Gradually Typed Symbolic Expressions

David Broman
KTH Royal Institute of Technology
Sweden
dbro@kth.se

Jeremy G. Siek
Indiana University
USA
jsiek@indiana.edu

Abstract

Embedding a domain-specific language (DSL) in a general purpose host language is an efficient way to develop a new DSL. Various kinds of languages and paradigms can be used as host languages, including object-oriented, functional, statically typed, and dynamically typed variants, all having their pros and cons. For deep embedding, statically typed languages enable early checking and potentially good DSL error messages, instead of reporting runtime errors. Dynamically typed languages, on the other hand, enable flexible transformations, thus avoiding extensive boilerplate code. In this paper, we introduce the concept of gradually typed symbolic expressions that mix static and dynamic typing for symbolic data. The key idea is to combine the strengths of dynamic and static typing in the context of deep embedding of DSLs. We define a gradually typed calculus $\lambda^{<\star>}$, formalize its type system and dynamic semantics, and prove type safety. We introduce a host language called Modelyze that is based on $\lambda^{<\star>}$, and evaluate the approach by embedding a series of equation-based domain-specific modeling languages, all within the domain of physical modeling and simulation.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; *Functional languages*; • **Computing methodologies** → *Modeling and simulation*;

Keywords Symbolic expressions, DSL, Type systems

ACM Reference Format:
David Broman and Jeremy G. Siek. 2018. Gradually Typed Symbolic Expressions. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3162068

PEPM'18, January 8–9, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5587-2/18/01…$15.00
https://doi.org/10.1145/3162068

1 Introduction

Implementing an efficient and user friendly *domain-specific language (DSL)* is hard because it requires both domain knowledge and expert knowledge in compilers and programming language design [46]. An attractive alternative to building languages from scratch is to grow the language [66] by pushing syntactic and semantic extensions into libraries [73]. One such approach, pioneered by Hudak [30], is to create *embedded* DSLs. In this approach, the underlying host language provides enough syntactic and semantic flexibility to make libraries appear to be language extensions. Embedded DSLs have been successfully deployed in many domains [3, 6, 19, 24, 68, 79].

Although embedded DSLs mitigate the development effort for the language designer, it is challenging to get the same quality of experience for the DSL user, compared to a DSL created from scratch. In particular, we would like to emphasize two main challenges when designing a host language for embedded DSLs. First, the host language's syntax should ideally be *seamlessly integrated* with the DSL, to make it feel as one consistent language. Even if the basic syntax of the DSL is chosen to suit the end user, some constructs may need to be staged into an abstract syntax tree, and further manipulated and interpreted. Such embedding is often referred to as *deep*. Other constructs may be possible to translate directly into the host language, often called *shallow* embedding. The separation between stages needs to be seamless and compiler error messages should be domain-specific and not leak details from the underlying host language. Second, the host language should be *expressive* enough to enable the embedding of arbitrary DSLs, and at the same time *easy to use* for domain engineers with limited compiler and language background. Language concepts, such as monads [76], type classes [77], and GADTs [16, 52, 59, 81], are powerful constructs for implementing embedded DSLs, but they also have a steep learning curve. The challenge is to provide language mechanisms that minimize the training needed to pattern match, transform, and analyze DSL constructs.

Both statically and dynamically typed general-purpose languages are common to use as host languages. *Statically typed* approaches—such as Lightweight Modular Staging (LMS) [57], Scala-Virtualized [56], Template Haskell [58], and Finally Tagless [15]—all enable early checking using static types. Also, by using advanced type systems, such as ML modules, type classes, and GADTs, a compiler can give static type safety guarantees for certain DSL transformations.

However, the DSL designer needs to be very knowledgeable of advanced types, and there are still transformations that cannot be performed conveniently in a typed setting. Moreover, even state-of-the-art approaches still require a significant amount of boilerplate code [54] when designing DSLs. By contrast, *dynamically typed languages* commonly used for embedding—such as Racket [22], LISP [65], Julia, and Python—do not have expressiveness limitations due static type system, but on the other hand they do not provide any static guarantees concerning correctness of transformations. As always in languages with dynamic typing, type errors are only discovered at runtime, which can make it challenging for the end user to understand the DSL error.

In this paper, we explore the design space of a host language that combines static and dynamic typing. In particular, we motivate the use of this mixture to provide the end user with relevant error messages (static typing), while at the same time enabling flexible and simple transformations (dynamic typing). The key innovation in this paper is the concept of *gradually typed symbolic expressions.* The theory is based on gradual typing [60, 61] and it tracks precise types for symbolic expressions, inspired by MetaML [70]. We present the following contributions:

i) We introduce gradually typed symbolic expressions within the context of the research host language Modelyze[1]. Modelyze has been available since 2012 [12], but this is the first formal peer-reviewed publication describing its core. In particular, we demonstrate how our approach uses early static checking and avoids boilerplate code (Section 2).

ii) We define the dynamic semantics and type system of a gradually typed calculus $\lambda^{<\star>}$ (pronounced "gradsym"), which is the core of Modelyze. To provide a seamless integration between the host language and a DSL, we introduce a symbolic lifting analysis that is inspired by binding-time analysis [27]. We prove type safety (Section 3).

iii) We evaluate our approach by defining a series of equation-based domain-specific modeling languages embedded in Modelyze (Section 4).

2 Motivation: Modeling and Simulation

This section describes and motivates the concept of gradually typed symbolic expressions, within the DSL domain of physical modeling and simulation.

2.1 Equation-Based Modeling and Simulation

Cyber-physical systems (CPS) [42], such as automobiles and power plants, are expensive to develop because of the complexity and need for safety and correctness. To master this complexity, equation-based modeling languages (for instance Modelica® [48] and VHDL-AMS [32]) can be used for simulation, before creating expensive physical prototypes. In these languages, the primary constructs for describing the

[1] http://www.modelyze.org

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

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

Figure 1. A pendulum model defined in Modelyze.

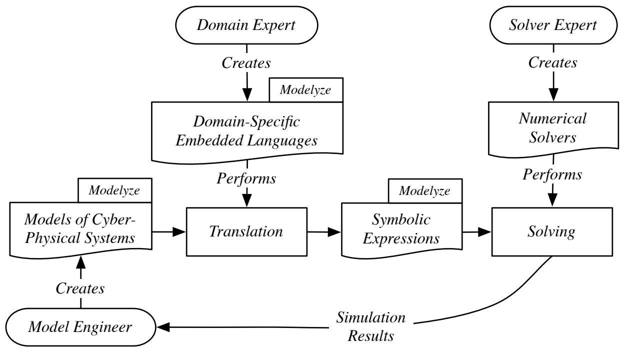

Figure 2. The roles, processes, and artifacts associated with the Modelyze approach to modeling cyber-physical systems.

continuous-time behavior are differential equations. For instance, Figure 1 lists the model of a pendulum, expressed as a system of differential-algebraic equations (DAEs) [38] in cartesian coordinates. Variables `x` and `y` are the coordinates for the ball of the pendulum, `l` the length of the string, and `T` the tension in the string. 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 [21, 31, 45, 51] and numerical approximation [28] techniques can be used to automatically solve such equation systems efficiently. Another key characteristic of equation-based modeling languages is to support hierarchical structures of systems, and to facilitate large scale reuse [20].

