Almost First-Class Language Embedding

Taming Staged Embedded DSLs

Maximilian Scherr
The University of Tokyo, Japan
scherr@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo, Japan
chiba@acm.org

Abstract

Embedded domain-specific languages (EDSLs), inheriting a general-purpose language's features as well as look-and-feel, have traditionally been second-class or rather non-citizens in terms of host-language design. This makes sense when one regards them to be on the same level as traditional, non-EDSL library interfaces. However, this equivalence only applies to the simplest of EDSLs. In this paper we illustrate why this is detrimental when moving on to EDSLs that employ staging, i.e. program reification, by example of various issues that affect authors and users alike.

We believe that if EDSLs are to be considered a reliable, language-like interface abstraction, they require exceptional attention and design scrutiny. Instead of unenforceable conventions, we advocate the acceptance of EDSLs as proper, i.e. almost first-class, citizens while retaining most advantages of pure embeddings. As a small step towards this goal, we present a pragmatic framework prototype for Java. It is based on annotations that explicate and document membership to explicit EDSL entities. In a nutshell, our framework identifies (annotated) method calls and field accesses as EDSL terms and dynamically constructs an abstract-syntax representation, which is eventually passed to a semantics-defining back end implemented by the EDSL author.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs & Features

Keywords Java, implementation, embedded DSLs, metaprogramming, program transformation, staging, programming languages, design

1. Introduction

The embedding of *domain-specific languages* (DSLs) within the confines of a general-purpose host language is an active area of research [16, 22, 25, 30]. By inheriting large parts of the host-language infrastructure [19] these *embedded DSLs* (EDSLs) manage to reduce the effort of creating a dedicated parser, compiler, or interpreter tool chain as well as enable easy usage alongside general-purpose code [13, 19].

While the idea of using a general-purpose language primarily as a sort of scaffolding [11, 25, 29] is an interesting take on modular (domain-specific) language prototyping and development, arguably EDSLs have found their way into the mainstream by way of providing rich interfaces to programming libraries, taking the place of traditional idioms and state-based interactions with libraries [1, 2, 13, 15]. Even the official Java API has started to embrace EDSLs: Now many forms of manual iterations can be replaced with more readable combinations of `map`, `filter`, and other methods. These ideas have long been in use and found recognition as EDSLs in the functional programming language community [20].

At the time EDSLs (then named DSELs) were described by Paul Hudak [19] the concept stood in contrast to manual preprocessing or compile-time metaprogramming approaches. In combination with the right host language, so-called *pure embedding* was seen as sufficient. Indeed, embedding DSLs in terms of run-time behavior is widely available, lightweight, and reliable, while at the same time compile-time properties can be exploited to reinforce language characteristics, e.g. EDSL syntax via static typing [37].

Recently, a trend to further augment these pure embedding approaches with additional pre-run-time support has emerged [5, 18, 22, 25, 30]. This trend indicates that EDSLs are beginning to be considered worthy of special treatment. We believe the time has come for accepting language embedding as a staple for the development of rich library interfaces, deserving its own linguistic-abstraction mechanism to achieve improved reliability, performance, and usability. We contribute to the state of the art as follows:

- A badly designed, manually implemented EDSL may easily devolve into a convoluted, inconvenient, and unreliable programming interface merely upholding the illusion of a language. To illustrate this we present a collection of pitfalls potentially encountered due to the process of *staging*, here referring to EDSL-program reification.
- We propose that a meta-level or semi-linguistic abstraction can ensure safe and predictable EDSL usage. At the same time it ought to reduce development effort and enable features hard to accomplish with only general-purpose mechanisms. This abstraction would raise language embedding close to a first-class language feature.
- Our concrete solution is a pragmatic framework for Java based on load-time metaprogramming. It makes use of annotations that, together with controlling the atomic staging operations, help communicate an EDSL's behavior to its users. It further helps simplify EDSL implementation and allows for a mixing of explicit and hidden dynamic staging styles.

2. Pitfalls of Manual Language Embedding

The most straightforward approach to implement an EDSL is to use a direct mapping from its syntax to its semantics in the host lan-

GPCE'15, October 26–27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3687-1/15/10...
http://dx.doi.org/10.1145/2814204.2814217

guage. Especially when we only deal with values that do not represent computation themselves it is hard to distinguish an EDSL from a traditional library interface. It is merely a question of perspective.

```
1 public final class Vec { ···
2   public Vec plus(Vec v)  { ··· /* el.-wise addit. */ }
3   public Vec times(double s) { ··· /* scalar mult. */ }
4 }
```

Figure 1. Non-staged vector EDSL

Without dwelling on a discussion on what exactly constitutes an EDSL, let us consider the skeleton shown in Figure 1, representing the definition of a type and an EDSL for vector operations. It is easy to implement and use since the individual EDSL constructs or *tokens* are implemented directly using *procedural abstraction*.

Not only are result values immediately available when needed, users of this EDSL can also rely on the general properties of the employed abstraction mechanism such as argument evaluation strategy and error handling. Take for instance the following expression:

```
a.plus(b.times(2.0)).plus(c)
```

The order of evaluation is clearly defined by the host language. Furthermore, if for instance value `a` is a vector of different dimension than `b`, a run-time exception can be thrown to clearly indicate where the issue has occurred.

Despite its ease of use, such a so-called *shallow embedding* is severely limiting since an EDSL program, or rather its abstract syntax, only ever exists implicitly [30]. This precludes structural optimizations as well as alternative interpretations such as the printing of an expression [16, 32].

Making the abstract syntax of EDSL programs explicit can be achieved by defining the EDSL constructs to build (and compose) elements of an *intermediate representation* (IR). When this is performed at run time we are dealing with a so-called *deep embedding* [16, 32], a relative of the *interpreter pattern* [14]. Alternatively, explicating the abstract syntax can be achieved using metaprogramming techniques such as syntactic macros or other forms of preprocessing that occurs before a user program is executed [5, 10, 22, 30].

The unifying property of such EDSL implementations is the fact that the handling of an EDSL program is delayed (as opposed to immediate) for the purpose of enabling beneficial processing. We refer to this property as staging[1] and EDSLs that use it as *staged EDSLs*. Conceptually, the workflow of using a staged EDSL can be divided into phases, which in practice sometimes overlap:

1. *Staging*, i.e. IR construction and composition.
2. *Processing*, e.g. optimization, compilation, etc.
3. *Materialization* or *Unstaging*, i.e. evaluation to a result.

The term "materialization" refers to the notion that EDSL programs abstractly represent a yet immaterial value (or effect). In preprocessing implementations this phase is commonly non-existent or would turn into a sort of preparation for eventual evaluation.

```
1 public final class Vec { ··· // base value
2   public VecE toVecE()      { ··· /* value lifting */ }
3 }
4 public class VecE { ···      // abstract syntax / IR
5   public VecE plus(VecE vE)       { ··· /* staging */ }
6   public VecE times(double s)     { ··· /* staging */ }
7   public Vec  toVec()     { ··· /* materialization */ }
8 }
```

Figure 2. Deeply embedded vector DSL

[1] Taha sates that "in essence, staging is altering a program's order of evaluation in order to change the cost of its execution" [34, Chapter 2]. Arguably, this applies both to run time as well as compile time [10].

