-SYM)}$$

$$\mathit{match}(v_1 @ v_2, x_1 @ x_2, e_1, (\lambda x_1\text{:<?>}.\lambda x_2\text{:<?>}.e_1)\ v_1\ v_2) \quad \text{(M-SAPP)}$$

$$\mathit{match}(\texttt{lift}\ v_1 : \tau_1, \texttt{lift}\ x : \tau_1, e_1, (\lambda x : \tau_1 . e_1)\ v_1) \quad \text{(M-LIFT)}$$

Figure 11: Definition of *match* predicate.

The following rules *deconstruct* symbolic expressions.

$$\begin{array}{lrl}
\texttt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_2' \mid S & \text{if } \mathit{match}(v_1, p, e_2, e_2') & \text{(E-CASE-T)} \\
\texttt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_3 \mid S & \text{if } \neg\mathit{match}(v_1, p, e_2, e_2') & \text{(E-CASE-F)}
\end{array}$$

The value $v_1$, the deconstructor pattern $p$, and the expression $e_2$ are given to the *match* predicate defined in Figure 11. If the match is true, the rule (E-CASE-T) applies and $e_2'$ is returned. If the match is false, the rule (E-CASE-F) applies and $e_3$ is returned. Note that the axioms (M-SYM), (M-SAPP), and (M-LIFT) of the match relation check the shape of the expression (that it is a symbol, symbolic application, or a lifted symbolic value). Moreover, (M-SYM) and (M-LIFT) also check that the type tag $\tau_1$ is equal to the type tag of the pattern.

**Example 3.5.** Let $e_x$ denote the following expression

$$(s_1 : \texttt{<Int}\rightarrow\texttt{Int>}) @ (\texttt{lift}\ 5 : \texttt{Int})$$

The following example evaluation steps show the basic idea of the symbolic expression deconstruction.

$$\begin{array}{rcl}
\texttt{case}(e_x, x @ y, x, \texttt{lift}\ 1.1 : \texttt{Real}) \mid S & \longrightarrow & s_1 : \texttt{<Int}\rightarrow\texttt{Int>} \mid S \\
\texttt{case}(e_x, \texttt{sym} : \texttt{Real}, x, \texttt{lift}\ 1.1 : \texttt{Real}) \mid S & \longrightarrow & \texttt{lift}\ 1.1 : \texttt{Real} \mid S \\
\texttt{case}(e_x, x @ y, y, \texttt{lift}\ 1.1 : \texttt{Real}) \mid S & \longrightarrow & \texttt{lift}\ 5 : \texttt{Int} \mid S \\
\texttt{case}(\texttt{lift}\ 5 : \texttt{Int}, \texttt{lift}\ x : \texttt{Int}, x, 20) \mid S & \longrightarrow & 5 \mid S \\
\texttt{case}(s_2 : \texttt{<Int}\rightarrow\texttt{Int>}, \texttt{sym} : \texttt{<Int}\rightarrow\texttt{Int>}, 1, 2) \mid S & \longrightarrow & 1 \mid S \\
\texttt{case}(s_3 : \texttt{<Int}\rightarrow\texttt{Int>}, \texttt{sym} : \texttt{<Real>}, 1, 2) \mid S & \longrightarrow & 2 \mid S
\end{array}$$

In addition to the rules for function application, there are also five rules for handling casts, which are standard for cast calculi [50] but perhaps deserve some review. Because we have casted values at function type, there must be a reduction rule for applying such a value. Reduction rule (E-CAST1) handles this case by distributing the function cast to the function's argument and return type. (There is an alternative approach that does not have casted values at function type, but instead creates a new wrapper function when a cast is applied to a function [51]. The two approaches are observationally equivalent.) The reduction rules (E-CAST2) and (E-CAST3) discard identity casts on ground types and on type ?. The rules (E-CAST4) and (E-CAST5) handle the important case of an injection to type ? meeting a projection from type ?. If the source $T_1$ and target $T_2$ are consistent, then the two casts collapse to a single cast. Otherwise, the casts result in a run-time cast error. Our use of consistency here instead of shallow consistency [50] provides earlier and more thorough error detection [52].

There is one one new reduction rule regarding casts, for when a casted symbolic value is decomposed by a `case`. Because the typing rule for `case` only cares whether the value is of symbolic type, we can drop the cast while preserving types.

$$\texttt{case}(\langle{<}\tau_2{>} \Leftarrow {<}\tau_1{>}\rangle v_1, p, e_1, e_2) \mid S \longrightarrow \texttt{case}(v_1, p, e_1, e_2) \mid S \qquad \text{(E-CAST-C)}$$

We succinctly express the very many congruence rules with the following rule schema, inspired by unpublished lecture notes by Andrew Myers.

$$\frac{e \mid S \longrightarrow e' \mid S'}{F[e] \mid S \longrightarrow F[e'] \mid S'}\text{(E-CONG)}$$

The $F$ is a frame, defined in Figure 10 and the notation $F[e]$ means to replace the hole, written $\square$, inside $F$ with the expression $e$. Thus, by choosing different $F$'s one arrives at each of the different congruence rules. For example, frame $\square\ e_2$ represents the following congruence rule:

$$\frac{e \mid S \longrightarrow e' \mid S'}{e\ e_2 \mid S \longrightarrow e'\ e_2 \mid S'}$$

Finally, we define the *reflexive transitive closure* relation

$$e \mid S \ \longrightarrow^* \ e' \mid S'$$

by the following two rules

$$e| \ S \ \longrightarrow^* \ e| \ S \quad \text{(RTC-REFL)}$$

$$\frac{e \mid S \ \longrightarrow^* \ e' \mid S' \quad e' \mid S' \ \longrightarrow \ e'' \mid S''}{e \mid S \ \longrightarrow^* \ e'' \mid S''}\ \text{(RTC-STEP)}$$

## *3.6. Type safety*

In this section we prove type safety by first proving the usual progress and preservation lemmas for the intermediate language $\lambda^{<?>}_{LC}$. Type safety for $\lambda^{<?>}$ is then established using the type preservation of symbolic lifting (Proposition 3) and type preservation of cast insertion (Proposition 4). The proof strategy for type safety that we use has its origins in the syntactic soundness approach by Wright and Felleisen [53], but is now typically organized in a different way. We are using an approach similar to Pierce [54].

We start by proving basic lemmas about the typing relation. First, we prove the inversion of typing relation.

**Lemma 1** (Inversion of Typing Relation)**.**

1. *If* $\Gamma \vdash x : \tau$ *then* $\Gamma(x) = \tau$.
2. *If* $\Gamma \vdash (s : \tau_1) : \tau$ *then* $\tau = \texttt{<}\tau_1\texttt{>}$.
3. *If* $\Gamma \vdash \lambda x{:}\tau_1.e_2 : \tau$ *then there exists* $\tau_2$ *such that* $\tau = \tau_1 \rightarrow \tau_2$ *and* $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.
4. *If* $\Gamma \vdash c : \tau$ *then* $\Delta(c) = \tau$.
5. *If* $\Gamma \vdash \nu(\tau_1) : \tau$ *then* $\tau = \texttt{<}\tau_1\texttt{>}$.
6. *If* $\Gamma \vdash (\texttt{lift}\, e_1 : \tau_1) : \tau$ *then* $\tau = \texttt{<}\tau_1\texttt{>}$ *and* $\Gamma \vdash e_1 : \tau_1$.
7. *If* $\Gamma \vdash e_1\ e_2 : \tau$ *then there exists a* $\tau_{11}$ *such that* $\Gamma \vdash e_1 : \tau_{11} \rightarrow \tau$ *and* $\Gamma \vdash e_2 : \tau_{11}$.
8. *If* $\Gamma \vdash e_1 @ e_2 : \tau$ *then* $\tau = \texttt{<?>}$, $\Gamma \vdash e_1 : \texttt{<?>}$, *and* $\Gamma \vdash e_2 : \texttt{<?>}$.
9. *If* $\Gamma \vdash \langle \tau_2 \Leftarrow \tau_1 \rangle e_1 : \tau$ *then* $\tau = \tau_2$, $\Gamma \vdash e_1 : \tau_1$, *and* $\tau_1 \sim \tau_2$.
10. *If* $\Gamma \vdash \texttt{case}(e_1, \texttt{sym}{:}\tau_4, e_2, e_3) : \tau$ *then there exists* $\tau_1$ *such that* $\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>}$, $\Gamma \vdash e_2 : \tau$, *and* $\Gamma \vdash e_3 : \tau$.
11. *If* $\Gamma \vdash \texttt{case}(e_1, x_1 @ x_2, e_2, e_3) : \tau$ *then there exists* $\tau_1$ *such that* $\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>}$ *and* $\Gamma, x_1 : \texttt{<?>}, x_2 : \texttt{<?>} \vdash e_2 : \tau$ *and* $\Gamma \vdash e_3 : \tau$.
12. *If* $\Gamma \vdash \texttt{case}(e_1, \texttt{lift}\, x : \tau_4, e_2, e_3) : \tau$ *then there exists* $\tau_1$ *such that* $\Gamma \vdash e_1 : \texttt{<}\tau_1\texttt{>}$ *and* $\Gamma, x : \tau_4 \vdash e_2 : \tau$ *and* $\Gamma \vdash e_3 : \tau$.

*Proof.* By inspection of the definition of $\Gamma \vdash e : \tau$. $\square$

The next lemma tell us the shape of a value, given its type:

**Lemma 2** (Canonical Forms)**.**

1. *If* $\Gamma \vdash v : \gamma$ *then* $\exists c \in \mathbb{C}.\ c = v$.
2. *If* $\Gamma \vdash v : \tau_1 \rightarrow \tau_2$ *then* $(\exists x\ e.\ (\lambda x{:}\tau_1.e) = v)$, $(\exists c.c = v)$, *or* $(\exists \tau_1\ \tau_2\ \tau_3\ \tau_4\ v'.\langle \tau_1 \rightarrow \tau_2 \Leftarrow \tau_3 \rightarrow \tau_4 \rangle v' = v)$.
3. *If* $\Gamma \vdash v : \texttt{<}\tau\texttt{>}$ *then* $(\exists s.\ s : \tau = v)$ *or* $(\exists v'.\ \texttt{lift}\ v' : \tau = v)$ *or* $(\exists v_1\ v_2.\ \tau = \texttt{<?>} \wedge v_1 @ v_2 = v)$ *or* $(\exists \tau'\ v'.\langle \texttt{<}\tau\texttt{>} \Leftarrow \texttt{<}\tau'\texttt{>} \rangle v' = v)$.
4. *If* $\Gamma \vdash v : \texttt{?}$ *then* $\exists \sigma v'.\ v = \langle \texttt{?} \Leftarrow \sigma \rangle v'$.
5. *There are no values of type* $D$. *(But there are values of type* $\texttt{<}D\texttt{>}$*.)*

We are now ready to state one of the main lemmas of the proof, that a well-typed expression is either a value or we can take a step:

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

*Proof.* By induction on a derivation of $\vdash e : \tau$. Case (T-VAR) cannot occur, because $e$ is closed. In cases (T-SYM), (T-ABS), and (T-CONST) we have $e \in \mathit{Values}$.