In the case study in this paper, we apply the embedded DSL approach to the domain of equation-based modeling. In particular, we describe a host language *Modelyze* that supports the development of modeling languages as embedded DSLs. A key motivation for Modelyze was to enable the development of extensible DSLs, where new language features can be gradually added. Figure 2 shows the human 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.

Returning to Figure 1, the Pendulum model is defined using a function abstraction. Line 2 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 2 is evaluated. As usual, we use the term *variable* for functional variables that can only be bound to a value once. Lines 3-4 specify the initial conditions for state variables x and y and lines 6-8 state the differential equations. The order of the equations is not important.

2.2 Seamless Integration - Reducing Annotations

In the Pendulum example, it is not obvious which parts of the syntax are from the host language and which are from the embedded DSL. This is intentional and is what we call *seamless integration* between the host language and the embedded DSL. 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 [65], a quasi-quoted expression `(+ 1 ,a) means that the expression should be treated as data 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 [70], angle brackets (< >) are quotation and tilde (~) is anti-quoting. However, one problem with quasi-quoting is that it adds an extra level of *annotation burden* on the model engineer to carefully add quotes at selected places in a program. For instance, if code line 8 of the Pendulum example uses MetaML's quasi-quote notation, the resulting code is

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

The model engineer must carefully consider the different sub-expressions. To relieve the model engineer from this annotation burden, the quotation of symbolic expressions is performed implicitly by the Modelyze compiler. We call this process *symbolic lifting analysis* (SLA). In contrast to *binding time analysis* (BTA) [27] in partial evaluation [36], 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 should be lifted. This idea has similarities to the Rep type of LMS [57]. See the related work section for details.

Example 2.1 (Symbolic Lifting)**.** Consider again the example in Figure 1, where three typed symbols are created on line 2. Each 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

$$\texttt{def}\ x{:}T;e \tag{1}$$

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 reusable models.

Let us zoom in on expression x/l 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 summarize this subsection, we gave some intuition regarding type checking and the symbolic lifting. The full details of the type system, including symbolic lifting and a proof of type safety, are presented in Section 3.

2.3 Matching Open Gradually Typed S-Expressions

In this section, we show how a domain expert can traverse typed symbolic expressions in a deeply embedded DSL.

Example 2.2 (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. An important function in the latter library, shown in Figure 3, collects all the unknowns of an equation system. This function recursively traverses a symbolic expression representing an equation system and returns all the typed

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

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

symbols of type `Real`, representing unknowns. The function takes two parameters `e` (the symbolic expression) and `acc` (an accumulator for a set of symbols) of types `<Dyn>` and `UkSet`, respectively. The first parameter uses the *dynamic* type `Dyn`, meaning that `e` 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 `<Dyn>` enables the expression of generic traversals over symbolic expressions, thus avoiding any boilerplate code. This is an example where gradual typing is used to improve expressiveness by using dynamic checking for fragments of the program. As always with dynamic typing, there are no static type guarantees for the traversal function.

Example 2.3 (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:

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

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. For instance, in the case study (Section 4) we extend an existing DAE DSL with modes and transitions, where a transition between modes is defined as a new symbol. 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`[2]. We also define a postfix symbolic function `'` for representing derivatives.

[2]Note that `der` with type `<Real -> Real>` is a symbol itself and applying a value to it results in a lifted symbolic expression that can later be deconstructed. By contrast, if a function has type `<Real> -> <Real>`, it is an ordinary function that takes a symbolic expression as input and returns another symbolic expression.

2.4 Static Error Checking at the DSL Level

When a model engineer makes mistake in constructing a model, it is important that the error messages directly reflect the abstraction level of the DSL for that model.

Assume we replace line 4 of the pendulum example with the following line:

```
init y; //Error: Missing initial value
```

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 cause the numerical solver to fail when trying to initialize the equation system. In such a case, the model engineer would 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 DSL author can provide error messages to the user 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:

```
pendulum2.moz 4:10-4:10 error: Missing
argument of type 'Real'.
```

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 [11, 13, 50].

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 some static error reporting at the DSL level. However, as always, static type checking can only detect some and not all kinds of program errors.

3 Formalization of $\lambda^{<\star>}$

This section presents the dynamic semantics and the type system for the gradually typed symbolic calculus $\lambda^{<\star>}$. As is standard in the literature for gradual types, we use $\star$ to denote the corresponding dynamic type `Dyn` in Modelyze. Consequently, $<\star>$ denotes the dynamic symbolic type. To prove type safety, we present two additional intermediate languages: $\lambda_L^{<\star>}$ and $\lambda_{LC}^{<\star>}$. We define a translation from $\lambda^{<\star>}$ to $\lambda_L^{<\star>}$ that lifts selected expressions into symbolic expressions. The reason for symbolic lifting is to create data structures that can later be inspected and analyzed. Both $\lambda^{<\star>}$ and $\lambda_L^{<\star>}$ are gradually typed languages. The dynamic aspect is made explicit through a cast insertion translation from $\lambda_L^{<\star>}$ to $\lambda_{LC}^{<\star>}$. We present an operational semantics for $\lambda_{LC}^{<\star>}$

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

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

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

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

Figure 4. Abstract syntax of $\lambda^{<\star>}$ and $\lambda_L^{<\star>}$.

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

$$\tau \sim \star \qquad \star \sim \tau \qquad B \sim B \qquad D \sim D$$

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

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

$$\tau \sqcap \star = \tau \qquad \star \sqcap \tau = \tau$$
$$B \sqcap B = B \qquad D \sqcap D = D$$
$$(\tau_1 \to \tau_2) \sqcap (\tau_3 \to \tau_4) = (\tau_1 \sqcap \tau_3) \to (\tau_2 \sqcap \tau_4)$$
$$<\tau_1> \sqcap <\tau_2> = <\tau_1 \sqcap \tau_2>$$

Figure 5. Type consistency relation and meet operation.

and prove that the translations between the intermediate languages are type preserving. We prove the usual progress and preservation lemmas for $\lambda_{LC}^{<\star>}$ and thereby obtain type safety for $\lambda^{<\star>}$. For complete proofs, see the tech report [12].