Figure 2 shows a skeleton definition of a deeply embedded version of the example vector DSL. Here the method `Vec.toVecE` would perform trivial staging by lifting base values, `VecE.plus` and `VecE.times` would stage and compose IR elements (or expressions, hence `VecE`), and `VecE.toVec` would perform materialization. Assuming `aE`, `bE`, `cE`, and `dE` are lifted values of type `VecE`, the expression presented earlier would look as follows:

```
aE.plus(bE.times(2.0)).plus(cE).toVec()
```

Users may now (by contract or convention) expect delayed processing. Unlike in our earlier example, conceptually the abstraction of domain-specific computation does not happen on the individual method-call level anymore but on the higher level of dynamically constructed expressions belonging to a language.

However, in the case of manual deep embedding this is only *emulated* by using lower-level procedural abstraction. Similar applies to other forms of staged-EDSL implementations relying on manual forms of metaprogramming. In the following we will look at several pitfalls potentially encountered due to this gap between the conceptual and real abstraction level.

2.1 Non-Compositional Staging

With manual deep embedding the abstractions in charge of IR construction are free to realize staging in any possible way. Unfortunately this also means that EDSL users must not make across-the-board assumptions about the staging process. This includes the expectation of *compositionality* for EDSL-program construction.

```
1 static void example(Vec a, Vec b, Vec c) {
2   VecE aPlusB = a.toVecE().plus(b.toVecE());
3   out.println(aPlusB.toVec());
4   VecE aPlusBtimes5 = aPlusB.times(5.0);
5   out.println(aPlusB.toVec());
6   out.println(aPlusBtimes5.toVec());
7 }
```

Figure 3. Deeply embedded vector DSL usage example

Consider the example in Figure 3 and let us assume for the sake of argument that `Vec` instances are immutable and `toVec` performs a side-effect-free and sound execution of constructed EDSL programs. We still cannot draw the conclusion that lines 3 and 5 yield the same output since the EDSL programs to be executed might differ. As contrived as this may seem, a non-compositional-staging implementation is easily sketched as shown in Figure 4.

```
1 public class VecE {
2   private final Vec base;
3   private final List<Consumer<double[]>> ops = new ···;
4   VecE(Vec base) { this.base = base; }
5   public VecE plus(VecE vE) {
6     ops.add(ds -> { ··· /* add vE[i] onto ds[i] */ });
7     return this;
8   }
9   public VecE times(double s) {
10    ops.add(ds -> {
11      for (int i = 0; i < ds.length; i++) {
12        ds[i] *= s;
13      }
14    });
15    return this;
16  }
17  public Vec toVec() {
18    double[] ds = Arrays.copyOf(base.elements, ···);
19    for (Op op : ops) { op.accept(ds); }
20    return new Vec(ds);
21  }
22 }
```

Figure 4. Non-compositional staging

Note that it is common to implement so-called *builders* [7, Chapter 2] in a similar fashion. With builders the general consensus may lie on repeatedly mutating an intermediate data structure

from which to subsequently construct a final (often immutable) one. However, with staged EDSLs it is arguably more reasonable for the goal and consensus to lie on the safe construction and assembly of EDSL-program fragments. It is quite surprising that non-compositionality manifested in the form of self-mutation with a **`return this`** statement seems to be the current norm or principle for *fluent-interface* [12, Chapter 4] EDSL designs.

One easy way for EDSL authors to ensure compositionality is to choose immutable, persistent data structures for the IR and avoid side effects within the staging (EDSL-token) methods. However, note that this is not a definitive necessity. It may very well make sense to judiciously employ side effects, for instance in order to achieve a form of *common-subexpression elimination* (CSE) [27], as long as compositionality is not impaired.

2.2 Opaque Workflow

Closely related to (yet not interdependent with) non-compositional staging is the issue of unpredictable existence, division, or overlap between the phases of the staged-EDSL workflow. For instance, naming conventions and type signatures may indicate the roles of methods and their workflow phases, but their actual behavior has to be inferred from documentation or source code (if present).

For instance, it is not even established that the methods in `VecE` do in fact perform IR construction. They might just trivially wrap `Vec` values and act as proxies to the shallow embedding. The same applies to materialization. For instance, in our vector EDSL, materialization could be delayed further, e.g. until one of its elements is accessed. Note that this can even affect seemingly shallowly embedded DSLs. Ultimately, for users it is not immediately obvious what kind of embedding they are dealing with.

```
1 public VecE plus(VecE vE) {
2   if ( /* this is the 5th addition in a row */ ) {
3     return this.toVec().toVecE().plus(vE);
4   }
5   return /* default IR construction */;
6 }
```

Figure 5. Opaque workflow-phase overlapping example

To illustrate the issue with opaque overlapping of workflow phases consider Figure 5. The `plus` method here performs conditional materialization. In general, any time IR elements are handed to an EDSL-token method (at staging time) it could potentially entail an internal decision to (partially) materialize. This is an issue for users since they cannot clearly anticipate and decide when a potentially costly (or effectful) computation is initiated, impairing program understanding and design.

Another form of overlap with similar consequences is encountered with on-the-fly processing. In practice, a common approach to optimizations is to perform IR transformations by rewriting *within* the staging methods [11], e.g. to remove multiplication with a constant `1`. While this is a clever idea, it may hinder the reusability of EDSL snippets and complicate debugging. Furthermore, certain optimization opportunities may be lost: Ones that rely on inspecting the entire constructed EDSL program in its original form.

2.3 Fuzzy Language Boundaries

Conceptually, *language boundaries* appear when terms of an EDSL are intermixed with those of different EDSLs or those of the general-purpose parts of the host language. Of course, since with manual implementations EDSLs are not explicit entities, neither are language boundaries.

This may lead to issues of unclear delimitation, which become further pronounced when staging itself occurs in a hidden fashion. This *hidden staging* is sometimes desired for the sake of simplifying EDSL usage [22, 32]. One generic host-language solution that enables it is compile-time metaprogramming via syntactic macros as found for instance in Lisp dialects. Note that this effectively achieves *static staging* as opposed to the *dynamic staging* seen with deep embedding. However, even in the case of traditional deep embedding, it is possible to make the staged nature of an EDSL less explicit by not fully exploiting static type checking, by design of the host language or that of constituent EDSLs.

Despite potential advantages for EDSL usability, the described concealment makes the communication (to users) as well as the enforcement of language boundaries hard. After all, the secondary general-purpose mechanisms that could emulate language boundaries are abandoned.