For case (T-ERROR), we immediately have $e = \texttt{error}$ .

For case (T-NEWSYM), the expression reduces by (E-NEWSYM).

For case (T-APP), the induction hypothesis says that each subexpression either reduces, is a value, or is an error. The error cases are handled by reduction rule (E-ERROR). If a subexpression reduces, the entire expression reduces by (E-CONG). Otherwise, both subexpressions are values. The Canonical Forms Lemma then tells us that the value in function position is either 1) a function, 2) a constant, or 3) a cast between function types. If it's a function, reduce by (E-BETA), if it's a constant, reduce by (E-DELTA), and finally, if its a cast, reduce by (E-CAST1).

Case (T-CAST): By induction hypothesis, $e_1$ can either take a step, it is a value, or it is an error. If it's an error, it reduces by (E-ERROR). If it can take a step, rule (E-CONG) applies. If $e_1$ is a value, we proceed by case analysis on $\tau_1 \sim \tau_2$. Suppose $\tau_2 = ?$. If $\tau_1$ is also ?, then (E-CAST3) applies. Otherwise, the entire expression is a value. Next, suppose $\tau_1 = ?$ and $\tau_2$ is not ?. Then by the Canonical Forms Lemma, $e_1$ has the form $\langle ? \Leftarrow \tau_3 \rangle v$. If $\tau_3 \sim \tau_2$, then (E-CAST4) applies, and otherwise, (E-CAST5) applies. Suppose $\tau_1 = \tau_2 = \gamma$. Then (E-CAST2) applies. Suppose $\tau_1 = \tau_2 = D$. Then the value $e_1$ has type $D$, but the Canonical Forms Lemma says that is not possible. Suppose $\tau_1 = \tau_{11} \to \tau_{12}, \tau_2 = \tau_{21} \to \tau_{22}$. Then the entire expression is a value. Suppose $\tau_1 = \texttt{<}\tau_1'\texttt{>}, \tau_2 = \texttt{<}\tau_2'\texttt{>}$. Again, the entire expression is a value.

For cases (T-LIFT) and (T-SAPP), the induction hypothesis tells us that the subexpressions are either values or errors or they reduce. If they are all values, then the entire expression is a value.

If one of the subexpression reduces, then the entire expression reduces by rule (E-CONG).

For cases (T-CASE-SYM), (T-CASE-APP), and (T-CASE-LIFT), the induction hypothesis tells us that $e_1$ can either take a step or is a value. If it can take a step, (E-CONG) applies. In case of a value, the Canonical Forms Lemma tells us there are four kinds of value of symbolic type to consider. If $e_1$ is of the form $\langle \texttt{<}\tau_1\texttt{>} \Leftarrow \texttt{<}\tau_3\texttt{>} \rangle v$, then (E-CAST-C) applies. For the other kinds of values of symbolic type, either (E-CASE-T) or (E-CASE-F) applies.

□

We require that the $\delta$ function agrees with the $\Delta$ function with respect to the types of the values it produces.

**Assumption 2** ($\delta$-typability)**.**
*If* $\Delta(c) = \tau_1 \to \tau_2$ *and* $\Gamma \vdash v : \tau_1$ *then* $\Gamma \vdash \delta(c, v) : \tau_2$.

Towards proving the Preservation Lemma, we need a Substitution Lemma (because substitution occurs during function application). In turn, the Substitution Lemma requires an Environment Weakening Lemma. The proofs of these lemmas are standard.

**Lemma 4** (Environment Weakening)**.** *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash e : \tau$.

**Lemma 5** (Substitution)**.** *If* $\Gamma, y : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash [y \mapsto e']e : \tau$.

**Lemma 6** (Match Preservation)**.** *Suppose* $\Gamma \vdash \mathtt{case}(v_1, p, e_2, e_3) : \tau$. *If* $\mathit{match}(v_1, p, e_2, e_2')$*, then* $\Gamma \vdash e_2' : \tau$.

*Proof.* We proceed by cases on the pattern $p$. If $p = \texttt{sym} : \tau_4$, then by the Inversion lemma, $\Gamma \vdash e_2 : \tau$. In this case we have $e_2' = e_2$, so we immediately have $\Gamma \vdash e_2' : \tau$.

If $p = x_1 \mathbin{@} x_2$, then by Inversion we have $\Gamma, x_1 : \texttt{<?>}, x_2 : \texttt{<?>} \vdash e_2 : \tau$. Let $e_\lambda = (\lambda x_1{:}\texttt{<?>}.\, \lambda x_2{:}\texttt{<?>}.\, e_2)$. Then we have $\Gamma \vdash e_\lambda : \texttt{<?>} \to \texttt{<?>} \to \tau$. Now because $\mathit{match}(v_1, x_1 \mathbin{@} x_2, e_2, e_2')$, we have $v = v1@v2$ for some $v_1$ and $v_2$. By Inversion we have $\Gamma \vdash v_1 : \texttt{<?>}$ and $\Gamma \vdash v_2 : \texttt{<?>}$. So we can conclude that $\Gamma \vdash e_\lambda\, v_1\, v_2 : \tau$.

If $p = \texttt{lift}\ x : \tau_4$, then by the Inversion lemma we have $\Gamma \vdash v_1 : \tau_1$ and $\Gamma, x : \tau_4 \vdash e_2 : \tau$. Also, because *match*$(v_1, \texttt{lift}\ x : \tau_4, e_2, e_2')$, we have $v_1 = \texttt{lift}\ v_1' : \tau_1$ and $\tau_1 = \tau_4$. By Inversion, we have $\Gamma \vdash v_1' : \tau_1$. So $\Gamma \vdash \lambda x{:}\tau_1.\, e_2 : \tau_1 \rightarrow \tau$ and we can conclude that $\Gamma \vdash (\lambda x{:}\tau_1.\, e_2)\, v_1' : \tau$.

□

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

*Proof.* The proof is by induction on the reduction $e \mid S \longrightarrow e' \mid S'$. We first dispatch the standard cases. The case for (E-BETA) follows from the Inversion, Canonical Forms, and Substitution lemmas. The case for (E-DELTA) follows from the Inversion and Canonical Forms lemmas and the $\delta$-typability assumption. The cast cases (E-CAST1) through (E-CAST4) follow immediately from the Inversion lemma. For (E-CAST5), note that the `error` expression can be assigned any type, so this case is trivial.

We now come to the cases that are unique to $\lambda^{<?>}_{LC}$. The case (E-NEWSYM) is truly trivial. The (E-CASE-T) case follows from the Match Preservation lemma. The (E-CASE-F) case is straightforward. The (E-CASE-C) case is unusual, but straightforward. This reduction removes a cast, which normally would not be guaranteed to preserves types, but here the enclosing case expression is well-typed so long as the discriminated expression has symbolic type, and indeed, $v_1$ is of symbolic type. Finally, the (E-CONG) case follows immediately from the induction hypothesis.

□

**Theorem 1** (Type Safety of $\lambda^{<?>}$). *If* $\vdash_L e_1 \rightsquigarrow e_2 : \tau$ *then there exists an* $e_3$ *such that* $\vdash_C e_2 \rightsquigarrow e_3 : \tau$ *and (if* $e_3 \mid S_3 \longrightarrow^* e_4 \mid S_4$ *then* $\vdash e_4 : \tau$ *and (*$e_4 \in$ *Values, or* $e_4 = \texttt{error}$*, or there exists* $e_5$ *and* $S_5$ *such that* $e_4 \mid S_4 \longrightarrow e_5 \mid S_5$*)).*

*Proof.* By applying Proposition 3, soundness of symbolic lifting, to $\vdash_L e_1 \rightsquigarrow e_2 : \tau$ we have $\vdash_C e_2 \rightsquigarrow e_3 : \tau$ for some $e_3$. Also, by soundness of cast insertion (Proposition 4), we have $\vdash e_3 : \tau$. By induction on a derivation of $e \mid S \longrightarrow^* e' \mid S'$ we have two cases: In the base case (RTC-REFL) $e = e'$ and we directly have $\vdash e' : \tau$. By applying Progress to $\vdash e' : \tau$ we show that $e'$ is a value, $e'$ is an error, or there exists $e''$ and $S''$ such that $e' \mid S' \longrightarrow e'' \mid S''$. For case (RTC-STEP) we have by induction hypothesis $\vdash e' : \tau$. Also, by applying Preservation to assumption $e' \mid S' \longrightarrow e'' \mid S''$, we obtain $\vdash e'' : \tau$. By applying Progress to $\vdash e'' : \tau$ we reach the conclusion. □

## 4. Evaluation

In this section we evaluate the Modelyze approach in the context of creating embedded equation-based modeling languages. The evaluation is divided into three units of analysis, each representing a specific equation-based DSL. The relationship between the DSLs are depicted in the following diagram

$$\begin{tikzcd} & \text{M-DAE} \arrow[dl] \arrow[dr] & \\ \text{M-EOO} & & \text{M-HC} \end{tikzcd}$$

where the arrows indicate the direction of DSL extensions. Language M-DAE is the basic DSL for expressing differential-algebraic equations, exemplified with the pendulum example in Section 2. We extend this DSL in two ways, with support for equation-based object-oriented models (M-EOO) and hybridcharts (M-HC). We selected these three DSLs to cover a variety of modeling paradigms/formalisms and to be representative of implementation challenges in the domain of equation-based modeling languages. The questions in this evaluation address the challenges previously presented in Section 2:

1. Is the host language and the DSELs seamlessly integrated?
2. Which kinds of errors can be statically reported at model-level?
3. What are the strengths and limitations for symbolic analysis and transformation?
4. To what extent do open symbolic types enable DSL reuse?

Questions one and two concern problems relevant to the model engineer, whereas questions three and four are important for the domain expert. These questions have an inherent qualitative nature, making our analysis and discussion qualitative rather than quantitative.

The analyzed artifacts are (1) the prototype implementation of the Modelyze interpreter, (2) the DSEL implementations, and (3) the example models implemented in each DSL. The interpreter is implemented in the OCaml [55] v3.12.1 except for the DAE solver, for which we use IDA from the SUNDIALS [12] solver suite v2.4.0. Plotting and visualization is performed with Gnuplot v3.3. The DSLs and the corresponding models are implemented in Modelyze code. In the rest of this section, we discuss each of the three DSLs separately.