3.1 Syntax

The abstract syntax for $\lambda^{<\star>}$ is defined in Figure 4. The first five expressions are standard. There are two new kinds of expressions in $\lambda^{<\star>}$. The "new" expression $\nu(\tau)$ creates a fresh symbol with type τ. The expression $\texttt{case}(e, p, e_t, e_f)$ eliminates symbolic data. The value of e is matched against the pattern p. Patterns are non-recursive in $\lambda^{<\star>}$. Nested patterns in a source language should be compiled into `case` expressions in $\lambda^{<\star>}$. 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, $x \mathbin{@} x$ for matching symbolic applications, and $\texttt{sval}\ x{:}\tau$ for values that have been lifted to symbolic values. In the `sval` pattern form, the x is a pattern variable and τ a type tag.

There are three standard types and two new types for this language. The metavariable B ranges over all base types $\mathbb{G}$ (e.g., booleans and integers), types of the form $\tau{\to}\tau$ are function types, and $\star$ is the dynamic type. To categorize symbolic data of type τ, we introduce the type $<\tau>$. Also, D ranges over primitive symbolic data types. There is a finite set of such types in a program. Figure 4 also introduces $\lambda_L^{<\star>}$ that adds two additional expressions: symbolic applications @ and lifted symbolic values `sval`.

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 $\sim$ [60, 61]. The definition of type consistency is given in Figure 5. The consistency relation is closely related to the meet operator $\sqcap$. The meet operator computes the greatest lower bound (if it exists) with respect to the naive subtyping relation [78] (or the least upper bound of the precision relation $\sqsubseteq$ [60]).

Proposition 3.1. *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. See the tech report [12]. □

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. We define the subset relation between typing environments as follows.

Definition 3.2. $\Gamma \subseteq \Gamma' \equiv \forall x\tau.\ \Gamma(x) = \tau$ *implies* $\Gamma'(x) = \tau$.

The type system for $\lambda^{<\star>}$ is the *symbolic lifting relation*

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

where e is an expression in $\lambda^{<\star>}$, e' an expression in $\lambda_L^{<\star>}$, τ is the type of the resulting value, and Γ is a typing environment. This relation is inductively defined by the inference rules in Figure 6, which we discuss shortly.

Definition 3.3 (Well-typed expression in $\lambda^{<\star>}$)**.** An expression e of $\lambda^{<\star>}$ is well typed (typable) in typing environment Γ at type τ if there exists e' such that $\Gamma \vdash_L e \rightsquigarrow e' : \tau$.

Language $\lambda^{<\star>}$ 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.

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 $\Delta : \mathbb{C} \to$ *Types* that when applied to a constant returns the constant's type. We assume that the Δ-function cannot return a symbolic type and therefore give the following assumption:

Assumption 1 (Δ-types)**.**
If $\Delta(c) = \tau$ then $\tau \in \mathbb{G}$ or there exists τ_1 and τ_2 such that $\tau = \tau_1{\to}\tau_2$.

$$\Gamma \vdash_L \nu(\tau_1) \rightsquigarrow \nu(\tau_1) : <\tau_1> \quad \text{(L-NEW)} \qquad\qquad \Gamma \vdash_L \texttt{error} \rightsquigarrow \texttt{error} : \tau \quad \text{(L-ERROR)}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : <\tau_{11}> \to \tau_{12} \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \\ <\tau_{11}> \not\sim \tau_2 \qquad \tau_{11} \sim \tau_2\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1'\ (\texttt{sval}\ e_2' : \tau_2) : \tau_{12}} \text{ (L-APP3)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \tau_{11} \to \tau_{12} \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : <\tau_2> \\ \tau_{11} \not\sim <\tau_2> \qquad \tau_{11} \sim \tau_2 \\ e_3 = \texttt{sval}\ e_1' : \tau_{11} \to \tau_{12}\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_3 @\ e_2' : <\tau_{12}>} \text{ (L-APP4)} \qquad \frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : <\tau_{11} \to \tau_{12}> \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2'' \\ <\tau_{11}> \sim \lceil \tau_2 \rceil\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1' @\ e_2'' : <\tau_{12}>} \text{ (L-APP5)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \rightsquigarrow e_1' : <\star> \\ \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \\ \lceil e_2' : \tau_2 \rceil = e_2''\end{array}}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1' @\ e_2'' : <\star>} \text{ (L-APP6)}$$

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

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

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

Figure 6. Type system and symbolic lifting for $\lambda^{<\star>}$. For brevity, standard rules for vars, lambdas, and consts are omitted.

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

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

$$\lceil \tau \rceil = \begin{cases} \tau & \text{if } \tau \sim <\star> \\ <\tau> & \text{otherwise} \end{cases}$$

Proposition 3.4.

1. *If* $\Gamma \vdash_L e \rightsquigarrow e' : \tau$, *then* $\Gamma \vdash_L \lceil e : \tau \rceil \rightsquigarrow \lceil e' : \tau \rceil : \lceil \tau \rceil$.
2. $\lceil \tau \rceil \sim <\star>$

Because $\lambda^{<\star>}$ 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 $\star$, in which case any argument type is allowed, as specified in rule (L-APP2). Next, to implement symbolic lifting, if 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 application remains a normal function application.

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

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.

$$\begin{array}{r}\vdash (\lambda x{:}\texttt{Int}.x)\ \nu(\texttt{Int}) \rightsquigarrow \\ (\texttt{sval}\ (\lambda x{:}\texttt{Int}.x) : \texttt{Int} \to \texttt{Int}) @\ \nu(\texttt{Int})\end{array}$$

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.

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

The next example shows gradual typing for S-expressions.

$$\vdash \nu(\star)\ 5 \rightsquigarrow \nu(\star) @\ (\texttt{sval}\ 5{:}\texttt{Int})$$

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 $<\star> \sim \tau_1$. 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 $<\star>$, 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 [60]. The explicit casts make it easier to reason about when errors should occur and better reflects the runtime representations that could potentilly be used in a compiled implementation.

The abstract syntax for $\lambda^{<\star>}_{LC}$ 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 τ_1 to target type τ_2. Also we add an expression for the runtime representation of a symbol $(s : \tau)$. Cast insertion is defined by a *cast insertion relation*

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

where e is an expression in $\lambda^{<\star>}_{L}$, e' an expression in $\lambda^{<\star>}_{LC}$, τ the resulting type, and Γ 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 [60, 63]. One interesting thing to note is that, in rules (C-SAPP1) and (C-SAPP2), the function and argument are cast to $<\star>$ because that is the type expected when a `case` expression decomposes a symbolic application. The notion of well-typed expression for $\lambda^{<\star>}_{L}$ is defined in terms of the cast insertion relation.