Consider the following expression, using an implementation of the vector EDSL using Lisp-style syntactic macros:

```
(times (times '(0 0) (e)) (+ (f) (* x y)))
```

One could provide a language labeling of the above where H stands for the host language. Let us here consider arithmetic operations as part of their own language:

$$\overbrace{(L_1\ \ (L_1\ \ H\,(H\ \ H)\ \ (H))\ \ \underbrace{(L_2\ \ (H)\ \ (L_2\ \ H\ \ H))}_{s_2})}^{s_1}$$

The implementation of `times` might decide that `'(0 0)` is indifferent to scalar multiplication and remove the multiplication. This is a problem if `(e)` or `(f)` cause side effects that might affect the rest of the program. Likewise, the implementation might fuse the arithmetic operations of L_2 or decide to interpret them differently, e.g. with (or without) overflow checks, again affecting more than L_1 should. In general, an inner-scope expression (s_2) cannot prevent its tampering by an outer-scope macro (s_1): Assuming L_2 here to be a macro-based EDSL implementation, it cannot force L_1 to leave its programs alone. Syntactic macros (even more limited varieties than Lisp's) bring about these and other issues due to their inherent power [17, Chapter 10].

This pitfall can also be encountered with deep embedding when type checking is not fully exploited. In that case IR elements can be handled by any IR-processing function since language boundaries only exist to the extent of ignorance (i.e. one language not knowing another's IR). However, at least the effect cannot reach into general-purpose code. Of course, one may readily blame individual sloppy implementation when problems with fuzzy language-boundaries occur. However, this only supports our claim that lower-level abstractions are unreliable tools for language embedding.

2.4 Loss of Static Context

Dynamic staging makes it hard to associate an IR element with the static context (e.g. source location) in which it was constructed. One use case for this would be debugging: While at the time of processing one can detect the source of an error *within* an EDSL program, this alone is of little help to EDSL users who want to investigate where and why a bug was introduced at staging time.

There may be ways to retrieve and conserve such knowledge with support of the language runtime, e.g. by inspecting stack traces in IR-constructing methods, but they may slow down staging and ought to be considered crude workarounds. Here, using syntactic macros may help, e.g. in the case of Scala where such context information is retrievable. However, one could claim yet again that using a macro system may introduce uncertainty as described in section 2.3. Some approaches [25] propose the addition of language features for this purpose.

For entirely statically staged EDSLs this pitfall is less of a concern since the disconnect between the point of staging and processing is not as drastic and context information may be readily available.

2.5 Redundant Processing

It is reasonable to assume that EDSL-program processing involves a non-negligible amount of computational effort. Some reified programs may undergo significant analysis and optimization or are even compiled and offloaded. In general, this is worthwhile when the computation during materialization would run a lot slower without prior processing, e.g. when dealing with large data sets.

An advantage of dynamic staging is that these programs are specialized based on EDSL-external, user-defined conditions. One may look at dynamic staging as a form of configuration of a library's implementation at run time.

```
1 static void example(boolean b, Vec v) {
2   VecE e = v.toVecE().plus(v.toVecE());
3   if (b) { e = e.plus(v.toVecE()); }
4   out.println(e.toVec());
5 }
```

Figure 6. Vector EDSL redundancy example

If handled naively this is bound to cause repeated processing overhead. For instance, consider frequent calling of the method in Figure 6. Only two different EDSL programs (with different `v` inputs) are ever generated here. A manual deep-embedding implementation cannot exploit this knowledge. At best, it could rely on manual caching, e.g. in the `toVec` method. However, such a solution complicates the EDSL's implementation and design despite its being a common aspect of the higher-level EDSL abstraction itself.

It should be easy to see how ignoring this concern may either be detrimental to the run-time performance or alternatively, unnecessarily limits the scope and usefulness of the EDSL as a rich library interface (due to users needing to circumvent the issue).

For statically staged EDSLs this is mostly a non-issue, i.e. when processing is limited to one time only.

3. Our Solution

To tackle the described pitfalls we propose to augment host languages with dedicated support for language embedding.

The first step towards this goal is to explicate the concept of an embedded language itself. Doing so enables us to clarify language boundaries and configurations. Secondly, clearly marking the constructs of an EDSL allows us to hand over the responsibilities for IR construction entirely to the dedicated EDSL-abstraction mechanism. This makes staging and materialization follow uniform, predictable rules whether a hidden or more obvious staging style is used. Finally, the processing of staged-EDSL programs is abstracted away and cached by indirection to a common interface implemented by the respective EDSL authors.

We have implemented this in the form of a prototype framework for Java. However, there is little preventing the adoption of the presented ideas in other host languages. In the following we will first concentrate on the framework's usage and meaning before delving into a discussion on implementation concerns.

3.1 The `@Stage` Annotation

Our framework hinges on the ability to discern between staged and non-staged parts of a program. By placing an `@Stage` annotation, whose declaration is shown in Figure 7, on a field or method declaration, EDSL authors can designate these to become tokens.

```
1 @Documented @Target({ METHOD, FIELD })
2 public @interface Stage {
3   Class<? extends Language<?>> language();
4   boolean isStrict() default false;
5   StaticInfo.Element[] staticInfoElements() default {};
6 }
```

Figure 7. `@Stage` annotation type declaration

Such a token is always associated with a *language*, represented by a Java class marked with the `Language` interface. This association is used for language-boundary checks and indirection to IR processing. The `isStrict` (immediate materialization) and `staticInfoElements` (specifying additional static information to be retained) options will be revisited in more detail later.

From a user's operational perspective, when they invoke an `@Stage`-annotated method (or likewise, access such a field) the original Java behavior is ignored and instead an abstract representation (of type `Expression.Staged`, E_S) is constructed. Our framework accomplishes this by transforming user programs accordingly prior to execution.

Composing complex terms within the same language is trivially understood. However, foreign terms undergo a prior conversion step: Host-language ones are always externalized as input values and lifted (to an `Expression.Value`, E_V), and by default, terms of a different embedded language are materialized as values and subsequently lifted as well.

```
1 public final class Vec { ···
2   @Stage(language = VecL.class)
3   public Vec plus(Vec v)          { ··· /* ignored */ }
4   @Stage(language = VecL.class)
5   public Vec times(double s)      { ··· /* ignored */ }
6 }
```

Figure 8. Annotated `Vec`

To illustrate the effect of using `@Staged` language constructs, consider the annotated vector EDSL presented in Figure 8 together with the example in Figure 9, which shows a simplified, conceptual description of the IR construction that will occur at run time: The $\rightsquigarrow$ symbol indicates the result (i.e. what actually happens at run time) of having transformed the line above it (i.e. what the user wrote). The double brackets ($[\![\cdot\cdot]\!]$) indicate materialization.