## *4.1. Differential-Algebraic Equation (DAE) DSL*

The M-DAE DSL is designed to be a declarative language for expressing differential-algebraic equations. In particular, the chosen notation is close to the mathematical notation of DAEs. In Section 2 we illustrated this declarative aspect using a pendulum example. If the host language and the DSL, however, are not seamlessly integrated, the declarative aspect may be lost. One aspect of such seamless integration is the ability to hide the existence of symbolic types from the modeling engineer. The pendulum example from Section 2 can be restated as

```
def PendulumExt(m:Real, l:Real, T:Force, x:Pos, y:Pos) = {
    -T*x/l = m*x'';
    -T*y/l - m*g = m*y'';
    x^2 + y^2 = l^2;
}
```

where the main change is that the symbolic definitions of `x`, `y`, and `T` are moved outside the function abstract and instead passed as arguments. Types `Force` and `Pos` are defined by the domain expert and are type aliases for the symbolic type <`Real`>, but the model engineer need not care how they are defined. Type aliases for physical dimensions, such as these, are typically defined by domain engineers in standard libraries. As explained in previous sections, the symbol lifting analysis phase enables such seamless integration. If symbol lifting was not performed, the model engineer would need to add explicit annotations in the equations. In the `PendelumExt` example, this means that the differences between variables of symbolic type (`T`, `x`, and `y`) and non symbolic types (`m` and `l`) need to be stated explicitly using quasi-quote notation.

Seamless integration between the host language and the DSL also means that the type checker should not leak out symbolic type information when reporting errors. In the `PendelumExt` model, assume that the first variable `y` in the second equation is replaced by `der`, resulting in equation

```
    -T*der/l - m*g = m*y'';
```

By executing the modified model in the prototype implementation, we get the following error message:

```
breakingpendulum.moz 3:10-3:10 error: Missing argument of
type 'Real'.
```

Both `y` and `der` represent symbols, but of different types <`Real`> and <`Real->Real`>, respectively. In this case, the model engineer does not even have to be aware of the existence of symbolic types. The type checker just informs that an argument to the derivative operator `der(x)` is missing, where the argument `x` needs to be of type `Real`. On the other hand, if the model would have been completely dynamically typed, the error would not have been detected until run-time.

Types at the model level rules out a large class of modeling errors, but not all possible semantic errors. For example, expression `PendulumExt(5, 2, 10, a, b)`, where `a` and `b` are symbols of type <`Real`>, is well typed. However, the system of equations is over-determined (three equations and two unknowns) because a value 10 is supplied instead of a symbol that represents the unknown for `T`. Such a semantic property of the equation system can be encoded as part of a type system [43, 45], but is not possible to express at the type level using our DSEL approach. Instead, such properties must be checked later by a function that analyze the resulting equation system.

Seamless integration and good error detection is important for the model engineer, whereas the domain expert needs flexibility to implement the semantics of a DSL. The semantics of M-DAE is defined by a sequence of symbolic transformation steps, followed by a run-time simulation loop. The two most essential transformation steps are symbolic differentiation and index reduction. In the former step, higher-order derivatives (e.g., `x''`) are eliminated, and expressions enclosed within differentiation terms (e.g., `(2*x^3)'`) are differentiated. For example, line 2 of the `PendulumExt` model is expanded into two equations

```
    -T*x/l = m*dx';
    x' = dx;
```

where `dx` represents a new fresh symbol. The second transformation step of index reduction is a symbolic technique for making it possible to simulate a larger class of DAEs than what can be solved by a numerical DAE solver such as IDA from SUNDIALS. One of the essential algorithms used in this step was invented by Pantelides [9]. This graph-based algorithm identifies equations that need to be differentiated. We use typed symbolic expressions for implementing the needed data structures. For example, the graph representing the equation system is expressed as a mapping `ENode => [VNode]` between equation nodes (`ENode`) and a list of variable nodes `[VNode]`. Both kinds of nodes are typed symbols. The benefit from a domain expert's point of view is that the M-DAE DSL can be defined compactly (less than 650 lines of Modelyze code), where only fundamental functional programming knowledge is needed. The domain expert is only required to master simple recursive functions together pattern matching on symbolic expressions.

The part of the DSL definition that makes especially use of dynamic typing is the run-time simulation semantics. When the DAE is numerically approximated using a standard DAE solver from the SUNDIALS [12] solver suite, we need to supply a function that evaluates the residual of the equation system (the difference between the right-hand-side and left-hand-side of the equation system). We provide the residual by interpreting the equation system (represented as a symbolic expression) within an `eval` (evaluation) function. We provide `eval` as a callback function to the solver library.

```
1  def eval(expr:<?>, yy:Varvals, yp:Varvals, ctime:Real) -> ? = {
2      match expr with
3      | time -> ctime
4      | der x ->
5          (match x with
6           | sym:Real -> eval(yp(x), yy, yp, ctime)
7           | _ -> error "Derivatives only allowed on unknowns")
8      | sym:Real -> eval(yy(expr), yy, yp, ctime)
9      | f e -> (eval(f,yy,yp,ctime)) (eval(e,yy,yp,ctime))
10     | lift v:? -> v
11     | _ ->  error "Unsupported model construct"
12 }
```

The `eval` function takes a symbolic expression `expr` of the dynamic symbolic type `<?>` and evaluates it to a value. Parameters `yy` and `yp` are higher order functions of type `Varvals` (alias for `<Real> -> <Real>`), returning the state variable and its derivative, respectively. Without dynamic types, it is difficult to assign types to this `eval` function. For example, the pattern variable `f` on line 9 can have different functional types, depending on the context. If gradual typing was not used and parameter `expr` has type `<Real>`, we would need to have different cases for different types. In such an example, code lines 9-10 can be replaced by the following code:

```
| (lift f:(Real -> Real -> Real)) e1 e2 ->
      f(eval(e1,yy,yp,ctime), eval(e1,yy,yp,ctime))
| (lift f:(Real -> Real)) e1 -> f(eval(e1,yy,yp,ctime))
| lift v:Real -> v
```

The problem with the latter static example is that functions in the equation system cannot have more than two parameters and these parameters must be of type `Real`. In the former `eval` function, which included dynamic types, such expressiveness problem does not exist. Hence, by utilizing gradual typing and selectively inserting dynamic types, we can make the expressive `eval` function typeable.

To avoid runtime errors, the `eval` function assumes that the symbolic expressions are well typed. Because symbolic expressions can be statically typed when introduced, but not when eliminated using pattern matching, the gradually typed semantics do not guarantee static type safety for code that decomposes symbolic expressions. Although the absence of static checks for type preservation of transformations may seem to be a weakness of our approach, such checking provides only a very limited verification of the overall correctness of the semantics. Many potential bugs may be the result of incorrect implementation of algorithms or wrong assumptions and interpretations of the domain. Such implementation and interpretation bugs cannot—in any case—be detected by a type checker. Because the domain expert is assumed to be highly skilled in the domain, he/she should be competent to generate comprehensive test suits, which checks the correctness of the semantics.

Hence, we contend that this approach gives a reasonable trade-off between good static error checking for the model engineer together with flexibility and expressiveness for the domain expert. Also, a good and flexible debug printing function has been of tremendous help during the implementation phases. The key importance of such a debug print function is to show symbolic expressions at the DSL notation level, i.e., that it outputs equations in infix order with real symbolic names.

## *4.2. Equation-Based Object-Oriented (EOO) DSL*

The second DSL is M-EOO, an equation-based object-oriented language with similar modeling capabilities as basic continuous-time modeling in Modelica [3]. The DSL reuses the DAE syntax and semantics from the M-DAE DSL and adds the capability of

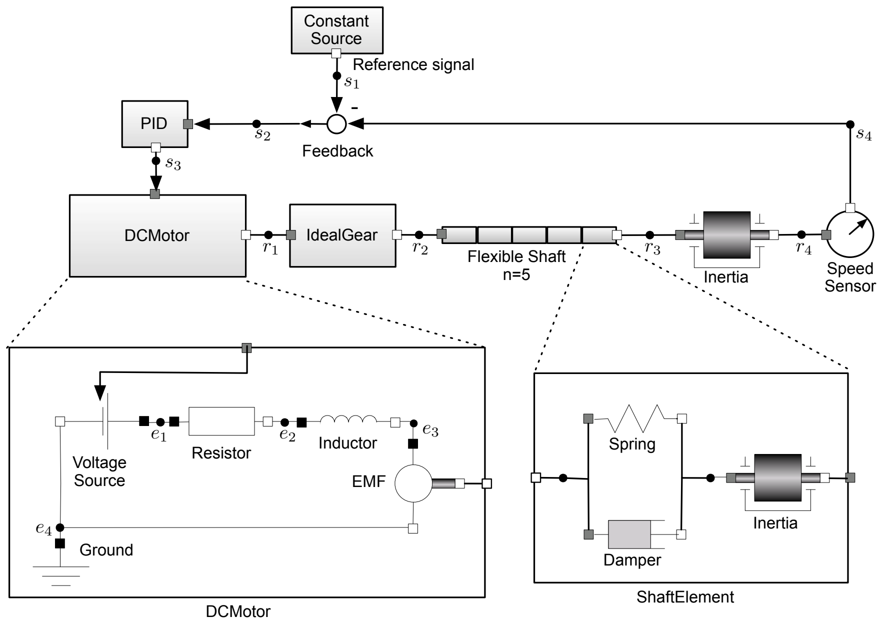

Figure 12: Example of a model in M-EOO representing a cyber-physical system containing a PID controller, an electrical DC Motor, and a rotational mechanical system.

defining hierarchical object-oriented models, meaning that the topology of the models have a direct correspondence to physical models.

Figure 12 depicts the structure of a model described in M-EOO. The model describes the dynamics of a mechatronic powertrain system, consisting of a direct current (DC) motor that drives an ideal gear, a flexible shaft, and an inertia. The rotational speed (angle velocity) is controlled by a feedback control loop using a PID [56] controller.

One benefit with block based modeling languages is to be able to compose submodels together to form new models. However, compared to Simulink models [57], EOO models are *acausal*, meaning that the direction of information flow between the model components is not determined at modeling time. For example, there is no direction of flow between the shaft and the inertia. Initially during simulation, the torque from the DC motor speeds up the inertia. However, when the inertia is rotating, its torque also affects the rotation of the shaft.