Definition 3.5 (Well-typed expression in $\lambda^{<\star>}_{L}$)**.** An expression e of $\lambda^{<\star>}_{L}$ is well typed (typable) in typing environment Γ at type τ if there exists e' such that $\Gamma \vdash_C e \rightsquigarrow e' : \tau$.

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

Proposition 3.6 (Symbolic Lifting Preserves Types)**.** *If* $\Gamma \vdash_L e \rightsquigarrow e' : \tau$ *then* e' *is well typed in* Γ *at type* τ.

Proof. By induction on a derivation of $\Gamma \vdash_L e \rightsquigarrow e' : \tau$. ◻

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

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

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

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

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

where e is an expression in $\lambda^{<\star>}_{LC}$, τ its type, and Γ 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 3.7 (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 $\Gamma \vdash_C e \rightsquigarrow e' : \tau$. ◻

3.5 Dynamic Semantics

We define the dynamic semantics of $\lambda^{<\star>}$ in Figure 10 by defining a partial function *eval* from well-typed $\lambda^{<\star>}$ expressions to observations. A valid implementation of $\lambda^{<\star>}$ 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 $\lambda^{<\star>}_{LC}$ in *small-step* style [53]. 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 metavariable $S \subseteq \mathbb{S}$ 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. 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 $\lambda^{<\star>}_{LC}$ has casts, there are several value forms for casted values. Lastly, there are three values forms for the three kinds of symbolic data.

The rule `(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 τ_1 from the ν-expression. Rules `(E-CASE-T)` and `(E-CASE-F)` *deconstruct* symbolic expressions. The value v_1, the deconstructor pattern p, and the expression e_2 are given to the following *match* predicates.

$$match(s : \tau_1, \texttt{sym} : \tau_1, e_1, e_1)$$

$$match(v_1 @ v_2, x_1 @ x_2, e_1, (\lambda x_1 : <\star>. \lambda x_2 : <\star>. e_1)\, v_1\, v_2)$$

$$match(\texttt{sval}\ v_1 : \tau_1, \texttt{sval}\ x : \tau_1, e_1, (\lambda x : \tau_1. e_1)\, v_1)$$

In addition to the rules for function application, there are also five rules for handling casts, which are standard for cast calculi [64] but perhaps deserve some review. Because we have casted values at function type, there must be a reduction

$$\frac{}{\Gamma \vdash_C \nu(\tau_1) \rightsquigarrow \nu(\tau_1) : <\tau_1>}\ \text{(C-NEWSYM)} \qquad \frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1}{\Gamma \vdash_C \texttt{sval}\ e_1 : \tau_1 \rightsquigarrow (\texttt{sval}\ e_1' : \tau_1) : <\tau_1>}\ \text{(C-LIFT)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_{11} \to \tau_{12} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow e_1'\ (\langle \tau_{11} \Leftarrow \tau_2 \rangle e_2') : \tau_{12}}\ \text{(C-APP1)} \qquad \frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \star \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow (\langle \star \to \star \Leftarrow \star \rangle e_1')\ \langle \star \Leftarrow \tau_2 \rangle e_2' : \star}\ \text{(C-APP2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : <\star> \\ \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \qquad <\star> \sim \tau_2\end{array}}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow e_1' @ \langle <\star> \Leftarrow \tau_2 \rangle e_2' : <\star>}\ \text{(C-SAPP1)} \qquad \frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : <\tau_{11} \to \tau_{12}> \qquad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ e_1'' = (\langle <\star> \Leftarrow <\tau_{11} \to \tau_{12}> \rangle e_1') \\ e_2'' = \langle <\star> \Leftarrow \tau_2 \rangle e_2' \qquad <\tau_{11}> \sim \tau_2\end{array}}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow \langle <\tau_{12}> \Leftarrow <\star> \rangle (e_1'' @ e_2'') : <\tau_{12}>}\ \text{(C-SAPP2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \qquad <\star> \sim \tau_1 \\ \tau_2 \sim \tau_3 \qquad \tau_5 = \tau_2 \sqcap \tau_3 \qquad e_1'' = \langle <\star> \Leftarrow \tau_1 \rangle e_1' \\ e_2'' = \langle \tau_5 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_5 \Leftarrow \tau_3 \rangle e_3'\end{array}}{\begin{array}{l}\Gamma\ \vdash_C\ \texttt{case}(e_1, \texttt{sym}{:}\tau_4, e_2, e_3) \rightsquigarrow \\ \qquad \texttt{case}(e_1'', \texttt{sym}{:}\tau_4, e_2'', e_3'') : \tau_5\end{array}}\ \text{(C-CSYM)} \qquad \frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \\ \Gamma, x_1 : <\star>, x_2 : <\star>\ \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \qquad <\star> \sim \tau_1 \qquad \tau_2 \sim \tau_3 \\ \tau_4 = \tau_2 \sqcap \tau_3 \qquad e_1'' = \langle <\star> \Leftarrow \tau_1 \rangle e_1' \\ e_2'' = \langle \tau_4 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_4 \Leftarrow \tau_3 \rangle e_3'\end{array}}{\begin{array}{l}\Gamma\ \vdash_C\ \texttt{case}(e_1, x_1 @ x_2, e_2, e_3) \rightsquigarrow \\ \qquad \texttt{case}(e_1'', x_1 @ x_2, e_2'', e_3'') : \tau_4\end{array}}\ \text{(C-CAPP)}$$

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

Figure 8. The cast insertion relation. Rules for variables, lambda, error, and const are omitted.

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

$$\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (\texttt{sval}\ e_1 : \tau_1) : <\tau_1>}\ \text{(T-LIFT)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : <\star> \\ \Gamma \vdash e_2 : <\star>\end{array}}{\Gamma \vdash e_1 @ e_2 : <\star>}\ \text{(T-SAPP)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : <\tau_1> \\ \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2\end{array}}{\Gamma \vdash \texttt{case}(e_1, \texttt{sym}{:}\tau_4, e_2, e_3) : \tau_2}\ \text{(T-CASE-SYM)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : <\tau_1> \\ \Gamma, x_1 : <\star>, x_2 : <\star>\ \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2\end{array}}{\Gamma \vdash \texttt{case}(e_1, x_1 @ x_2, e_2, e_3) : \tau_2}\ \text{(T-CASE-APP)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : <\tau_1> \\ \Gamma, x : \tau_4\ \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2\end{array}}{\Gamma \vdash \texttt{case}(e_1, \texttt{sval}\ x{:}\tau_4, e_2, e_3) : \tau_2}\ \text{(T-CASE-LIFT)}$$

Figure 9. Type system for $\lambda_{LC}^{<\star>}$. For brevity, we omit standard rules for var, const, lambda, and application.