Reading and writing of local variables transfers the staged-EDSL terms as is. Hence, the resulting IR is a *directed acyclic graph* (DAG) of `Expression` instances, whose structure follows from the bindings in the user program.

```
1  static void sink(Vec v0, Vec v1, Vec v2) { ··· }
2  static Vec example(int n, Vec a, Vec b, Vec c) {
3  ○ a = a.plus(b);
   ○ ⇝ E t_a = E_S(VecL, plus, [E_V(a), E_V(b)]);
4    if (n > 10) {
5    ○ a = a.plus(a);
     ○ ⇝ t_a = E_S(VecL, plus, [t_a, t_a]);
6    }
7  ○ sink(a, b, a);
   ○ ⇝ sink([[t_a]], b, [[t_a]]);
8  ○
   ○ ⇝ E t_c = E_V(c);
9    for (int i = 0; i < n; i++) {
10   ○ c = c.times(5 + i);
     ○ ⇝ t_c = E_S(VecL, times, [t_c, E_V(5 + i)]);
11   }
12 ○ return c;
   ○ ⇝ return [[t_c]];
13 }
```

Figure 9. Usage and transformation example

Staging is hidden and occurs dynamically at run time. Arguably, especially in light of the existence of potential side effects in the host program, this kind of staging would seem like a recipe for disaster. The following two aspects make it not so:

- The `@Stage` annotation and its configuration provide sufficient documentation so that even with only basic tool support (e.g. Javadoc, quick jumps to definitions in an IDE, etc.) users can understand which parts of their programs are subject to staging as part of an EDSL.

- The scope of staging or rather of staged-IR transfer is limited to the body of the method (for now). We say the staged terms are *locally carried*, which means that they either remain (externally) unused or get materialized when externally used (e.g. by `sink`), i.e. upon attempting to "leave" the method's local scope. We will introduce *globally carried* terms in Section 3.4.

3.2 Materialization Triggers

For locally carried terms, materialization is triggered lazily when encountering a language boundary. At the call `sink(a, b, a)` in Figure 9 (l. 7), materialization of the first occurrence of `a`'s corresponding term t_a is triggered before the second. Similar to thunks in other on-demand evaluation schemes, in our framework EDSL-term instances are materialized only once. Subsequent materializations yield cached result values.

By the nature of this scheme, materialization will by default only happen when a term is actually needed (and not at all if unused) EDSL-externally. This behavior can be changed with the `isStrict` setting on an `@Stage`-annotation instance. Setting it to **`true`** causes immediate materialization where the so-annotated language construct occurs. Methods with **`void`** return type always show this behavior.

Above we illustrated the frequent but special case of host-language boundaries. Boundaries between two different staged EDSLs, or rather between their terms, need different handling. They occur when a term of an embedded language L_1 takes an argument of an embedded language L_2 that is not *accepted* (see Section 3.5) at the given parameter position. By default a term accepts terms of its own language. Trivially staged, lifted values (E_V) are always accepted and for them materialization merely means unlifting, i.e. retrieving the contained, original value.

In general, the language of a staged-EDSL-term argument cannot always be statically determined. Consequently, the decision on boundary-based materialization triggering is made dynamically.

3.3 From Expression DAGs to Values

Materialization involves intermediate processing that turns staged abstract syntax into meaningful computation. In short, EDSL authors have to write a compiler similar to the one sketched in Figure 10. However, there is quite a bit of scaffolding whose understanding is necessary for this code example to make sense.

The first waypoint to materialization is the creation of a closure object representing the EDSL program's computation. Only its eventual evaluation is to yield the result of materialization. This indirection allows us to cache (or preprocess) the closures associated with EDSL programs that only differ in their lifted dynamic input values (E_V) but not their particular shape. The term "shape" here refers to a specific composition of staged-EDSL constructs (E_S). Consider the following term (where L stands for `VecL`):

$$E_S(L, \texttt{times}, \big[E_S(L, \texttt{plus}, [E_V(x), E_V(y)]), E_V(z)\big])$$

Its shape ranges over all possible value instantiations for x, y, and z. When the materialization of a staged-EDSL program is triggered we first probe a cache for an existing mapping between an isomorphic expression DAG to a closure object: If present the existing closure is reused, if absent the author-defined processing mechanism is invoked. The latter causes dispatch to a static method on the language (class) of the expression DAG's root node. It follows a naming convention and type signatures as follows:

- `makeObjectClosure(Staged, Binder, boolean)` if the (return) type of the root-node construct is a reference.
- `makeDoubleClosure(Staged, Binder, boolean)` if the (return) type of the root-node construct is **`double`**.
- Similarly for the other primitive Java types.

EDSL authors implement these methods as needed at their discretion while minding the fact that the results are cached: Processing ought only to depend on a given expression DAG's shape and not some external state. The supplied **`boolean`** argument indicates whether the resulting closure will be *permanently cached* (i.e. memoized). This is the case when the shape is statically known.

```
1  class VecLCompiler implements Expression.Visitor {
2    private final Binder binder;
3    private ObjectClosure<Vec> closure;
4    ···
5    public void visit(MethodInvocation staged) {
6      staged.getArgument(0).accept(this);
7      ObjectClosure<Vec> a0 = closure;
8      switch (staged.getMember().getName()) {
9        case "plus": {
10         staged.getArgument(1).accept(this);
11         ObjectClosure<Vec> a1 = closure;
12         closure = env -> {
13           Vec v0 = a0.evaluate(env);
14           Vec v1 = a1.evaluate(env);
15           double[] ds = new double[v0.elements.length];
16           for (int i = 0; i < ds.length; i++) {
17             ds[i] = v0.elements[i] + v1.elements[i];
18           }
19           return new Vec(ds);
20         }
21         break;
22       }
23       case "times": ···
24     }
25   }
26   public void visit(ObjectValue value) {
27     closure = value.bind(binder);
28   }
29 }
```

Figure 10. Trivial compiler skeleton

To accomplish the separation of expression-DAG shape and concrete instances, lifted-value nodes cannot be inspected for their actual contents (without disabling caching) unless they are constants. Instead, the supplied `Binder` instance is used to create value-access closures from lifted-value nodes (see Figure 10, l. 27).

Caching and reusing closures requires injecting the actual input-value instances as arguments at materialization time. The interface methods (`evaluate`) of the closure interfaces take only a single `Environment`-typed argument housing the input values found in the expression DAG to be materialized. These *environments* are isomorphic in the same way cached expression graphs are. For instance, an environment could be backed by an array of values with the binder-generated value-access closures being mere indices.[2]

Finally, the reader may revisit Figure 10, showing a part of a trivial compiler for the vector EDSL of Figure 8 that does not implement optimizations. Note that while this shows translation using the *visitor pattern* [14], our framework is not concerned with how traversal occurs in detail. For instance, we also created a support package for implementations in Scala using *pattern matching*.

3.4 Global Carrying

Our description until now still only covers hidden staging limited to the local scope of a method's body, i.e. local carrying. Not having to be confronted with concerns of explicit staging can be good for usability. Yet, sometimes a more obvious and *global* style of staging as traditionally seen with deep embedding is desirable. For instance, this is arguably better suited for EDSLs that deal with mutable input than a hidden style as materialization is explicitly triggered. Furthermore, in some languages, not every fragment bears meaning on its own. Local carrying is problematic for these

[2] We found that making these access closures simply graph-walk to value nodes turns out to be a faster solution.

cases since every single (locally carried) term may hit a language boundary and cause its materialization.