The acausal modeling capabilities of the DSL are achieved by translating the hierarchical model into a DAE in two main phases. In the first phase, the hierarchy of the EOO model is collapsed into a large equation system. This equation system contains—besides differential equations—information about the structure of the hierarchical model. This structured information is used in the second phase to generate additional unknowns and equations for the final equation system. Broman and Nilsson [58] describe the details of this process. In the rest of this section, we focus on the model engineer's and domain expert's implementation perspective, rather that the algorithmic perspective per se.

The following Modelyze source code[5] shows the concrete top level implementation of the CPS model outlined in Figure 12.

```
1  def CPS() = {
2      def s1, s2, s3, s4:Signal;
3      def r1, r2, r3, r4:Rotational;
4      ConstantSource(1, s1);
5      Feedback(s1, s4, s2);
6      PID(3, 0.7, 0.1, 10, s2, s3);
7      DCMotor(s3, r1);
8      IdealGear(4, r1, r2);
9      serialize(5, r2, r3, ShaftElement);
10     Inertia(0.3, r3, r4);
11     SpeedSensor(r4, s4);
12 }
```

Line two and three defines signal and mechanical rotational *nodes* (connection points in the component graph). All these nodes are typed symbols, e.g., `Rotational` is a symbolic data type. These nodes are then supplied to the various model components. This is the way a model engineer connects components together. For example, in line

[5]In the current prototype implementation, the model engineer uses a text based concrete syntax. However, we do not see any technical challenges of implementing a graphical GUI (similar to Modelica tools) where the engineer edits models graphically and these models are automatic translation to textual source code.

7 the `DCMotor` model is applied to signal `s3` and node `r1`. Models in M-EOO are normal Modelyze functions. Hence, applying a model to a node is a standard function application. Because nodes (symbols) are values, the symbolic lifting analysis does not lift these applications to symbolic expressions. Consequently, phase one of collapsing the model hierarchy comes for free directly from the host language, that is, during evaluation, nodes are substituted and function abstractions eliminated. Again, note that the underlying symbolic types of the host language do not affect the DSL experience for the model engineer.

On line 9, a recursive function `serialize` is used for creating 5 model components in series. The supplied shaft element is a higher-order model [59], a model containing equations is supplied to a generic function that combines model components in series (in this case a flexible shaft). Again, this is an example were the functionality comes directly from the host language's support for first class functions. We are in this example using dynamic types for getting polymorphism (elements with different node types). Hence, we loose in this case static type information. This limitation of the current approach can be improved by adding parametric polymorphism. Combining gradual typing with parametric polymorphism is solved by Ahmed *et al.* [60], and we regard it as future work to combine parametric polymorphism with typed symbolic expressions.

One level down in the model hierarchy,the `DCMotor` is defined as follows:

```
1  def DCMotor(V:Voltage,flange:Rotational) = {
2    def e1, e2, e3, e4:Electrical;
3    SignalVoltage(V, e1, e4);
4    Resistor(200, e1, e2);
5    Inductor(0.1, e2, e3);
6    EMF(1, e3, e4, flange);
7    Ground(e4);
8  }
```

Within the electrical domain, another node of type `Electrical` is used. The main benefit of defining different types of these nodes is that a model engineer gets early and precise error feedback. For example, if a node within the mechanical domain is supplied to the `Resistor` model component on line 4, we get the error message:

```
controlsys.moz 4:19-4:25 error: Illegal argument type.
Expected an argument of type 'Electrical'.
```

At the lowest level in the hierarchy, the differential equations are explicitly stated. For example, in the `Inductor` model

```
1  def Inductor(L:Real, p:Electrical, n:Electrical) = {
2    def i:Current;
3    def v:Voltage;
4    Branch i v p n;
5    L * i' = v;
6  }
```

line 5 shows the differential equation describing the behavior of the inductor.

A new construct added to M-EOO is the `Branch` construct (line 4). A *branch* is conceptually a path between two nodes through a component model. The branch encodes information about the model structure and is used in the second phase when generating new equations and unknowns. Example of equations that are generated during this phase are sum-to-zero equations for nodes, for example, obeying Kirchhoff's current law.

The model engineer—who is typically creating and composing EOO models—do not need to understand the underlying semantics of how equations and unknowns are generated from the `Branch` construct. Such details are handled by the domain expert. Without going into details of the actual algorithm, we can study how the `Branch` construct is defined. The `Branch` is a symbol and defined as

```
def Branch : Real -> Real -> ? -> ? -> Equations
```

The first two parameters represent (in the electrical domain) the current flowing *through* the model component and the voltage drop *across* the component. This approach is applicable in other physical domains (e.g., the mechanical domain), and consequently, the same branch can be used in these domains as well. The third and fourth parameters correspond to the connected nodes. These nodes can be of different types (e.g., `Electrical` or `Rotational`). We are therefore using dynamic types to make this polymorphic[6].

Adding the `Branch` symbol to M-EOO is an example that shows why symbolic types needs to be open. To review, the DSL M-DAE defined the constructs for adding initial values and equations.

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

In the M-EOO DSL, we extend the `Equation` data type, using the definition of `Branch`.

```
def Branch : Real -> Real -> ? -> ? -> Equations
```

If the `Equation` data type was closed, such an extension would not be possible. An equation system can now contain either `init` symbols (from M-DAE), `Branch` symbols (from M-EOO), or both. To make generic traversal and transformation functions safe, it is important that additional symbols do not change existing functions' behavior. For example, recall the function `getInitValues` from M-DAE.

```
1  def getInitValues(eqs:Equations, acc:InitVals) -> InitVals = {
2      match eqs with
3      | e1 ; e2 -> getInitValues(e2, getInitValues(e1,acc))
4      | init x v -> Map.add x v acc
5      | _ -> acc
6  }
```

This function searches and collects initial value definitions in an equation system. Note how line 3 performs the generic traversal for all equations, how line 4 only considers cases where symbol `init` is a constructor, and how the default case (line 5) returns

[6]Extending the language with support for generic data types would, in this case, increase static checking without loosing expressiveness. Such an extension is planned as future work.

the accumulated result. Even though another DSL extends the equation data type—in the M-EOO case the the `Branch` construct—the behavior of function `getInitValues` does not need to be changed.

### 4.3. Hybridchart (HC) DSL

The third DSL M-HC extends M-DAE by adding language constructs for defining hierarchical state machines where each state (called mode) consists of DAEs. Language M-HC introduces structurally dynamic systems, systems where the structure of the equation system changes during run-time. We call this DSL construct *hybridcharts* because it resembles the hierarchy aspect of Harel's statecharts [61], but it does not include concurrency or communication. In contrast to Henzinger's hybrid automata [62], the continuous-time behavior is described by DAEs instead of ODEs and there are no invariants within the modes.

Figure 13 shows an example of a model implemented in M-HC. The model shows a hybridchart consisting of two modes: `Pendulum` (lines 9-15) and `BouncingBall` (lines 16-21). The model is initiated to mode `Pendulum` (line 8) so that the model `PendulumExt` (defined in Section 4.1) is instantiated. When the time is greater than 3.2 seconds, a transition is made to mode `BouncingBall` (lines 13-14). No actions are taken during the transition. Note that the state variables `x` and `y` are declared outside the scope of the hybridchart, making the states of the variables to be trans-

```
 1  def BreakingPendulum(m:Real, l:Real, angle:Real) = {
 2      def x,y:Position;
 3      def Pendulum, BouncingBall:Mode;
 4      init x (l*sin(angle));
 5      init y (-l*cos(angle));
 6      probe("y") = y;
 7
 8      hybridchart initmode Pendulum {
 9          mode Pendulum {
10              def T:Force;
11              probe("T") = T;
12              PendulumExt(m, l, T, x, y);
13              transition BouncingBall
14                  when (time >= 3.2) action nothing;
15          };
16          mode BouncingBall {
17              x'' = 0;
18              -m*g = m*y'';
19              transition BouncingBall
20                  when (y <= -4) action (y' <-  y' * -0.7);
21          };
22      };
23  }
```

Figure 13: Source of the breaking bouncing pendulum.

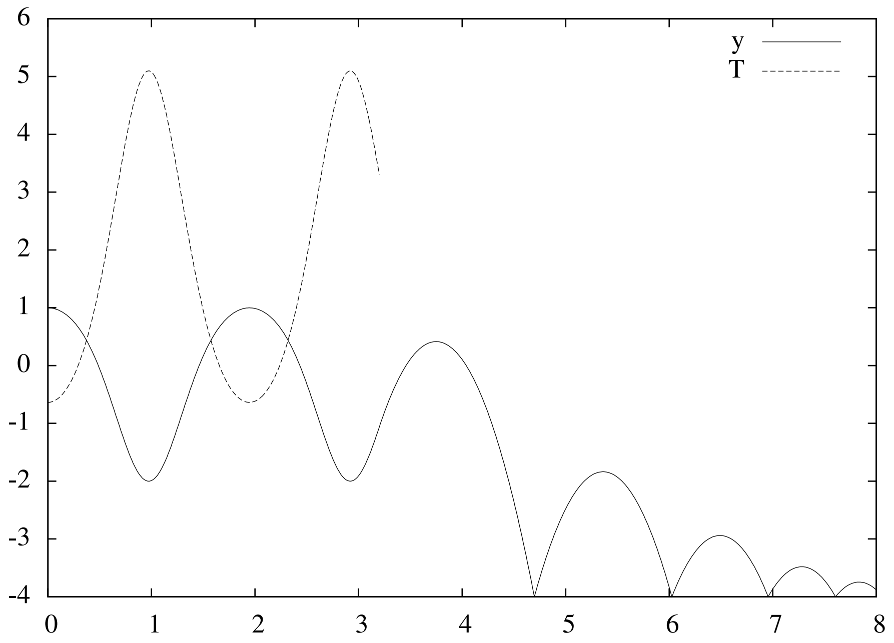

Figure 14: Plot of the breaking pendulum that, after breaking, starts to bounce.

ferred to the next mode. However, in `BouncingBall`, acceleration of `x` is set to zero, so that the ball keeps on flying at the same velocity as when the `Pendulum` broke. In the `BouncingBall` mode, the ball of the old pendulum starts to bounce, which is modeled using a mode self transition (lines 19-20). Figure 14 shows a simulation of `BreakingPendulum`.

The first difference of M-HC, compared to the previously presented DSLs, is that a fair amount of new syntax is introduced. These syntax elements are all introduced using typed symbols, whereas the application to the symbol is expressed using juxtaposition, that is, arguments are separated with spaces and not by parenthesis and commas. For example, the following definitions

```
type Mode
type Action
def  action:Action
type When
def  when:When
def transition : Mode -> When -> Bool -> Action ->
                 Equations -> Equations
```

define the syntax for the transition construct. Note how we use symbolic data types for creating the new symbols `when` and `action` to behave as new keywords in the DSL. Again, it is the combination of using symbol lifting analysis together with function application (juxtapostion without the need for parenthesis) that is the underlying mechanism that enables such keyword extensions. When the new keywords (represented as symbols) are used as arguments, the application is lifted into symbol applications.