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 [17]. The two approaches are observationally equivalent.) The reduction rules (`E-CAST2`) and (`E-CAST3`) discard identity casts on base types and on type $\star$. The rules (`E-CAST4`) and (`E-CAST5`) handle the important case of an injection to type $\star$ meeting a projection from type $\star$. 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 [64] provides earlier and more thorough error detection [62].

There is 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 (`E-CAST-C`).

We succinctly express the very many congruence rules with the rule schema (`E-CONG`), inspired by unpublished lecture notes by Andrew Myers. 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. We omit the standard definitions for reflexive transitive closure.

3.6 Type Safety

We prove type safety with the usual progress and preservation lemmas. We omit the basic lemmas for inversion, canonical forms, substitution, and environment weakening.

Lemma 3.8 (Progress). *If* $\vdash e : \tau$ *then* $e \in$ *Values, or for all* S *there exists* S' *and* e' *such that* $e \mid S \longrightarrow e' \mid S'$, *or* $e = \texttt{error}$.

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

$\boxed{e \longrightarrow e}$

$$
\begin{array}{lr}
(\lambda x{:}\tau_1.e_1)\ v_1 \mid S \longrightarrow [x \mapsto v_1]e_1 \mid S & \text{(E-BETA)} \\
c_1\ v_1 \mid S \longrightarrow \delta(c_1, v_1) \mid S & \text{(E-DELTA)} \\
((\langle\tau_1\to\tau_2 \Leftarrow \tau_3\to\tau_4\rangle v_1)\ v_2 \mid S \longrightarrow & \text{(E-CAST1)} \\
\qquad \langle\tau_2 \Leftarrow \tau_4\rangle(v_1\ \langle\tau_3 \Leftarrow \tau_1\rangle v_2) \mid S & \\
\langle\gamma \Leftarrow \gamma\rangle v_1 \mid S \longrightarrow v_1 \mid S & \text{(E-CAST2)} \\
\langle\star \Leftarrow \star\rangle v_1 \mid S \longrightarrow v_1 \mid S & \text{(E-CAST3)} \\
\langle T_2 \Leftarrow \star\rangle\langle\star \Leftarrow T_1\rangle v \mid S \longrightarrow \langle T_2 \Leftarrow T_1\rangle v \quad \text{if } T_1 \sim T_2 & \text{(E-CAST4)} \\
\langle T_2 \Leftarrow \star\rangle\langle\star \Leftarrow T_1\rangle v \mid S \longrightarrow \mathtt{error} \quad \text{if } T_1 \nsim T_2 & \text{(E-CAST5)} \\
\nu(\tau_1) \mid S \longrightarrow s{:}\tau_1 \mid S \cup \{s\} \quad \text{if } s \notin S & \text{(E-NEWSYM)} \\
\mathtt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_2' \mid S & \text{(E-CASE-T)} \\
\qquad \text{if } \mathit{match}(v_1, p, e_2, e_2') & \\
\mathtt{case}(v_1, p, e_2, e_3) \mid S \longrightarrow e_3 \mid S & \text{(E-CASE-F)} \\
\qquad \text{if } \neg\mathit{match}(v_1, p, e_2, e_2') & \\
\mathtt{case}(\langle <\tau_2> \Leftarrow <\tau_1>\rangle v_1, p, e_2, e_3) \mid S & \text{(E-CASE-C)} \\
\qquad \longrightarrow \mathtt{case}(v_1, p, e_2, e_3) \mid S & \\
\dfrac{e \mid S \longrightarrow e' \mid S'}{F[e] \mid S \longrightarrow F[e'] \mid S'} & \text{(E-CONG)} \\
F[\mathtt{error}\ ] \longrightarrow \mathtt{error} & \text{(E-ERROR)}
\end{array}
$$

Observations:

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

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

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

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

Proof. By induction on a derivation of $\vdash e:\tau$. □

We require that the δ function agrees with the Δ function with respect to the types of the values it produces.

Assumption 2 (δ-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 the Match Preservation Lemma.

Lemma 3.9 (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. By cases on pattern p, using the inversion lemma. □

Lemma 3.10 (Preservation)**.** *If $\Gamma \vdash e:\tau$ and $e \mid S \longrightarrow e' \mid S'$ then $\Gamma \vdash e':\tau$.*

Proof. By induction on the reduction $e \mid S \longrightarrow e' \mid S'$. □

Theorem 3.11 (Type Safety of $\lambda^{<\star>}$)**.** *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 $e_4 = \mathtt{error}$, or there exists e_5 and S_5 such that $e_4 \mid S_4 \longrightarrow e_5 \mid S_5$)).*

Proof. By applying soundness of symbolic lifting, soundness of cast insertion, progress, and preservation. □

4 Case Study: Equation-Based DSLs

In this section, we evaluate our approach in the context of equation-based modeling languages. We develop three DSLs that are embedded into our host language Modelyze. The Modelyze interpreter (http://www.modelyze.org) is a non-trivial language implementation that extends the core language presented in Section 3 with new syntactic constructs and additional language extensions, which are essential to make the language useful in practice. The implementation includes desugaring, pattern compilation, type checking, and interpretation. The current implementation does not support cast insertion, which was used in the previous section in the type safety proof. It is implemented in OCaml [34] v4.05.0, together with the SUNDIALS [28] solver suite.

4.1 Overview of DSLs

Figure 11 gives an overview of the three DSLs. For brevity, we only show the most essential parts of the process.

The M-DAE DSL is show to the left of the figure. At the top, we show how a plain DAE model is the input. This is the same model as discussed earlier in Section 2. The simulation process consists of two main phases, i) the `daeInit()` phase, and ii) the `simLoop()` phase. The init phase performs symbolic manipulations and transformations of the equation system and prepares it for numerical approximations. The two main functions are index reduction using Pantelides' algorithm [51] and evaluation of the equation system by generating a residual function. The former part includes bipartite graph algorithms, and the latter part uses a form of online partial evaluation to improve the simulation performance. The second phase iteratively invokes a numerical solver and approximates the simulation result before plotting.