In the following we will describe how a small extension to the framework described so far enables obvious deep embedding by permitting terms to be *globally carried*. Figure 11 shows global carrying in action. The term in `rE` is carried over to `times2`, which acts as a sort of macro expansion at run time. Leaving `example`'s scope does not cause automatic materialization.

```
1 static VecE times2(VecE vE) { return vE.times(2.0); }
2 static Vec example(int n, Vec a, Vec b, Vec c) {
3   VecE rE = a.toVecE().plus(b);
4   Vec r = times2(rE).toVec();
5   if (n > 5) { return c; }
6   return r;
7 }
```

Figure 11. Global carrying example

With traditional deep embedding, EDSL authors define their own types (like `VecE`) and data structures for representing abstract syntax. In our framework EDSL authors can instead just declare a type extending the `GlobalCarrier` class. An instance of such a *global carrier* can have two meta states: Either it stands for an actual value of the data type as defined at the author's discretion, or it stands for abstract syntax, i.e. it literally *carries* an expression DAG. Hybrid states are possible but usually not recommended.

Defining an (`@Stage`-annotated) EDSL construct with (return) type `GlobalCarrier`, or preferably a subtype thereof, suffices to signal that the returned instance shall carry the constructed term. The code in Figure 12 completely defines `VecE` and adds relevant annotations. When an `@Stage`-annotated construct receives a carrier-typed argument this will first be checked for carrying a staged term and if so move on to language-boundary checks.

```
1 public final class Vec { ···
2   @Stage(language = VecEL.class)
3   public VecE toVecE()            { ··· /* ignored */ }
4 }
5 public final class VecE extends GlobalCarrier {
6   @Stage(language = VecEL.class)
7   public VecE plus(VecE vE)       { ··· /* ignored */ }
8   @Stage(language = VecEL.class)
9   public VecE times(double s)     { ··· /* ignored */ }
10  @Stage(language = VecEL.class)
11  public Vec toVec()              { ··· /* ignored */ }
12 }
```

Figure 12. `VecE` as a global carrier

Recall that with the default local carrying, materialization is triggered implicitly on demand. Globally carried terms do not exhibit this behavior. Instead, materialization is inducible in two ways:

- By using a strict (`isStrict`) EDSL construct designated to be an explicit trigger (as usual in traditional deep embedding).
- By transitioning to a locally carried construct, which will lead to subsequent implicit materialization.

Let us interpret Figure 11 with the latter behavior, i.e. `toVec` returns `Vec`, a locally carried value. Since `toVec` is not strict there is only one point where materialization could happen: At the **`return`** statement in line 6 if it is reached during execution. Note

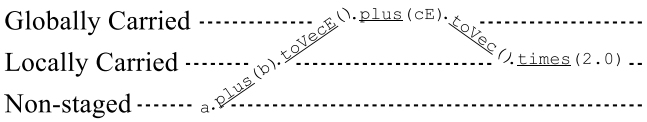

Figure 13. Carrying-level transitioning

that thanks to the way we named the methods and structured the EDSL we could simply reuse the compiler of Figure 10.

The local and global carrying scheme affords an interesting design opportunity: If instead of `VecEL.class` we use `VecL.class` as language in the annotations, we can intermix local (i.e. methods on `Vec`) with global carrying, as shown in Figure 13. Since there are no language boundaries besides the host-language one, the composition will result in a term that can be handled by the the same compiler. While seeming contrived here, a use case for such an arrangement may arise in EDSLs where some constructs require explicit handling by users while others do not.

3.5 Further Features and Considerations

Having covered the basics of our framework let us introduce additional features helpful for design, implementation, and usage.

Language Boundary Customization. For the sake of modularity it is desirable to make EDSL syntax composable. This requires a means for changing the default boundary behavior of terms only accepting language constructs of the same language. To this end we provide the `@Accept` annotation which is to be placed on the parameters of `@Staged` constructs. Its only element (`languages`) is an array of language classes that will be interpreted as the set of languages accepted at the given parameter.

Suppression of Staging Behavior. In our examples we have omitted the method bodies of `@Staged` EDSL tokens. We did so because their implementation was not relevant: Using such a token is not linked to its original implementation anymore.

However, we found that there are use cases where EDSL users might want to switch between a staged and a non-staged interpretation, e.g. for benchmarking. Of course, the EDSL author has to account for this case, i.e. implement the method bodies.

Staging can be suppressed with the `@Suppress` annotation. Like `@Accept` it takes an array of language classes for which suppression should be effective. It can be placed both on methods as well as types and is effective in their scopes (but not inherited).

Visibility Control. An EDSL author might not want users to suppress staging of their language. Likewise, allowing just anyone to add constructs to an existing language with the `@Stage` annotation is dangerous. After all, the author of the EDSL should not have to consider additional language constructs when building their compiler. Similar issues apply to `@Accept`.

A simple solution is to rely on Java's visibility control and set the language class non-public. However, sometimes this might be too drastic, for instance when suppression should be allowed "publicly" but none of the other annotations. To this end, we added yet another annotation, called `@Configure`, that may be placed on a language class. Using this an EDSL author can individually set whether its access is restricted for `@Stage`, `@Accept`, or `@Suppress`. This configuration only affects usage outside the package of the language class.

Static `@Stage` Inheritance. Our framework's interpretation of a construct as staged or not is static, i.e. early bound. This simplifies implementation and arguably increases predictability and consistency for EDSL users: They can statically determine whether staging will occur and which language a token belongs to.

Inheritance of `@Staged` methods is allowed. For the sake of consistency it behaves similar to visibility-inheritance rules in Java: Just like a method's visibility cannot be reduced by inheritance, so can an `@Staged` construct not become non-staged. We apply this to fields as well for consistency. However, we currently do allow changing languages or `@Accept` options as it might prove useful for specialization on a lower level of a class hierarchy. We have yet to investigate whether predictability should outweigh this freedom.

Static Information. Our framework automatically exploits static knowledge and provides some of it to EDSL authors, e.g. constant input values. Retaining additional information from the static context in which an EDSL term was constructed can be very useful for debugging, error reporting, and optimization. To capture some of this information, the `staticInfoElements` value of the `@Stage` annotation may be set by EDSL authors.

Currently, our framework only supports collecting two types of information: The origin (`ORIGIN`), i.e. method, line number, and bytecode position of a constructed term[3], and the inferred types (`INFERRED_TYPES`) of a token's arguments and (return) type.

3.6 Behind the Scenes

Ideally, the EDSL-abstraction mechanism provided by our framework would be supported natively by the compiler and language runtime. However, such a heavyweight extension is neither easy to implement and maintain, nor is it attractive to current users (of Java). Instead, we have pragmatically implemented our framework to perform on-the-fly transformations on client code.