Also, because the `transition` symbol is typed, the type system is used to detect syntactic errors for the new DSL constructs. For example, if the user on line 14 forgets to write the `when` keyword, the following error message is supplied:

```
breakingpendulum.moz 14:18-14:29 error: Illegal argument type.
Expected an argument of type 'When'.
```

The runtime simulation semantics for M-HC are significantly different from M-DAE. The M-HC DSL makes use of more solver functionality, including zero-crossing detection for discovering events (e.g., when the ball hits the ground). Hence, the reuse from M-DAE is limited to certain symbolic elaboration algorithms, e.g., Pantelides and symbolic differentiation.

### *4.4. Evaluation Summary*

This qualitative evaluation illustrated the development of three equation-based DSLs. From the model engineer's point of view, we have shown that the symbol lifting analysis gives in general a seamless integration between the host and the DSL language. The exception is that the user needs sometimes to be exposed to the notation of symbolic types, which is mostly the case for the domain expert.

Model errors can be reported statically, both in the case of illegal equation definition and also for symbolic expressions used for defining new DSL syntax. However, we also explained the limitation that certain errors—such as over- or under-determined systems of equations—cannot be captured by the type system.

The main strength of gradual typing and fine-grained insertion of dynamic types is optional expressiveness freedom for the domain expert. The price paid for inserting dynamic types is that we loose type preserving transformations. We contend that the extra price of loosing some static checking is motivated by the domain expert to gain more expressiveness when needed.

Finally, we also showed that open types are necessary to enable DSL reuse. Functions of the reused DSL can be used as long as they have a well defined behavior in the default case.

## 5. Related Work

In this section we discuss related work in the areas of implementing DSLs, mixing static and dynamic types, representing code and data types, and EOO languages.

### *5.1. Implementing Domain-Specific Languages*

As Mernik *et al.* [24] points out, DSL implementation is hard because it requires both domain knowledge and expert knowledge in language development, but few people are expert in both. DSL implementation has a long history, but it is not until lately that DSL implementation strategies has been categorized and compared. In one recent study, Kosar *et al.*[63] compare several implementation approaches. The most common approaches are based on compiler construction, preprocessing, and embedding.

Our approach is based on the latter and we therefore first discuss how this approach compares with the former two.

*Compiler construction.* A traditional approach to implementing a DSL is to create a compiler from scratch including parsing, static semantics analysis, and transformation. To partially overcome the high development costs involved in developing new compilers, compiler construction tools can be used. For example, Hedin, Magnusson, and Ekman [64, 65] propose the JastAdd system, a Java based aspect-oriented compiler construction framework. Åkesson *et al.* [66] utilize JastAdd to implement an open source environment for the EOO language Modelica. Petterson's [67] RML language and Pop and Fritzson's [68] MetaModelica language are used for implementing the OpenModelica [69] environment. These approaches are well suited for implementations of existing equation-based languages that require full flexibility of the syntax and semantics. However, compared to our approach, it demands that the domain expert have significant compiler experience.

In a *preprocessing* approach, DSL constructs are translated into an existing base language [24]. One of the first preprocessing mechanisms for defining DSLs was LISP's macro system [37]. Although macros are very flexible and expressive, a problem is that error reporting is not performed until after macro expansion. Template metaprogramming is another preprocessing technique, advocated by Veldhuizen [38] for C++. Templates in C++ are evaluated at compile time and can be used as a preprocessing stage to generate specialized and more efficient programs. Tratt [70] presents a homogeneous metaprogramming approach where arbitrary syntax extensions can be expressed within special DSL blocks. Sheard and Peyton Jones [39] work on Template Haskell is an extension to Haskell with support for compile time metaprogamming. Template Haskell can both be seen as a template environment and a macro expansion system.

Both Template Haskell and Modelyze support name generation using a gensym style construct. Template Haskell uses monads [31] for name generation, whereas name (symbol) creation in Modelyze is part of the core language. Also, in Modelyze, typed symbol generation is used both by the model engineer (at the DSL level) and by the domain expert (when implementing the DSL). A third approach to preprocessing is source-to-source transformation. Cordy's [71] TXL language is an example of a source translation language design for experimenting with various programming language notations. Stratego/XP [40] is another program transformation language and tool. Both TXL and Stratego support the definition of concrete syntax and rewriting rules for program transformation. Bravenboer *et al.* [72] describes an approach called METABORG, where concrete syntax for domain abstractions can be embedded into an existing host language and the embedded code is translated into the host language. METABORG uses Stratego for translation, but the approach is not coupled with a specific host language. The Jakarta Tool Suite (JTS) by Batory *et al.* [73] consists of several tools for extending languages (particular Java) with new domain-specific constructs. In contrast to these transformation approaches, Modelyze does not support arbitrary syntax extension using grammar definitions. Instead, extensions are described using symbolic types, which enables type checking of new syntactic DSL constructs. This in turn provides model (DSL) level error checking.

In *domain-specific embedded languages* (DSEL) [18], the infrastructure of the DSL is inherited from a host language and includes both concrete syntax and semantic for the programming constructs. Haskell has extensively been used as a host language for DSELs. Example of DSELs implemented in Haskell include Fran [19], FRP [20], FHM [21], Lava [22], and Paradise [23]. A DSEL, as advocated by Hudak [74], is based on pure embedding, meaning that there is no macro expansion or source-to-source transformation. Instead, the embedding is based on language features such as higher-order functions, polymorphism, lazy evaluation, and type classes [74]. Modelyze uses the DSEL approach and includes both higher-order functions and polymorphism, but not lazy evaluation or type classes [32]. The polymorphic behavior is achieved using dynamic types and built-in types for pure maps, sets, and lists. Instead of using type classes for overloading functions and algebraic data structures, we use the simpler approach of automatic lifting into symbolic expressions. The main benefit with our approach compared to the Haskell DSEL approach is its simplicity for domain experts; we do not assume they are full-fledged functional programmers. That is, instead of learning several advanced language concepts such as type classes [32], monads [31], and GADTs [33], the domain expert only has to learn one concept, which is typed symbolic expressions.

## *5.2. Mixing Static and Dynamic Types*

Mixing static and dynamic types in a programming language is not a new idea. Several dynamically typed programming languages have support for stating explicit type annotations, such as Common LISP [37], Dylan [75], Cecil [76], Boo [77], Meijer and Drayton's [78] extensions to Visual Basic.NET and C$\#$, and Bracha's Strongtalk [79, 80]. These languages use type annotations to improve static checking and to increase execution performance, but they do not give any guarantees that a fully typed program prevents type errors and type exceptions at run-time.

In contrast, our work on Modelyze's semantics are based on Siek and Taha's [28, 29] approach named gradual typing. This approach gives the guarantee that fully typed programs do not produce run-time type exceptions. In this paper, we follow their approach of explicitly defining the consistency relation as well as defining the semantics using cast insertion translation. Instead of extending the core concepts of gradual typing with references [28] or subtyping [29], this paper studies how the approach can be extended with symbolic types. Ahmed *et al.* [60] develops the *polymorphic blame calculus*, which is an extension of Wadler and Findler's [81] blame calculus, where the former combines parametric polymorphism, static, and dynamic typing. Based on this work, we also believe that it is practically possible to extend Modelyze with parametric polymorphism.

Static and dynamic typing can be combined using *soft typing* [82]. In such approach, the static type checker does not reject programs, but instead reports compile time warning messages and inserts run-time checks in places that can contain potential errors. Cartwright and Fagan [82], Flanagan and Felleisen [83], Aiken, Wimmers and Lakshman [84], and Henglein and Rehof [85, 86] propose variants of such static analysis that can be used for catching bugs in dynamic programming languages. In contrast to our approach of using gradual typing, the programmer does not control which portions of a program are dynamically or statically typed.

Abadi *et al.* [87] develop a method to have dynamic types in a statically typed language. They introduce a type Dynamic whose values are pairs of a value and a type tag. These values can be eliminated using a typecase construct. Compared to our dynamic type ?, Abadi *et al.* require that the programmer explicitly add introduction and elimination expressions for type Dynamic, whereas gradual typing automates this.

Several works address the problem of *interoperability*, that is, when values are passed between different languages with different notion of types. Gray, Findler, and Flatt [88] develop an interoperability semantics between Java and Scheme, where Java is extended with a type Dynamic. This work originally inspired the design of Siek and Taha's gradual typing [29]. Matthews and Findler [89] introduce an operational semantics for interoperability between multi-program languages, using an approach of mixing dynamic and statically types similar to type Dynamic by Abadi *et al.* Tobin-Hochstadt and Fellisen [90] show how inter-language mitigation from a dynamically typed to a statically typed language can be performed on a module basis. This module level mixing of types is the basis for Typed Scheme [91], an explicitly typed extension to PLT Scheme, that, in turn, recently was renamed to Racket [17]. The key difference to our gradual typing approach is that our mixing of types is at a much finer level of granularity. This expression level control of gradual typing is vital to support our DSEL approach, such that the domain expert can "escape" out of static typing only when more expressiveness is needed.

There are several works that are closely related to gradual typing. Knowles and Flanagan [92] develop hybrid type checking using static checking together with casts for run-time checks, similar to our approach. A notable difference is that hybrid type checking is based on contract types, whereas gradual typing uses consistency checking. Ou *et al.* [93] combine dependent types with a simply typed language. Similar to the gradual approach, a stronger and weaker type system can be mixed in one language and coercions are inserted for the latter. Thatte's quasi-static typing [47] is related to gradual typing, with the main difference that quasi-static typing is relying on subtyping with the top element representing the dynamic type. However, Siek and Taha [28] show that quasi-static typing does not catch all type errors, even if a term is fully typed. Riely and Hennessy [94] present a partially typed semantics for a distributed $\pi$-calculus, called D$\pi$. Similar to quasi-static typing, the type system of D$\pi$ relies on subtyping, with the difference that the dynamic type (called lbad) is the bottom element.

Combining static and dynamic typing is also applied to popular dynamic scripting languages. Furr *et al.* [95] introduce DRuby that extends the Ruby language with annotations and the possibility to infer static types. Anderson and Drossopoulou [96] develop the language BabyJ$^T$, a formalization of a subset of JavaScript that can mix dynamic and static types. Similar to our approach, they introduce dynamic type *, and types can gradually be added to a program. The main difference is that BabyJ$^T$ is a nominally typed object-based language, whereas Modelyze has a functional core extended with symbolic types.