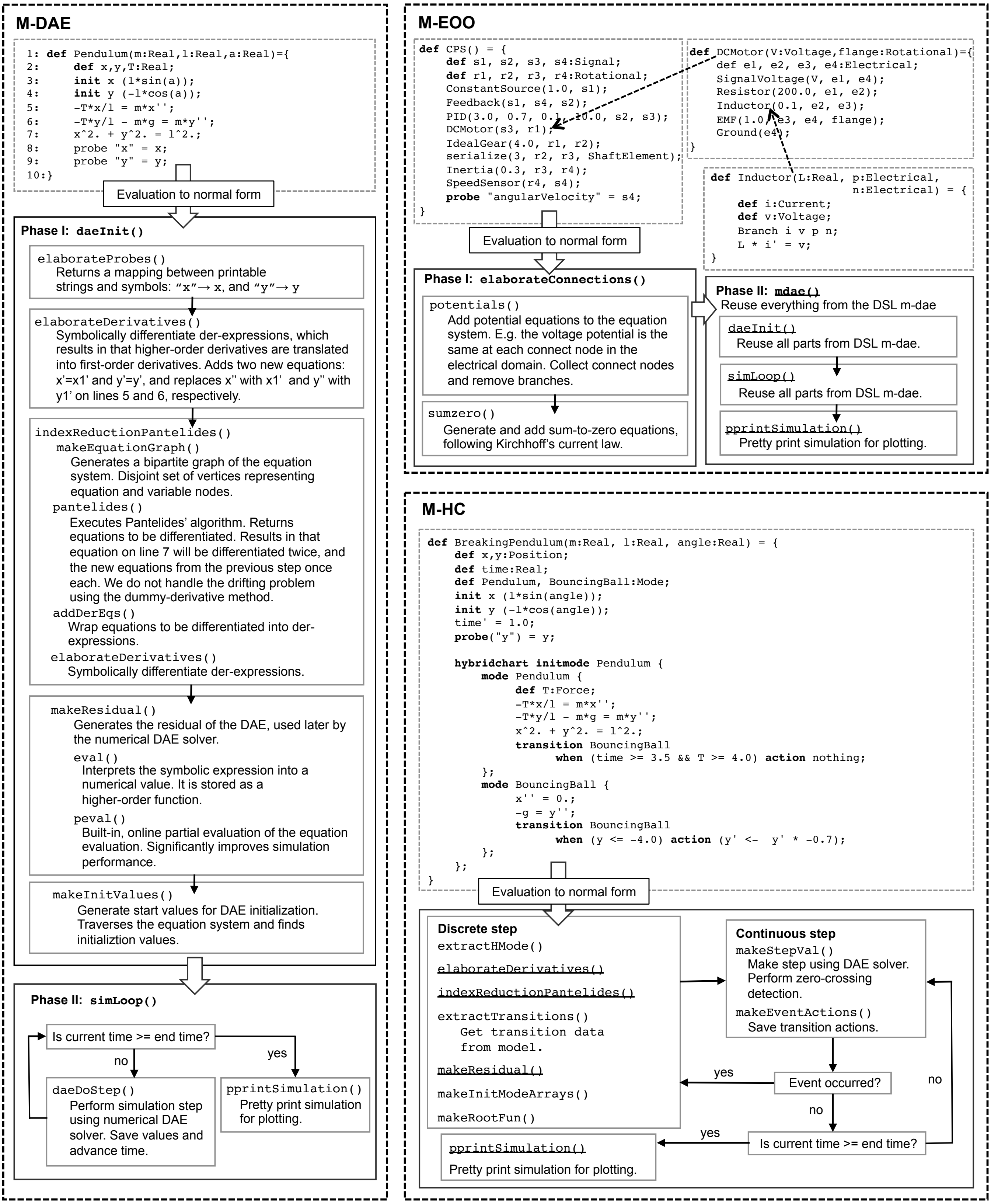

Figure 11. General overview of the translation processes for the three experimental DSLs.

The second DSL, called M-EOO, extends the syntax and semantics from M-DAE for handling basic DAEs. The M-EOO DSL adds equation-based object-oriented (EOO) modeling capabilities, making it possible to hierarchically model complex physical system. The example shows a complete mechatronic powertrain system, combining a direct-current motor, mechanical components, and a PID feedback controller. Note how the DCMotor and the Inductor models are hierarchically defined using functions (dashed arrows). The hierarchy is collapsed into equations in two steps. The first step comes for free by normal evaluation of the model. It generates a deeply embedded data structure, that is first transformed in phase I, and simulated in phase II. Phase I, elaborateConnections, follows the connection semantics defined by Broman and Nilsson [10]. This is an example of *translational* DSL reuse. The DSL is defined by translating the M-EOO model (Phase I) into a M-DAE model.

The third DSL M-HC extends M-DAE by adding state machines where each state (called mode) consists of DAEs. Language M-HC introduces structurally dynamic systems, where the structure of equations changes during run time. The BreakingPendulum model has two modes, where the string of a pendulum with an attached ball breaks, transitions into another mode, where the ball starts to bounce. Note that all syntax extensions are added using symbolic expressions. Keywords such as hybridchart or transition are symbols defined in the DSL. This DSL exemplifies *functional* reuse. It is not possible to directly translate the DSL into M-DAE, but functions from M-DAE can be reused. The reused functions are underlined in the figure.

4.2 Discussion

We will now discuss the strengths and weaknesses of using gradually typed symbols for embedded DSLs.

The symbol lifting approach requires the modeler to use types when defining their models. Without type definitions, the type checker cannot separate the different stages of the program. The benefit of using this approach, compared to force manual quasi-quote notation, should be obvious. However, a more subtle implication is the translation from hierarchical models into equation systems. The transformation can be seen as a staged computation that is not used for performance improvements, but as part of the translational semantics of the DSL.

Another implication of using static types as part of the model definitions is improved error reporting. Obviously, it is better to get a type error that pinpoints the error to a specific source code line, than getting a numerical error during simulation. However, all errors cannot be detected using types, and type errors can also be confusing to model engineers, especially if the host language's internal type system is exposed. Dynamic typing has, on the other hand, both pros and cons, depending on the view point. We would like to point out some observations that we have made during the development of these DSLs.

Dynamic typing enables generic traversals, with minimal boilerplate code. Recall the function uk for getting unknowns in Section 2.3. Dynamic typing is also used for evaluating residual expressions when numerically solving DAEs.

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

Note how parameter e has the dynamic symbolic type <Dyn>, and how curried function applications are matched using pattern f e. Because type checking is done at the DSL level, runtime errors will not occur during evaluation, presupposed the transformations did not introduce any errors.

Dynamic typing directly enables a translational DSL reuse approach in M-EOO. For instance, M-EOO programs include a symbolic constructor Branch (see the Inductor model in Figure 11), which does not exisit in M-DAE. This branch construct is used for expressing connections in, for instance, models of electrical circuits. During elaboration (translation into equations), the following function returns a new equation system without the branches, and collects the branch symbols in a set of branches, BSet.

```
potentials(m:Equations)->(Equations,BSet)
```

Note that if the data type Equations in M-DAE is closed, it is not possible to extend it with the new constructor Branch in DSL M-EOO, without creating a new data type. In this case, by keeping the data type Equations open, we allow static type checking at the model level (introduction of a Branch in a model), and at the same time allow pattern matching when we traverse the equation system. The combination of dynamic typing and open types remove expression limitations, with the cost of loosing static type checking of the translations. Is this a price worth to pay? It is a subjective question, and we do not believe there is a scientific answer. From our experience of developing these quite comprehensive DSLs, we have made extensively use of types in all translation steps, when possible. The dynamic types are only inserted in a few places, when needed. The main problems and debugging efforts have not mainly been due to type problems, but rather because of numerical aspects and equation solving problems, which neither dynamic nor static type checking solves.