The Case for Load Time. As in our previous work [30], we opted to use load-time metaprogramming using a Java agent in combination with Javassist [3, 9]. The reasons are manifold: For one, our framework's run-time support needs to make certain methods public and adapt certain classes without leaking this nature to source files. Performing these adaptations and transformations on the fly keeps source-level tampering (by EDSL users and authors) at bay.

Furthermore, by transforming bytecode as it is loaded we are delaying processing as long as possible. This fits very nicely into the rest of the design rationale, i.e. on-demand processing. Further advantages will become clear as we proceed through this section.

Overview. When a new class is loaded into the system we scan its methods for containing usages of `@Staged` constructs. If present we perform proper data-flow analyses in the following order:

1. *Type-inference analysis* yields type approximations.
2. *Value-flow analysis* relates sources and uses of values.
3. *Constant analysis* marks values as constant (or not).
4. *Stage analysis* builds a *stage graph* from `@Staged` tokens.
5. *Weave analysis* determines bytecode adaptation.

These are simple iterative forward-flow data-flow analyses [23] whose detailed discussion is beyond the scope of this paper.

Everything is finalized with a weaving step that performs the necessary transformations before the class is handed to the class loader. The agent that does all that is contained in a JAR file to be used with the `-javaagent:···` JVM option.

The Stage Graph. The stage analysis yields a graph that represents both the data and control flow between the staged-EDSL constructs contained in a method and also external sources and uses.

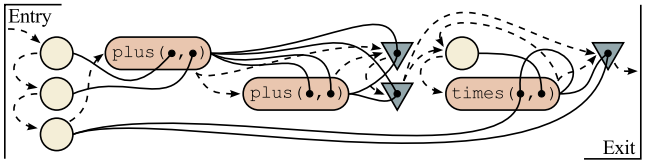

Figure 14. Stage graph for Figure 9

Figure 14 shows an example graph for the code in Figure 9 of Section 3.1. Rounded rectangles represent EDSL terms, triangles represent their host-language uses, and circles represent host-language input. Dashed lines represent control flow, solid ones represent data flow. We can use this graph to analyze interactions among only those primitives relevant for the contained staged programs (with control-flow dependent shapes).[4]

For instance, we perform an analysis on the stage graph that determines which parts will never take different shapes and can thus be permanently cached (or preprocessed). Simplified, these are the terms stemming from constructs that have no dynamic dependencies and are at least used once externally. In the graph of Figure 14 this applies to the `plus` nodes but not the `times` one.

Expression-Type Generation. Every `@Staged` construct is represented by its own class extending `Expression.Staged` that is generated on the fly before weaving.

Class-specific code for materialization contains the logic for cache lookup, environment and binder creation, and dispatch to the processing methods on the EDSL's class representation (e.g. `VecL.makeObjectClosure`). Additionally, a factory method is generated that performs checked incorporation of its arguments: This includes the unloading of expression DAGs from carrier instances, enforcing language boundaries, and in some cases value lifting and conversion.

Without this code generation we would need to provide additional information for every IR-node instantiation and use many conditionals or run-time reflection.

Weaving. The weaving step entails injecting bytecode for lifting, materialization, term construction, and carrier instantiation and loading. Here, working on the bytecode level is beneficial as it is much less restrictive than source code. Yet, it is still restrictive enough that we are forced to make adjustments: Suppose we have determined that a value, originally **`double`**-typed, will have been lifted at a certain program point. Then all instructions working on that value need to be changed into ones suitable for reference types. Similar changes (e.g. value lifting before control-flow merges) have to be performed all over to satisfy the JVM's bytecode verifier.

Although the implementation is cumbersome, we believe the details and edge cases of the weaving step are not particularly interesting here and beyond the scope of this paper.

Run-Time Support. Our framework's run-time support consists mainly of static methods in the hidden `Dispatcher` class, which is made public only at the start of the Java agent.

Carrier subclasses are modified in such a way that only our framework can construct "empty" instances to carry abstract syntax. We ensure this by adding a new constructor accepting a special parameter type generated at load time. Carriers contain a `payload` field of type `Expression` that is accessed using hidden methods.

Permanent caching works by using static (closure-holder) fields on the declaring class of a woven method. Their contents are provided to corresponding IR-constructor calls. For default, non-permanent caching we use a size-limiting implementation provided by Guava [2].

4. Discussion and Evaluation

The previous sections focused mostly on operational explanations. Let us now turn to assessing our EDSL-abstraction framework.

4.1 Pitfall Avoidance

Using `@Stage` to implement EDSLs alleviates or even eliminates the issues mentioned in Section 2. Since IR construction is governed by annotation and code transformations at load time, non-compositional staging is avoided for `@Staged` EDSL tokens. Recall the example given in Figure 3. With our system users could

[3] It serves a similar purpose as Scala-Virtualized's `SourceContext` [25].

[4] The actual graph contains further detailed information on the nodes than this simplified depiction.

rely on `aPlusB.toVec()` in both line 3 and line 5 to have the same shape. Furthermore, there is no workflow overlap since token-method behavior cannot be customized. Likewise, language boundaries are clear and enforced as described in Sections 3.1 and 3.2.

Static-context information is retained by our constant-analysis phase as well as by the optional collection of `StaticInfo`. To illustrate the latter, consider Figure 15 showing the factorial function written in a simple EDSL we named Mini.

```
1 static int factorial(int x) {
2   IntV n = intVar("n");
3   IntV a = intVar("a");
4   return   intAssign(n, intLit(x))
5     .then( intAssign(a, intLit(1)) )
6     .then( whileDo(leq(intLit(1), n),
7              intAssign(a, mul(a, n))
8       .then( intAssign(n, add(n, intLit(-1))) ))
9     ).intRun(a);
10 }
```

Figure 15. Mini factorial

Not shown here is that the called methods are `@Staged` and except for `intRun` their return types extend `GlobalCarrier`. The `intVar` construct creates a new variable identifier which can be used within the EDSL program. Since our framework retains the original bindings (and accounts for it in shape-isomorphism checks), Mini's compiler can use object identities for variable identifiers. Calling `intRun` with such an identifier materializes the given sequence and returns the final value of the variable.

For debugging purposes `then` is set to collect static information of type `ORIGIN` (see Section 3.6), which is used by Mini's compiler: For instance, if line 5 were to be commented out, users would be provided with an error message pinpointing that at line 7 the read variable (`a`) is unassigned.

Our framework also uses static context to avoid redundant processing. As mentioned earlier, static EDSL programs are detected and given special permanent-caching treatment. For the dynamic case redundant processing is avoided by the caching described in Section 3.3, at least in theory. We will shortly see that the overhead of our abstraction may outweigh this benefit.

One could argue that we cannot avoid the pitfalls entirely. The culprit is, by design, found in global carrying. Although on one hand methods like `times2` in Figure 11 enable code reuse and may act as auxiliary EDSL constructs, their opaque internals yet again open the door to staging-related pitfalls. Still, our abstraction governs how the core `@Staged` EDSL tokens can be combined. Hence, users can easily determine for which parts guarantees apply and for which parts they do not.