Groski *et al.* [97] develops a language SAGE that performs hybrid type checking, including type Dynamic, refinement types, and first-class types. Subtyping is used for upcasting a type to Dynamic, and implicit downcasts are performed using run-time checks. Writstad *et al.* [98] introduce Thorn, where statically typed and dynamically typed code are integrated using *like types*. The separation of dynamic types, like types, and concrete types is used to enable compiler optimizations in statically typed code fragments. Because the current prototype of Modelyze is interpreted, we have not yet evaluated the performance aspect. It is unclear how typed symbolic expression and symbol lifting analysis would interact with like types, but to enable efficient compiler optimization, it is an interesting possibility for future work.

## *5.3. Representing Code and Data Types*

Expressions representing program code can be dynamically typed as in LISP [26] or Mathematica [99]. At the other end, some language designs always assign static types to expressions that represent. Taha and Sheard [27, 30] introduce MetaML that have expressions of the type <T>, meaning the code of type `T`. The code type of MetaML is similar to our symbolic type in that an arbitrary type can be given as a code type and that static checking is performed on the mix of code and non code types. One difference is that we combine symbolic types with the dynamic type `?`, making our approach not statically but gradually typed. Another difference is, as explained in Section 2, that quasi-quoting is explicit in MetaML, whereas it is implicit in Modelyze due to symbol lifting analysis. Moreover, MetaML and its related implementation MetaOCaml [100] are type safe multi-stage programming languages used for *extensional programming*, that is, for generating and executing specialized code. In contrast, Modelyze's `case` construct can be seen as a light form of *intensional analysis* [101], meaning that code can analyzed and transformed. We say that Modelyze supports a *light* form of intensional analysis because only lifted symbolic expressions can be analyzed. In contrast to Template Haskell [39], for example, function bodies cannot be analyzed in Modelyze. This design is intentional because we do not want to make the internal representation of code available to the programmer. In Modelyze, symbolic expressions are transformed at runtime, whereas in Template Haskell they are transformed at compile time.

Neverov and Roe [102] develop the Metaphor language, a metaprogramming extension to a subset of C#. The language supports both intensional and extensional metaprogramming, but no type safety proof is provided. Stump [103] presents ARCHON, a metaprogramming language with support for intensional analysis. ARCHON allows opening lambda abstraction and execution of open terms with free variables. In contrast to Modelyze, ARCHON is untyped and thus does not fulfil our objective of being able to give static error feedback for the model engineer.

Binding time analysis (BTA) in partial evaluation [42] is a technique to determine, given static and dynamic input, which operations that can be performed statically. Henglein [41] shows how BTA can be treated as a type inference problem in a two-level $\lambda$-calculus. In contrast to BTA, our symbol lifting analysis determines which values *cannot* be evaluated at runtime, and lifts these expressions into data in form of symbolic expressions.

The only way to express user defined data types in Modelyze is by using symbolic types. Syntactically, this approach has similarities to Löh and Hinze's [104] open data types, which is implemented as an extension to Haskell. An open data type is defined by explicitly stating that it is open

```
open data Exp::*
```

In Modelyze, we would define a symbolic data type,

```
type Exp
```

which is by definition open, i.e., we do not support closed data types. In their extension, a constructor can then at a later point be added, e.g.,

```
Str :: String -> Exp
```

Constructors are in Modelyze represented as symbols, so the constructor `Str` can be define as a symbol

```
def Str : String -> Expr
```

Löh and Hinze's also define open functions, which can be extended at a later point, which is not yet supported by Modelyze. Although arguably a pleasant feature, we have not seen a strong use case in the context of equation-based languages. The reason is that functions translating or analysing symbolic expressions typically are implicitly closed by providing default case for unknown symbolic expressions. Millstein, Bleckner and Chambers's [105] extension to ML supports open datatypes and open functions with modular typechecking, as well as exhaustive checking of pattern matching. Modelyze is simpler and limited in this regard because modules are not separately compiled and it is not checked if patterns are exhaustive.

*Generic queries and transformations.* In a series of "scrap your boilplate" papers, Lammel and Peyton Jones [106, 107, 108] show how boilplate code can be avoided when performing queries and transformation of complex recursive data structures. Their approach exploits Haskell's type-class mechanism together with two language extensions: rank-2 types and type-safe casts. Our approach is simpler due to the uniform structure of symbolic expressions and the use of dynamic types. For example, the `getUnknowns` example in Section 2.3 illustrates how dynamic types of symbolic expressions are used when traversing the data structure. Axelsson [109] presents the Syntactic library, a Haskell library for defining modular abstract syntax trees and generic traversal of such ASTs. The library enables type safe transformations, open data types, and extensible functions, with the cost of heavy type-level programming, requiring that the domain expert has significant Haskell knowledge. Jay [110] introduces several variants of the *pattern calculus*, a fundamental calculus for computation with pattern matching function. The pattern calculus has, similar to Modelyze, a uniform data structure based on an application term. In contrast, the pattern calculus use the same application term for both representing function application and the data structure of a pair (in our case symbolic application). The pattern calculus can avoid boilplate code similarly to Modelyze by performing traversals on the uniform data structure. A difference is that constructors in the pattern calculus are fixed at compile time, whereas in Modelyze, symbols are created dynamically at runtime.

Generalized abstract data type (GADT) [33], also known as guarded recursive data type [34], first-class phantom type [35], and equality-qualified type [36], is a generalization of algebraic data type. GADTs can be used in a DSL to ensure creation of well typed terms, as well as for creating evaluators that are type preserving. Similarly, symbolic types with static types ensure that only well typed terms can be introduced, but elimination of symbolic expressions using `case` expressions and introduction using type <?> does not guarantee type preserving translations. We motivate the usefulness of our approach as a trade off between simplicity and static guarantees. Symbolic type is one simple concept giving static guarantees for the model engineer, but removes some guarantees for the domain expert in exchange for simplicity and flexibility.

### *5.4. Equation-based Modeling Languages*

Modelyze is an extension of our previous work on the Modeling Kernel Language (MKL), developed in the first author's Ph.D. thesis [25]. The main difference to MKL's semantics is that Modelyze is based on full gradual typing, whereas MKL only has dynamic types for symbolic types. The syntax also differs, where Modelyze has adopted a syntax that we believe is easier to adopt for engineers with limited functional programming background. Broman and Fritzson [59] introduce higher-order acausal models as a programming abstraction in MKL, that is, models being first class in an EOO language. This work was based on the flow lambda calculus [111], a functional approach to encode EOO languages without metaprogramming capabilities.

Modelica [2, 3] is currently state-of-the-art regarding EOO languages. It has grown to be a very large and complex language, with frequent releases of new language versions. This fact was one of the motivating factors for developing Modelyze. Modelica is an industrial strength language, supporting numerous language features that are not captured by the simple EOO DSL presented in Section 4. However, at the same time, we show the DSL of hybridcharts that expresses models that cannot be encoded in Modelica. Both Nytsch Geusen's *et al.* [112] Mosilab tool and Zimmer's [113, 114] Sol language support Modelica like language extensions for structurally dynamic systems. Åkesson *et al.* [115] develop another Modelica extension, where models are used for optimization instead of simulation. Compared to Modelyze, which is a host language for embedding equation-based DSLs, Sol, Mosilab and Optimica are DSLs implemented as new interpreters and compilers.

The main purpose of *hardware description languages* (HDLs), such as VHDL and Verilog, is to define digital electrical circuits. However, *analog and mixed signal* (AMS) extensions, such as VHDL-AMS [5, 6] and Verilog-AMS [4] include similar semantics as EOO langauges. Both VHDL-AMS and Verilog-AMS are using variants of node-based approaches for the semantics of connecting model components. This is similar to the node based connection semantics [58] exemplified in this paper. Other functional HDLs, such as Bjesse's *et al.* [22] Lava and Axelsson's *et al.* [116] Wired are both examples of embedded DSLs, but do not include equation-based modeling as presented in this paper.

Functional hybrid modeling (FHM) [117] is a paradigm combining acausal modeling (à la EOO) with functional programming. It can be seen as a generalization of functional hybrid programming (FRP) [20]. Giorgidze and Nilsson [118] develop Hydra, a DSL based on FHM, which is embedded in Haskell. The central model abstraction in FHM is *signal relation*, which is similar to the function based model abstractions in the DSL embedded in Modelyze, but parameterized on signals and not nodes.

Detecting errors statically in an EOO language is hard. Detecting and isolating over- and under-constrained systems of equations in EOO langauges have been studied by Bunus and Fritzson [44], Broman, Furic, and Fritzson [43], Ohlsson *et al.* [119], and Nilsson [45]. Physical unit checking for Modelica has been investigated by Aronsson and Broman [120, 121], and by as by Mattsson and Elmqvist [122]. Introducing any of these error detection mechanisms in Modelyze require access to the model structure before elaboration or operations on types. Currently, this is not possible and we regard it as future work to be able to express this within a library.

Finally, Ptolemy II [123] is a heterogeneous coordination software system for connecting component of different models of computation (MoC). A Ptolemy director can be viewed as a DSL, implementing a specific MoC. Although fundamentally different, both Modelyze and Ptolemy are host systems for implementing various MoCs.

## 6. Conclusions

In this paper we introduce a new host language called Modelyze, designed for embedding equation-based DSLs. The language is based on gradual typing, making it possible to mix static and dynamic typing at a fine grained level of abstraction. The main novelty of Modelyze from a programming language perspective is the notion of typed symbolic expressions together with symbolic types. To provide seamless integration between the host language and the DSL, we introduce symbolic lifting analysis, which is performed in conjunction with type checking. We formalize a core of the language and prove type safety. We show that one of the main strengths of our approach is its simplicity; the domain expert only has to learn one new concept—typed symbolic expressions.

Although Modelyze has only been evaluated for equation-based languages, it is interesting to investigate if it can be used in other domains as well. Future work also includes how the presented approach can be combined with parametric polymorphism and extensional metaprogramming to enable more expressiveness without loosing static typing and to increasing runtime performance, respectively.

**Acknowledgements**

This research work was funded by the Swedish Research Council (VR), by the ITEA2 OPENPROD project, and by Linköping University under the ELLIIT project. Additional funding was provided by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley. The second author's work was funded by NSF CAREER Award 0846121.

The authors would like to thank Peter Fritzson, Thomas Schön, Henrik Nilsson, Walid Taha, Sibylle Schupp, and Johan Åkesson for comments and suggests on earlier drafts of this work. Our special thanks go to Michael Zimmer for his painstaking effort of reading the final draft.