We have not found that dynamic typing helps in any direct way for the end user or the model engineer, especially concerning error reporting. However, we have found that dynamic typing gives a reasonable way to enable expressive transformations for the domain expert, and that static typing is vital for good error reporting at the DSL level.

5 Related Work

Domain-Specific Languages. There are different ways to develop DSLs [46], such as tools for *compiler construction* [18], and *preprocessing*. Examples of the latter are LISP's macro system [67], template metaprogramming in C++ [75], homogeneous metaprogramming [74], Template Haskell [58], Stratego/XP [7], and METABORG [8]. In contrast to Template Haskell—where code is transformed at compile time and type checked before execution—Modelyze transforms symbolic expressions at runtime, after type checking.

In contrast to the above approaches, *embedded* DSLs [30] inherit constructs from a host language. Haskell has extensively been used as a host language for embedded DSLs, e.g., Fran [19], FRP [79], FHM [24], Lava [6], and Paradise [3]. Racket [22] is based on Scheme and designed for creating programming languages. Racket uses dynamic typing, but can be extended using libraries, macros, and syntax objects to support a typed variant [73]. To support the benefits of external DSLs in an embedded setting, polymorphic embedding [29] uses virtual types in Scala. A popular embedded language in Scala is Chisel [5], a hardware description language. Scala-Virtualized [56] improves Scala's support for deep embedding by defining built-in constructs as method calls. There are also efforts of combining shallow and deep embedding [69] and to understand the relation to folds [23].

Staging and Partial Evaluation. In multi-stage programs, the execution of certain parts of a program can be delayed to a later stage. MetaML [70] makes use of syntactic stage annotations to separate stages. MetaML has code types `<T>`, which are similar to our symbol types. MetaOCaml [14, 37] implements the MetaML approach in OCaml. Lightweight modular staging (LMS) [57] introduces stages by using the `Rep` type, instead of explicit quasi-quote notation. Several DSLs have been implemented using LMS, including Delite [68]. LMS is similar to our symbol lifting approach in that the type system guides the lifting process. However, the motivation for our work is different. In LMS, staging is used for runtime code generation, whereas we use the symbol lifting to enable seamless embedding [3]. The essence of LMS [55] can be seen as a two-level language [49], where levels are explicitly defined on terms. We distinguish levels by using the three constructs symbolic application, symbolic value, and typed symbols, instead of introducing two levels on all terms. Our approach has also strong relation to partial evaluation [36] and binding-time analysis using type inference [27]. The novelty of our approach is not related to binding-time analysis itself, but rather to gradual typing and its use in DSLs.

Data Types. Our approach is, compared to previous work on open data types [43, 47], simpler and more limited: modules are not separately compiled and patterns are not checked exhaustively. In a series of "scrap your boilerplate" papers, Lammel and Peyton Jones [39–41] show how boilerplate code can be avoided. Axelsson [4] presents the Syntactic library and Jay [35] introduces the pattern calculus. Generalized abstract data types (GADT) [16, 52, 59, 81] can be used in a DSL to ensure well typed terms and type safe transformations.

Gradual Typing. Our work is based on Siek and Taha's [60, 61] approach named gradual typing. This approach gives the guarantee that fully typed programs do not produce runtime type exceptions. The *polymorphic blame calculus* [1, 2] is an extension of Wadler and Findler's [78] blame calculus, where the former combines parametric polymorphism, static, and dynamic typing. Based on this work, as well as that of Igarashi *et al.* [33], we believe that it is possible to extend $\lambda^{<\star>}$ with polymorphism.

Several research works address the problem of *interoperability*. Gray, Findler, and Flatt [25] develop an interoperability semantics between Java and Scheme. Matthews and Findler [44] introduce an operational semantics for interoperability between multi-program languages. Groski *et al.* [26] develops a language SAGE that performs hybrid type checking, and Writstad *et al.* [80] introduce Thorn. Tobin-Hochstadt and Fellisen [71] show how inter-language mitigation can be performed on a module basis, which is the basis for Typed Scheme [72]. The difference to our approach is that our mixing of types is at a finer level of granularity. This expression level control of gradual typing is vital to support our embedded DSL approach, such that the domain expert can "escape" out of static typing when more expressiveness is needed.

6 Conclusions

In this paper, we explain a new approach to embedding DSLs by mixing static and dynamic typing. We have also introduced a host language called Modelyze and evaluated it by embedding equation-based DSLs. The main novelty is the semantics of gradually typed symbolic expressions. We conclude that static typing is definitely important for the model engineer, and that dynamic typing can make it rather easy to extend and reuse functionality for the domain expert.

Acknowledgments

This research is in part financially supported by the Swedish Foundation for Strategic Research (FFL15-0032), by the Swedish Research Council (#623-2011-955), by the ITEA2 OPENPROD project, by the ELLIIT project, by the CHESS center at UC Berkeley, and by NSF Awards 1518844 and 0846121. We thank Peter Fritzson, Thomas Schön, Henrik Nilsson, Walid Taha, Sibylle Schupp, Johan Åkesson, and Michael Zimmer for comments on drafts of this work.

[3]Note that the foundation of the lifting approach presented here was developed in 2010 [9] in parallel and independently of the LMS work.

References

[1] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. 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, 201–214.

[2] Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, With and Without Types. In *International Conference on Functional Programming (ICFP)*.

[3] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. 2008. 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, 225–228.

[4] Emil Axelsson. 2012. 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.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*. ACM, 1216–1225.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. ACM Press, New York, USA, 174–184.

[7] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1 (2008), 52–70.

[8] Martin Bravenboer and Eelco Visser. 2004. 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, 365–383.

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

[10] David Broman and Henrik Nilsson. 2012. 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) (LNCS)*, Vol. 7149. Springer-Verlag, 258–272.

[11] David Broman, Kaj Nyström, and Peter Fritzson. 2006. 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, 151–160.

[12] David Broman and Jeremy G. Siek. 2012. *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*. Technical Report UCB/EECS-2012-173. EECS Department, University of California, Berkeley.

[13] Peter Bunus and Peter Fritzson. 2004. Automated Static Analysis of Equation-Based Components. *SIMULATION* 80, 7–8 (2004), 321–245.