An open question is what could be done to improve this situation. Possibly, an annotation that checks certain behaviors of auxiliary methods could help. Another approach would be to add an annotation (e.g. `@Rewrite`) for rewriting or expanding while staging. While this again reminds us of the opaque-workflow pitfall, at least it would properly document and restrict such behavior.

4.2 The Cost of Abstraction

To estimate our framework's overhead at run time, we have implemented several versions of the vector EDSL: A non-staged, shallow embedding (S), an `@Stage`-annotated version (see Figure 8) with both the trivial compiler (A_N) of Figure 10, and an optimizing one (A_O). We also manually implemented non-optimizing (M_N) as well as optimizing (M_O) deeply embedded versions. The optimizations are as follows: Nested and repeated `plus` operations are collected and performed using an accumulator vector, and repeated `times` operations are merged into a single one.

Our benchmarks were run on the code of Figure 9 and on an adapted version for the handcrafted versions (M_N, M_O), called with `example(i % 20, a, b, c)` with `i` incrementing from 0 to 100,000 - 1. The variables `a`, `b`, and `c` contain randomly initialized vectors of size 100, 1,000, or 10,000. For each implementation we measured the execution time 10 times.[5]

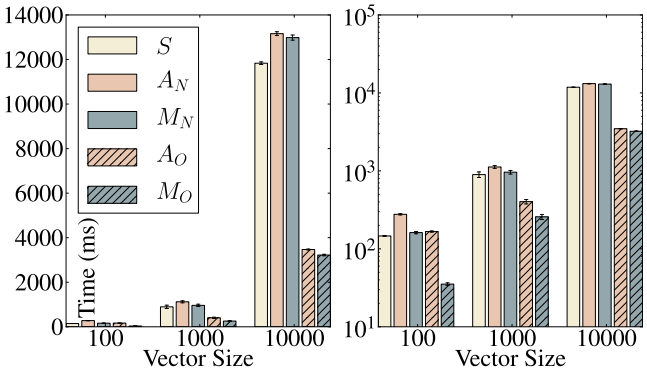

Figure 16. Benchmark results (lin. and log. scale)

Figure 16 shows the averaged results of our benchmarks for the different vector sizes. In terms of performance, using our `@Stage`-based implementations (A_N, A_O) becomes only worthwhile for vectors with more than 1,000 elements. Below that the performance is abysmal and even above the handcrafted deep embedding (M_N, M_O) yields slightly better results. The overhead mostly stems from environment accesses and isomorphism-checked cache lookups.

It is not surprising that our abstraction comes at a cost. However, it also comes with a potential for improvement unavailable otherwise: Without changing the interface it is possible to change and optimize our back end. For instance, one could consider early processing of permanently cached terms at load time [30] or before [22]. Our indirection mechanisms would allow for this.

4.3 Future Language Integration

By using annotations and load-time metaprogramming we have achieved a pragmatic solution. Our prototype's usage is rather simple and becomes a mere extension of the general EDSL idea. After all, we inherit large parts of the host-language infrastructure [19]: The look-and-feel is unaffected and the compiler, type system, IDEs, and virtual machine remain effective as is.

We believe it is worthwhile to investigate tighter (host) integration to solidify language embedding as a first-class feature. For instance, a simple, noninvasive idea would be the creation of an IDE plugin that highlights staging constructs and materialization points using EDSL-dependent color coding. Also, an early checking and warning mechanism, e.g. for language visibilities, would be desirable. Here again, an IDE plugin could assist. Alternatively, an annotation preprocessor or plugin checker [4] could do the trick.

Native support is a long-term goal for when the idea has matured and garnered acceptance. The annotations could be replaced by keywords and language classes could receive special treatment. Support by the virtual machine sounds promising to avoid the mentioned cost of abstraction and improve general performance, e.g. by integrating the feature into JIT compilation.

5. Related Work

Combining various domain knowledge within a single host language is not a novel idea. In fact, many APIs or libraries follow protocols that resemble languages. However internalizing a domain-specific language means focusing strictly on language-like qualities as in the type of embedding for which our framework is designed.

[5] Warmed up (JRE 8, 3 GHz Intel Core i7, 8 GB RAM).

5.1 Custom DSL Syntax

The idea of providing language support for DSLs has been investigated before. First and foremost there is the Python-inspired Converge [35]. In this language DSLs or sub-languages can be defined both in terms of syntax and semantics and are used in explicitly language-delimited blocks. The Jakarta Tool Suite [6] is another case of language extension for DSL development.

Examples of statically typed languages with a sole focus on custom syntax are Wyvern [24] and ProteaJ [21]. Both use type-associated syntax or operators to integrate sub-languages into their respective host languages. These approaches deliberately argue for a departure from the host language's look-and-feel and attempt to solve nontrivial parsing issues. Custom processing or optimization is a secondary concern and currently only addressed by Converge. For better or worse, dynamic composition of DSL fragments or its pitfalls is not specifically addressed.

Note that these custom-syntax systems come with their own dedicated tool chain that requires implementation and maintenance.

5.2 Language Piggybacking

It is possible to avoid the issues of a custom language and syntax by piggybacking on the features of an existing programing language. As described in Section 2, this enables the development of DSLs that follow the host language's look-and-feel. The oldest form of this technique is accomplished with syntactic macros in Lisp [17] or, as modernly revived, in Scala or Template Haskell [31].

Hudak [19] described a style of language embedding that specifically stresses the inheritance of host-language infrastructure and pureness, i.e. forgoing meta-level preprocessing. The idea is very general and leaves us with the choice of using either shallow or deep embedding [16, 32]. The latter adds a new dimension to the reasoning about embedded-language programs: They do not need to just sit there as static snippets but may be actively and dynamically constructed using host-language idioms.

One framework that has perfected deep embedding is Lightweight Modular Staging (LMS) [27], which is closely-related to the *tagless final* approach by Carette et al. [8]. LMS was motivated by bringing the ideas of *multi-staged programming* (MSP) languages [33, 36] to Scala as a library. It heavily uses a combination of Scala-specific features such as implicit conversion, type inference, and traits. It is modular in the sense that it allows mixing together several EDSL-component traits to create a new EDSL. Usage of these EDSLs occurs in a trait-inheriting program with subsequent interpretation or compilation.

As a powerful framework for constructing and prototyping expressive DSLs, LMS cannot really afford to be overly restrictive. For the most part, IR nodes as well as EDSL-construct behaviors are defined manually (e.g. on-the fly rewriting on author-defined data). Despite or maybe because of its power, as hinted at by Jovanovic et al. [22], direct use of LMS-style deep embedding might not be the best choice for realizing rich library interfaces.