## Appendix

### Free variables $\boxed{FV(e)}$

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x\!:\!\tau.e) & = & FV(e) \setminus \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(c) & = & \emptyset \\
FV(\texttt{error}\ ) & = & \emptyset \\
FV(\nu(\tau)) & = & \emptyset \\
FV(\texttt{case}(e_1, p, e_2, e_3)) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(e_1 @ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\texttt{lift}\ e\!:\!\tau) & = & FV(e) \\
FV(\langle \tau_2 \Leftarrow \tau_1 \rangle e) & = & FV(e) \\
FV(s\!:\!\tau) & = & \emptyset
\end{array}
$$

### Substitiution $\boxed{[x \mapsto e]e}$

$$
\begin{array}{lcll}
[x \mapsto e]x & = & e & \\
[x \mapsto e]y & = & y & \text{if } x \neq y \\
[x \mapsto e]\lambda y\!:\!\tau.e_1 & = & \lambda y\!:\!\tau.[x \mapsto e]e_1 & \text{if } x \neq y \text{ and } y \notin FV(e) \\
[x \mapsto e]e_1\ e_2 & = & [x \mapsto e]e_1\ [x \mapsto e]e_2 & \\
[x \mapsto e]c & = & c & \\
[x \mapsto e]\texttt{error} & = & \texttt{error} & \\
[x \mapsto e]\nu(\tau) & = & \nu(\tau) & \\
[x \mapsto e]\texttt{case}(e_1, p, e_2, e_3) & = & \texttt{case}([x \mapsto e]e_1, p, [x \mapsto e]e_2, [x \mapsto e]e_3) & \\
[x \mapsto e]e_1 @ e_2 & = & [x \mapsto e]e_1 @ [x \mapsto e]e_2 & \\
[x \mapsto e]\texttt{lift}\ e_1\!:\!\tau & = & \texttt{lift}\ [x \mapsto e]e_1\!:\!\tau & \\
[x \mapsto e]\langle \tau_2 \Leftarrow \tau_1 \rangle e_1 & = & \langle \tau_2 \Leftarrow \tau_1 \rangle ([x \mapsto e]e_1) & \\
[x \mapsto e]s\!:\!\tau & = & s\!:\!\tau &
\end{array}
$$

## References

[1] E. A. Lee, CPS foundations, in: Proceedings of the 47th Design Automation Conference, DAC '10, ACM Press, New York, USA, 2010, pp. 737–742.

[2] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley-IEEE Press, New York, USA, 2004.

[3] Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.3, Modelica Association, 2012. Available from: http://www.modelica.org.

[4] Accellera Organization, Verilog-AMS Language Reference Manual - Analog & Mixed-Signal Extensions to Verilog HDL Version 2.3.1, 2009.

[5] E. Christen, K. Bakalar, VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications, IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 46 (1999) 1263–1272.

[6] IEEE Std 1076.1-2007, IEEE Standard VHDL Analog and Mixed-Signal Extensions, IEEE Press, 2007.

[7] P. Kunkel, V. Mehrmann, Differential-Algebraic Equations Analysis and Numerical Solution, European Mathematical Society, 2006.

[8] Iain S. Duff, On Algorithms for Obtaining a Maximum Transversal, ACM Transactions on Mathematical Software 7 (1981) 315–330.

[9] C. C. Pantelides, The Consistent Initialization of Differential-Algebraic Systems, SIAM Journal on Scientific and Statistical Computing 9 (1988) 213–231.

[10] S. E. Mattsson, G. Söderlind, Index reduction in differential-algebraic equations using dummy derivatives, SIAM Journal on Scientific Computing 14 (1993) 677–692.

[11] H. Elmqvist, M. Otter, Methods for Tearing Systems of Equations in Object-Oriented Modelling, in: Proceedings ESM'94 European Simulation Multiconference, pp. 326–332.

[12] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, C. S. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM Transactions on Mathematical Software 31 (2005) 363–396.

[13] H. Elmqvist, S. E. Mattsson, M. Otter, Modelica - A Language for Physical System Modeling, Visualization and Interaction, in: Proceedings of the IEEE International Symposium on Computer Aided Control System Design.

[14] J. Batteh, M. Tiller, C. Newman, Simulation of Engine Systems in Modelica, in: Proceedings of the 3rd International Modelica Conference, Linköping, Sweden, pp. 139–148.

[15] G. Hirzinger, J. Bals, M. Otter, J. Stelter, The DLR-KUKA success story: robotics research improves industrial robots, IEEE Robotics & Automation Magazine 12 (2005) 16–23.

[16] G. L. Steele, Growing a Language, Higher-Order and Symbolic Computation 12 (1999) 221–236. Keynote at OOPSLA 1998.

[17] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen, Languages as libraries, in: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, ACM Press, New York, USA, 2011, pp. 132–141.

[18] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys (1996).

[19] C. Elliott, P. Hudak, Functional reactive animation, in: Proceedings of the second ACM SIGPLAN international conference on Functional programming, ICFP '97, ACM, New York, NY, USA, 1997, pp. 263–273.

[20] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, ACM Press, New York, USA, 2000, pp. 242–252.

[21] G. Giorgidze, H. Nilsson, Embedding a functional hybrid modelling language in Haskell, in: Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages.

[22] P. Bjesse, K. Claessen, M. Sheeran, S. Singh, Lava: hardware design in Haskell, in: Proceedings of the third ACM SIGPLAN international conference on Functional programming, ACM Press, New York, USA, 1998, pp. 174–184.

[23] L. Augustsson, H. Mansell, G. Sittampalam, Paradise: a two-stage dsl embedded in haskell, in: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming, ICFP '08, ACM, New York, NY, USA, 2008, pp. 225–228.

[24] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (2005) 316–344.

[25] D. Broman, Meta-Languages and Semantics for Equation-Based Modeling and Simulation, Ph.D. thesis, Department of Computer and Information Science, Linköping University, Sweden, 2010.

[26] G. L. Steele, Common LISP. The Language, Digital Press, 2nd edition, 1990.

[27] W. Taha, T. Sheard, Multi-stage programming with explicit annotations, in: PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, ACM Press, New York, USA, 1997, pp. 203–217.

[28] J. Siek, W. Taha, Gradual typing for functional languages, in: In: Scheme and Functional Programming Workshop.

[29] J. Siek, W. Taha, Gradual Typing for Objects, in: Proceedings of the 21st European conference on ECOOP 2007: Object-Oriented Programming, volume 4609 of *LNCS*, Springer-Verlag, 2007, pp. 2–27.

[30] W. Taha, T. Sheard, MetaML and multi-stage programming with explicit annotations, Theoretical Computer Science 248 (2000) 211–242.

[31] P. Wadler, Comprehending monads, Mathematical Structures in Computer Science 2 (1992) 461–493.

[32] P. Wadler, S. Blott, How to make ad-hoc polymorphism less ad hoc, in: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '89, ACM, New York, NY, USA, 1989, pp. 60–76.

[33] S. Peyton Jones, D. Vytiniotis, S. Weirich, G. Washburn, Simple unification-based type inference for gadts, in: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, ACM Press, New York, USA, 2006, pp. 50–61.

[34] H. Xi, C. Chen, G. Chen, Guarded recursive datatype constructors, in: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, New York,

USA, 2003, pp. 224–235.
[35] J. Cheney, H. Ralf, First-Class Phantom Types, CISTR TR2003-1901, Cornell University, 2003.
[36] T. Sheard, E. Pasalic, Meta-programming With Built-in Type Equality, Electronic Notes in Theoretical Computer Science 199 (2008) 49–65.
[37] G. L. Steele, Jr., An overview of common lisp, in: Proceedings of the 1982 ACM symposium on LISP and functional programming, LFP '82, ACM, New York, NY, USA, 1982, pp. 98–107.
[38] T. Veldhuizen, Using C++ template metaprograms, C++ Report 7 (1995) 36–43.
[39] T. Sheard, S. P. Jones, Template Meta-programming for Haskell, in: Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, ACM Press, New York, USA, 2002, pp. 1–16.
[40] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. A language and toolset for program transformation, Science of Computer Programming 72 (2008) 52–70.
[41] F. Henglein, Efficient type inference for higher-order binding-time analysis, in: Functional Programming Languages and Computer Architecture, volume 523 of *LNCS*, Springer-Verlag, 1991, pp. 448–472.
[42] N. D. Jones, An introduction to partial evaluation, ACM Computing Surveys 28 (1996) 480–503.
[43] D. Broman, K. Nyström, P. Fritzson, Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta, in: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06), ACM Press, Portland, Oregon, USA, 2006, pp. 151–160.
[44] P. Bunus, P. Fritzson, Automated Static Analysis of Equation-Based Components, SIMULATION 80 (2004) 321–245.
[45] H. Nilsson, Type-Based Structural Analysis for Modular Systems of Equations, in: Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools, LiU Electronic Press, Paphos, Cyprus, 2008, pp. 71–81.
[46] S. Peyton Jones, The Implementation of Functional Programming Languages, Prentice Hall, 1987.
[47] S. Thatte, Quasi-static typing, in: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '90, ACM, New York, NY, USA, 1990, pp. 367–381.
[48] M. Felleisen, R. B. Finlder, M. Flatt, Semantics Engineering with PLT Redex, MIT Press, 2009.
[49] G. D. Plotkin, A Structural Approach to Operational Semantics, Technical Report, Department of Computer Science, University of Aarhus, 1981.
[50] J. G. Siek, P. Wadler, Threesomes, with and without blame, in: Proceedings for the 1st workshop on Script to Program Evolution, STOP '09, ACM, New York, NY, USA, 2009, pp. 34–46.
[51] C. Dimoulas, R. B. Findler, C. Flanagan, M. Felleisen, Correct blame for contracts: no more scapegoating, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, ACM, New York, NY, USA, 2011, pp. 215–226.
[52] J. G. Siek, R. Garcia, W. Taha, Exploring the design space of higher-order casts, in: European Symposium on Programming.
[53] A. K. Wright, M. Felleisen, A Syntactic Approach to Type Soundness, Information and Computation 115 (1994) 38–94.
[54] B. C. Pierce, Types and Programming Languages, The MIT Press, 2002.
[55] INRIA, The Caml language: Home, 2012. http://caml.inria.fr/ [Last accessed: June 29, 2012].
[56] K. J. Åström, T. Hägglund, Advanced PID control, ISA-The Instrumentation, Systems, and Automation Society, 2006.
[57] MathWorks, The Mathworks - Simulink - Simulation and Model-Based Design, 2012. http://www.mathworks.com/products/simulink/ [Last accessed: June 7, 2012].
[58] D. Broman, H. Nilsson, Node-Based Connection Semantics for Equation-Based Object-Oriented Modeling Languages , in: Proceedings of the Fourteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2012), volume 7149 of *LNCS*, Springer-Verlag, 2012, pp. 258–272.
[59] D. Broman, P. Fritzson, Higher-Order Acausal Models, Simulation News Europe 19 (2009) 5–16.
[60] A. Ahmed, R. B. Findler, J. G. Siek, P. Wadler, Blame for all, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, ACM, New York, NY, USA, 2011, pp. 201–214.
[61] D. Harel, Statecharts: A visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.
[62] T. A. Henzinger, The theory of hybrid automata, in: Proceedings of Eleventh Annual IEEE Sympo-