[14] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of Second International Conference on Generative Programming and Component Engineering (GPCE'03) (LNCS)*, Vol. 2830. Springer-Verlag, 57–76.

[15] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543.

[16] James Cheney and Hinze Ralf. 2003. *First-Class Phantom Types*. CISTR TR2003-1901. Cornell University.

[17] Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. 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, 215–226.

[18] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd system–modular extensible compiler construction. *Science of Computer Programming* 69, 1-3 (2007), 14–26.

[19] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming (ICFP '97)*. ACM, New York, NY, USA, 263–273.

[20] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. 1999. Modelica - A Language for Physical System Modeling, Visualization and Interaction. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*.

[21] Hilding Elmqvist and Martin Otter. 1994. Methods for Tearing Systems of Equations in Object-Oriented Modelling. In *Proceedings ESM'94 European Simulation Multiconference*. 326–332.

[22] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (LIPIcs)*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 113–128.

[23] Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices*, Vol. 49. ACM, 339–347.

[24] George Giorgidze and Henrik Nilsson. 2008. Embedding a Functional Hybrid Modelling Language in Haskell. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*.

[25] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. 2005. 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, 231–245.

[26] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. Sage: Hybrid checking for flexible specifications. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*. 93–104.

[27] Fritz Henglein. 1991. Efficient type inference for higher-order binding-time analysis. In *Functional Programming Languages and Computer Architecture (LNCS)*, Vol. 523. Springer-Verlag, 448–472.

[28] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Software* 31, 3 (2005), 363–396.

[29] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*. ACM, 137–148.

[30] Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* (1996).

[31] Iain S. Duff. 1981. On Algorithms for Obtaining a Maximum Transversal. *ACM Trans. Math. Software* 7, 3 (1981), 315–330.

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

[33] Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On Polymorphic Gradual Typing. In *International Conference on Functional Programming (ICFP)*. ACM.

[34] INRIA. 2012. The Caml language: Home. http://caml.inria.fr/ [Last accessed: June 29, 2012].

[35] Barry Jay. 2009. *Pattern Calculus: Computing with Functions and Structures*. Springer-Verlag.

[36] Neil D. Jones. 1996. An introduction to partial evaluation. *Comput. Surveys* 28, 3 (1996), 480–503.

[37] Oleg Kiselyov. 2014. The design and implementation of BER MetaOCaml. In *International Symposium on Functional and Logic Programming*. Springer, 86–102.

[38] Peter Kunkel and Volker Mehrmann. 2006. *Differential-Algebraic Equations Analysis and Numerical Solution*. European Mathematical Society.

[39] Ralf Lämmel and Simon Peyton Jones. 2003. 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, 26–37.

[40] Ralf Lämmel and Simon Peyton Jones. 2004. 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, 244–255.

[41] Ralf Lämmel and Simon Peyton Jones. 2005. 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, 204–215.

[42] Edward A. Lee. 2010. CPS foundations. In *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM Press, New York, USA, 737–742.

[43] Andres Löh and Ralf Hinze. 2006. 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, 133–144.

[44] Jacob Matthews and Robert Bruce Findler. 2009. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.* 31, 3, Article 12 (April 2009), 44 pages.

[45] Sven Erik Mattsson and Gustaf Söderlind. 1993. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 14, 3 (1993), 677–692.

[46] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37 (December 2005), 316–344. Issue 4.

[47] Todd Millstein, Colin Bleckner, and Craig Chambers. 2002. 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, 110–122.

[48] Modelica Association 2017. *Modelica - A Unified Object-Oriented Language for Systems Modeling - Language Specification Version 3.4*. Modelica Association. Available from: http://www.modelica.org.

[49] Flemming Nielson and Hanne R Nielson. 1986. Code generation from two-level denotational meta-languages. In *Programs as Data Objects*. Springer, 192–205.

[50] Henrik Nilsson. 2008. 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, 71–81.

[51] Constantinos C. Pantelides. 1988. The Consistent Initialization of Differential-Algebraic Systems. *SIAM J. Sci. Statist. Comput.* 9, 2 (1988), 213–231.

[52] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*. ACM Press, New York, USA, 50–61.

[53] Gordon D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report. Department of Computer Science, University of Aarhus.

[54] Tiark Rompf. 2016. Reflections on LMS: Exploring Front-end Alternatives. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. ACM, New York, NY, USA, 41–50.

[55] Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World*. Springer, 318–335.

[56] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* 25, 1 (2012), 165–207.

[57] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136.

[58] Tim Sheard and Simon Peyton Jones. 2002. Template Metaprogramming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM Press, New York, USA, 1–16.

[59] Tim Sheard and Emir Pasalic. 2008. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science* 199 (2008), 49–65.

[60] Jeremy Siek and Walid Taha. 2006. Gradual typing for functional languages. In *In: Scheme and Functional Programming Workshop*.

[61] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European conference on ECOOP 2007: Object-Oriented Programming (LNCS)*, Vol. 4609. Springer-Verlag, 2–27.

[62] Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *European Symposium on Programming*.

[63] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL: Summit on Advances in Programming Languages (LIPIcs: Leibniz International Proceedings in Informatics)*.

[64] Jeremy G. Siek and Philip Wadler. 2009. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution (STOP '09)*. ACM, New York, NY, USA, 34–46.

[65] Guy L. Steele. 1990. *Common LISP. The Language* (2nd ed.). Digital Press.

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

[67] Guy L. Steele, Jr. 1982. An overview of COMMON LISP. In *Proceedings of the 1982 ACM symposium on LISP and functional programming (LFP '82)*. ACM, New York, NY, USA, 98–107.

[68] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4s, Article 134 (April 2014), 134:1–134:25 pages.

[69] Josef Svenningsson and Emil Axelsson. 2012. Combining deep and shallow embedding for EDSL. In *International Symposium on Trends in Functional Programming*. Springer, 21–36.

[70] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1–2 (2000), 211–242.

[71] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. 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, 964–974.

[72] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. 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, 395–406.

[73] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. ACM Press, New York, USA, 132–141.

[74] Laurence Tratt. 2008. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 31 (Oct. 2008), 40 pages.

[75] Todd Veldhuizen. 1995. Using C++ template metaprograms. *C++ Report* 7, 4 (1995), 36–43.

[76] Philip Wadler. 1992. Comprehending monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493.

[77] Philip Wadler and Stephen Blott. 1989. 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, 60–76.

[78] Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *European Symposium on Programming (ESOP) (LNCS)*, Vol. 5502. Springer-Verlag, 1–15.

[79] Zhanyong Wan and Paul Hudak. 2000. 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, 242–252.

[80] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. 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, 377–388.

[81] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, USA, 224–235.