5.3 Hybrid Approaches

Hybrid approaches combine DSL embedding with meta-level support. For instance, while LMS can stand on its own, it is considerably enriched by Scala-Virtualized [25, 26], a custom branch of the Scala compiler. It explores various general-purpose tweaks for improving Scala as a meta language for DSL hosting. The main idea is to allow extensive customization of host-language behavior (or language virtualization) way beyond simple operator overloading, e.g. allowing for the semantic redefinition of inbuilt control structures and even variable assignment. Furthermore it adds `SourceContext`, an implicit parameter that allows for better error reporting, thus addressing an important aspect of retaining static information (see Section 2.4). However, this is not used further, e.g. for constant detection, caching, or preprocessing.

An approach that does go further is Yin-Yang [22]. It enables the generation of an LMS-based deep embedding from a shallow interface description and macros for its usage. These macros conceal the leaky-abstraction deep embedding: At compile time they transform shallow method calls in EDSL blocks on-the-fly into counterparts of the deep embedding. EDSL authors may refine the generated deep embedding in an elegant, type-safe fashion. To customize the semantics of these EDSLs it is necessary to customize the generated source code. This may entail manually adjusting staging-time behavior (like with LMS) and leaves the door slightly open to staging-related pitfalls. EDSL code is written in (macro-based) DSL-delimited blocks. This mostly avoids fuzzy language boundaries by excluding foreign-language constructs altogether.

There also exist projects like CodeBoost [5] or Broadway [18] closely related to EDSLs. Their goal is not to extract and provide programs of an explicit EDSL as a whole. Instead they enable domain-specific optimizations by defining (rewrite) rules.

Most existing approaches rely on preprocessing at or before compile time which may limit separate compilation as well as maintainability of application code that uses EDSLs. Project Lancet [28], a custom JVM implementation, has a feature called *JIT macros*. This allows changing the semantics of expressions and thus domain-specific optimizations at JIT-compile time. However, these JIT macros are enabled and configured dynamically at run time, which makes it hard to guarantee, document, and communicate behaviors uniformly to EDSL users.

Our present framework also constitutes a hybrid approach and is loosely based on our previous work on the *implicit staging* of EDSL expressions [30], which also suggests to explicate EDSLs. Conceptually, implicit staging proposes the static extraction and reification of EDSL code, custom processing, and reintegration of the residue into user programs. The concrete load-time implicit-staging prototype only achieves expression-local handling. It successfully delimits EDSL code from host-language code and avoids fuzzy language boundaries. Yet, the EDSL-token filtering is rather opaquely defined and fails to reliably document language association.

Lack of documentation contributes to the pitfalls mentioned in Section 2, even more so when leaving the expression-local as well as static-staging realm. Especially when staging is hidden, predictability is key. We have come to believe that addressing this (as we did in the current work) is a necessary step to making hybrid language-piggybacking approaches digestible.

6. Conclusion

Major pitfalls of designing and using staged EDSLs can be overcome by using a dedicated abstraction mechanism. However, providing this feature is not without its own shortcomings and challenges. Not only is it associated with overhead it also requires careful consideration of how it fits into the host language as a whole.

Proposing language embedding as an almost first-class feature might seem at odds with the idea of DSL embedding. However, hybrid approaches have been proposed before. Furthermore, many modern languages are already taking EDSL development and usage into account in their design, e.g. in terms of syntactic flexibility.

The alternative to a specialized approach would be to devise more general host-language extensions. However, it is yet unclear how EDSL-specific concerns, including pitfall avoidance, would be comprehensibly addressed in general without complicating the host language. Exploring this is a noble goal but arguably a much harder task that would surely benefit from the existence of a specialized one for inspiration. We have merely provided an initial design and pragmatic implementation that can be used as a starting point for further detailed exploration and improvement.

References

[1] `http://www.jooq.org/` (2015-06-06).

[2] `https://github.com/google/guava` (2015-06-09).

[3] `http://javassist.org/` (2015-06-09).

[4] `http://checkerframework.org/` (2015-03-19).

[5] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs. In D. Binkley and P. Tonella, editors, *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75, Amsterdam, The Netherlands, Sept. 2003. IEEE Computer Society Press.

[6] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 143–153, Washington, DC, USA, 1998. IEEE Computer Society.

[7] J. Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[8] J. Carette, O. Kiselyov, and C.-c. Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.*, 19(5):509–543, Sept. 2009.

[9] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Berlin Heidelberg, 2000.

[10] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin Heidelberg, 2004.

[11] C. Elliott, S. Finne, and O. De Moor. Compiling Embedded Languages. *J. Funct. Program.*, 13(3):455–481, May 2003.

[12] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[13] S. Freeman and N. Pryce. Evolving an Embedded Domain-specific Language in Java. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 855–865, New York, NY, USA, 2006. ACM.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] P. G. Giarrusso, K. Ostermann, M. Eichberg, R. Mitschke, T. Rendel, and C. Kästner. Reify Your Collection Queries for Modularity and Speed! In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 1–12, New York, NY, USA, 2013. ACM.

[16] J. Gibbons and N. Wu. Folding Domain-specific Languages: Deep and Shallow Embeddings. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347, New York, NY, USA, 2014. ACM.

[17] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[18] S. Guyer and C. Lin. Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries. *Proceedings of the IEEE*, 93(2):342–357, Feb. 2005.

[19] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–142, Washington, DC, USA, 1998. IEEE Computer Society.

[20] G. Hutton. *Programming in Haskell*. Cambridge University Press, Jan. 2007.

[21] K. Ichikawa and S. Chiba. Composable User-defined Operators That Can Express User-defined Literals. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 13–24, New York, NY, USA, 2014. ACM.

[22] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 73–82, New York, NY, USA, 2014. ACM.

[23] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[24] C. Omar, D. Kurilova, L. Nistor, B. Chung, A. Potanin, and J. Aldrich. Safely Composable Type-Specific Languages. In R. Jones, editor, *ECOOP 2014 - Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 105–130. Springer Berlin Heidelberg, 2014.

[25] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation*, 25(1):165–207, 2012.

[26] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 238–261, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[27] T. Rompf and M. Odersky. 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, pages 127–136, New York, NY, USA, 2010. ACM.

[28] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical Precision JIT Compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 41–52, New York, NY, USA, 2014. ACM.

[29] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-Blocks for Performance Oriented DSLs. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, pages 93–117, 2011.

[30] M. Scherr and S. Chiba. Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 385–410. Springer Berlin Heidelberg, 2014.

[31] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.

[32] J. Svenningsson and E. Axelsson. Combining Deep and Shallow Embedding for EDSL. In H.-W. Loidl and R. Peña, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 21–36. Springer Berlin Heidelberg, 2013.

[33] W. Taha and T. Sheard. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, Oct. 2000.

[34] W. M. Taha. *Multistage Programming: Its Theory and Applications*. PhD thesis, 1999. AAI9949870.

[35] L. Tratt. Domain Specific Language Implementation via Compile-Time Meta-Programming. *TOPLAS*, 30(6):1–40, 2008.

[36] E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java Multi-stage Programming Using Weak Separability. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 400–411, New York, NY, USA, 2010. ACM.

[37] H. Xu. EriLex: An Embedded Domain Specific Language Generator. In J. Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 192–212. Springer Berlin Heidelberg, 2010.