sium on Logic in Computer Science, IEEE Press, 1996, pp. 278–292.
[63] T. Kosar, P. E. M. López, P. A. Barrientos, M. Mernik, A preliminary study on various implementation approaches of domain-specific language, Information and Software Technology 50 (2008) 390–405.
[64] G. Hedin, E. Magnusson, The JastAdd system - an aspectoriented compiler construction system, Science of Computer Programming 47 (2003) 37–58.
[65] T. Ekman, G. Hedin, The jastadd system–modular extensible compiler construction, Science of Computer Programming 69 (2007) 14–26.
[66] J. Åkesson, T. Ekman, G. Hedin, Implementation of a modelica compiler using jastadd attribute grammars, Science of Computer Programming 75 (2010) 21–38.
[67] M. Pettersson, Compiling Natural Semantics, volume 1549 of *Lecture Notes in Computer Science*, Springer, 1999.
[68] A. Pop, P. Fritzson, MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language, in: 7th Joint Modular Languages Conference, volume 4228 of *LNCS*, Springer-Verlag, 2006, pp. 211–229.
[69] P. Fritzson, P. Aronsson, H. Lundvall, K. Nyström, A. Pop, L. Saldamli, D. Broman, The OpenModelica Modeling, Simulation, and Software Development Environment, Simulation News Europe 15 (2005) 8–16.
[70] L. Tratt, Domain specific language implementation via compile-time meta-programming, ACM Trans. Program. Lang. Syst. 30 (2008) 31:1–31:40.
[71] J. R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (2006) 190–210.
[72] M. Bravenboer, E. Visser, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, in: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '04, ACM, New York, NY, USA, 2004, pp. 365–383.
[73] D. Batory, B. Lofaso, Y. Smaragdakis, Jts: Tools for implementing domain-specific languages, in: Proceedings of the Fifth International Conference on Software Reuse, IEEE, pp. 143–153.
[74] P. Hudak, Modular domain specific languages and tools, in: Proceedings: Fifth International Conference on Software Reuse, IEEE Press, 1998, pp. 134–142.
[75] A. Shalit, The Dylan reference manual: the definitive guide to the new object-oriented dynamic language, Addison-Wesley, 1996.
[76] C. Chambers, et al., The cecil language – specification and rationale: Version 3.2, 2004. Department of Computer Science and Engineering, University of Washington.
[77] R. B. "de Oliveira", The boo programming language, 2005. Available at: `http://http://boo.codehaus.org/` [Last accessed: April 26, 2012].
[78] E. Meijer, P. Drayton, Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, in: OOPSLA workshop on revival of dynamic languages.
[79] G. Bracha, Pluggable type systems, in: OOPSLA workshop on revival of dynamic languages.
[80] G. Bracha, D. Griswold, Strongtalk: typechecking smalltalk in a production environment, in: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '93, ACM, New York, NY, USA, 1993, pp. 215–230.
[81] P. Wadler, R. B. Findler, Well-Typed Programs Can't Be Blamed, in: European Symposium on Programming (ESOP), volume 5502 of *LNCS*, Springer-Verlag, 2009, pp. 1–15.
[82] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, PLDI '91, ACM, New York, NY, USA, 1991, pp. 278–292.
[83] C. Flanagan, M. Felleisen, Componential set-based analysis, ACM Trans. Program. Lang. Syst. 21 (1999) 370–416.
[84] A. Aiken, E. L. Wimmers, T. K. Lakshman, Soft typing with conditional types, in: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '94, ACM, New York, NY, USA, 1994, pp. 163–173.
[85] F. Henglein, Dynamic typing: Syntax and proof theory, Science of Computer Programming 22 (1994) 197–230.
[86] F. Henglein, J. Rehof, Safe polymorphic type inference for a dynamically typed language: translating scheme to ml, in: Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA '95, ACM, New York, NY, USA, 1995, pp. 192–203.
[87] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin, Dynamic typing in a statically typed language, ACM

Transactions on Programming Languages and Systems 13 (1991) 237–268.
[88] K. E. Gray, R. B. Findler, M. Flatt, Fine-grained interoperability through mirrors and contracts, in: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, ACM, New York, NY, USA, 2005, pp. 231–245.
[89] J. Matthews, R. B. Findler, Operational semantics for multi-language programs, ACM Trans. Program. Lang. Syst. 31 (2009) 12:1–12:44.
[90] S. Tobin-Hochstadt, M. Felleisen, Interlanguage migration: from scripts to programs, in: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 964–974.
[91] S. Tobin-Hochstadt, M. Felleisen, The design and implementation of typed scheme, in: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '08, ACM, New York, NY, USA, 2008, pp. 395–406.
[92] K. Knowles, C. Flanagan, Hybrid type checking, ACM Trans. Program. Lang. Syst. 32 (2010) 6:1–6:34.
[93] X. Ou, G. Tan, Y. Mandelbaum, D. Walker, Dynamic Typing with Dependent Types , in: Exploring new frontiers of theoretical informatics, volume 155 of *IFIP*, Springer-Verlag, 2004, pp. 437–450.
[94] J. Riely, M. Hennessy, Trust and partial typing in open systems of mobile agents, Journal of Automated Reasoning 31 (2003) 335–370.
[95] M. Furr, J.-h. D. An, J. S. Foster, M. Hicks, Static type inference for ruby, in: Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, ACM, New York, NY, USA, 2009, pp. 1859–1866.
[96] C. Anderson, S. Drossopoulou, BabyJ: From object based to class based programming via types, Electronic Notes in Theoretical Computer Science 82 (2003) 53–81.
[97] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, C. Flanagan, Sage: Hybrid checking for flexible specifications, in: Proceedings of the 2006 Scheme and Functional Programming Workshop, pp. 93–104.
[98] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, J. Vitek, Integrating typed and untyped code in a scripting language, in: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '10, ACM, New York, NY, USA, 2010, pp. 377–388.
[99] S. Wolfram, The mathematica book, Cambridge University Press, 4 edition, 1999.
[100] C. Calcagno, W. Taha, L. Huang, X. Leroy, Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection, in: Proceedings of Second International Conference on Generative Programming and Component Engineering (GPCE'03), volume 2830 of *LNCS*, Springer-Verlag, 2003, pp. 57–76.
[101] T. Sheard, Accomplishments and research challenges in meta-programming, in: Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation, volume 2196 of *LNCS*, Springer-Verlag, 2001, pp. 2–44.
[102] G. Neverov, P. Roe, Towards a fully-reflective meta-programming language, in: Proceedings of the Twenty-eighth Australasian conference on Computer Science - Volume 38, ACSC '05, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2005, pp. 151–158.
[103] A. Stump, Directly reflective meta-programming, Higher-Order and Symbolic Computation 22 (2009) 115–144.
[104] A. Löh, R. Hinze, Open data types and open functions, in: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming, PPDP '06, ACM, New York, NY, USA, 2006, pp. 133–144.
[105] T. Millstein, C. Bleckner, C. Chambers, Modular typechecking for hierarchically extensible datatypes and functions, in: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, ICFP '02, ACM, New York, NY, USA, 2002, pp. 110–122.
[106] R. Lämmel, S. P. Jones, Scrap your boilerplate: a practical design pattern for generic programming, in: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation, TLDI '03, ACM, New York, NY, USA, 2003, pp. 26–37.
[107] R. Lämmel, S. P. Jones, Scrap more boilerplate: reflection, zips, and generalised casts, in: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming, ICFP '04, ACM, New York, NY, USA, 2004, pp. 244–255.
[108] R. Lämmel, S. P. Jones, Scrap your boilerplate with class: extensible generic functions, in: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, ICFP '05, ACM Press, New York, USA, 2005, pp. 204–215.
[109] E. Axelsson, A generic abstract syntax model for embedded languages, in: Proceedings of the 17th

ACM SIGPLAN International Conference on Functional Programming (to appear), ICFP '12, ACM Press, New York, USA, 2012.

[110] B. Jay, Pattern Calculus: Computing with Functions and Structures, Springer-Verlag, 2009.

[111] D. Broman, Flow Lambda Calculus for Declarative Physical Connection Semantics, Technical Reports in Computer and Information Science No. 1, LiU Electronic Press, 2007.

[112] C. Nytsch-Geusen et. al., MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics, in: Proceedings of the 4th International Modelica Conference, Hamburg, Germany.

[113] D. Zimmer, Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems, in: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, pp. 47–56.

[114] D. Zimmer, Equation-Based Modeling of Variable-Structure Systems, Ph.D. thesis, Swiss Federal Institute of Technology, Zrich, Switzerland, 2010.

[115] J. Åkesson, K.-E. Årzén, M. Gfvert, T. Bergdahl, H. Tummescheit, Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem, Computers and Chemical Engineering 34 (2010) 1737–1749.

[116] E. Axelsson, K. Claessen, M. Sheeran, Wired: Wire-aware circuit design, in: Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME), volume 3725 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

[117] H. Nilsson, J. Peterson, P. Hudak, Functional Hybrid Modeling, in: Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003, volume 2562 of *LNCS*, Springer-Verlag, New Orleans, Lousiana, USA, 2003, pp. 376–390.

[118] G. Giorgidze, H. Nilsson, Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems, in: Proceedings of the 7th International Modelica Conference, LiU Electronic Press, Como, Italy, 2009, pp. 208–218.

[119] H. Olsson, M. Otter, S. E. Mattsson, H. Elmqvist, Balanced Models in Modelica 3.0 for Increased Model Quality, in: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, pp. 21–33.

[120] P. Aronsson, D. Broman, Extendable Physical Unit Checking with Understandable Error Reporting, in: Proceedings of the 7th International ModelicaConference, Como, Italy, pp. 890–897.

[121] D. Broman, P. Aronsson, P. Fritzson, Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica, in: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, pp. 3–12.

[122] S.-E. Matttsson, H. Elmqvist, Unit Checking and Quantity Conservation, in: Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, pp. 13–20.

[123] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, Taming heterogeneity - the Ptolemy approach, Proceedings of the IEEE 91 (2003) 127–144.