THE UNIVERSITY OF TOKYO
Graduate School of Information Science and Technology
Department of Creative Informatics

博士論文

Pragmatic Extensions for Language Embedding Using Load-Time Metaprogramming

（言語埋込みのためのロード時メタプログラミングを用いた実用的拡張）

Doctoral Dissertation of:
Maximilian Pascal Scherr
シェア　マクシミリアン　パスカル

Academic Advisor:
Shigeru Chiba
千葉　滋

June 2015

Acknowledgments

This thesis is the result of my three years of research under the supervision of Prof. Shigeru Chiba in the Department of Creative Informatics. I am very lucky that a fair mixture of struggle and fortunate coincidences lead me to engage in postgraduate studies and research at the University of Tokyo.

I am greatly indebted to my advisor Prof. Chiba for welcoming me into his research group as a doctoral student and supporting me in my research endeavors. On countless occasions throughout my time under his supervision he offered invaluable advice and guidance that was crucial to the completion of this thesis.

I would also like to express my gratitude to the members of my thesis committee: Prof. Kenji Yamanishi, Prof. Kei Hiraki, Prof. Shinichi Honiden, Prof. Hiroshi Esaki, Prof. Toshiya Hachisuka, and Prof. Shigeru Chiba. They provided me with valuable insights and comments on how to improve and clarify my arguments and presentation.

Of course, my gratitude also extends to all members of the Core Software Group, in particular Prof. Yoshiki Sato, Fuminobu Takeyama, YungYu Zhuang, Kazuhiro Ichikawa, Thanh-Chung Dao, Hiroshi Yamaguchi, Masayuki Ioki, and Ryo Fukumuro.

Thanks also to my family and friends, whose assistance and company have been a great source of energy, joy, and comfort throughout all of my studies and my work on this thesis.

Finally, without the scholarship provided by Japan's Ministry of Education, Culture, Sports, Science and Technology (MEXT) I would not have been able to come to this country and fulfill my goals. I am deeply thankful for it.

MAXIMILIAN PASCAL SCHERR
Tokyo, Japan
June 2015

Abstract

This dissertation details my research on the improvement of techniques for embedding *domain-specific languages* (DSLs) in a general-purpose host language. Standalone, dedicated DSLs provide a high-level abstraction to problem solving. Their *embedding* by using existing host-language constructs and language features not only reduces the initial implementation effort but also opens avenues to enrich traditional programming-library interfaces.

Historically this idea evolved from *compile-time metaprogramming* techniques such as syntactic macros. In modern high-level languages one can find a tendency of avoiding such so-called *impure* techniques. Instead, language constructs that have a fixed run-time behavior are preferred and often sufficient. This pure embedding is arguably easier for authors as well as (and maybe more importantly) users of *embedded DSLs* (EDSLs) due to the predictable, uniform behavior of the employed constructs or abstraction mechanisms.

Still, existing manual implementation approaches suffer from tradeoffs in terms of *reliability*, *performance*, and *usability*. The major cause for this is found in the low-level emulation of high-level embedded-language aspects. Ultimately, EDSLs are *non-citizens* in their general-purpose host language, which leads to missed opportunities for eliminating the aforementioned tradeoffs.

My proposal is to raise the status of EDSLs to almost first-class citizens by means of meta-level, semi-linguistic support. It lies in the nature of doing so to tamper with the workings of the host language, essentially extending it. This is prone to lead to heavyweight designs unlikely to appeal to mainstream users. Also, many of the strengths of language embedding may be weakened when interfering with the host-language front end. As an alternative, *load-time metaprogramming* offers a good compromise for realizing a pragmatic form of language extension in our context.

The initial milestone is to turn EDSLs into explicit entities with ownership of associated host-language constructs such as methods. I address this with the so-called *implicit-staging* approach, which is rather abstract in nature. Based on a prototype framework and closer investigation I conclude that further restrictions and documentation mechanisms are necessary to render EDSLs more reliable and useful. This is explored with a concrete framework that, in addition to load-time metaprogramming, relies on Java's annotation feature to enable control and communication of *staged-EDSL* behavior.

Contents

List of Figures

List of Code Listings

List of Tables

Chapter 1

Introduction

The defining goal of computer languages has always been to enable the communication of non-tangible, thought-up ideas, plans, problems and their solutions between man and machine. Thanks to extensive research and the desire to lower the barrier of entry into the field of computer science, the level and quality by which this communication may be conducted has increased significantly since the era of punch-card programming.

However, despite the existence of high-level programming languages there has not yet been found a single, universal programming language, despite efforts to do so. One may argue that any significantly large and feature-rich language can fulfill that role and this is true when one considers theoretical properties such as computability. However, when it comes to suitability to express and solve certain classes of problems this claim loses its weight. Furthermore, modern software systems have grown so large that it is not possible to simply sweep under the rug concerns of modularity, code reuse, and genericity.

The very fact that various programming paradigms are competing with each other, have emerged, disappeared, and reemerged, appears to be a strong indication that a single general-purpose language is unlikely to become the single best tool for every task, for every user, and for every type of environment (e.g. sequential vs. concurrent). The same seems to be the case for data-description languages and computation and storage paradigms in general. If there is such a thing as a universal solution to everything we do not seem to be close to finding it.

While it still makes sense to consider and investigate the general (-purpose) case, i.e. to look for novel features that cover as many use cases and concerns as possible, for the time being it seems just as worthwhile to focus on very limited, special problem areas and instances and simplify their solution. This gives rise to the concept of *domain-specific languages* (DSLs), i.e. computer languages that are designed with a specific application domain in mind.

The concept is neither novel nor surprising. So-called "little languages" [20] have been around for a long time and most notably find use in Unix shell programming. Their success can be attributes precisely to their domain specialization. On the other hand many modern *general-purpose languages* and runtime systems have started out

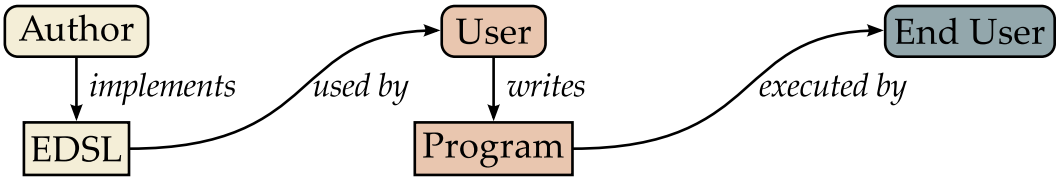


FIGURE 1.1: EDSL implementation and usage roles

with certain domains in mind: ML [110] was designed to support theorem proving, Prolog's [114] initial application domain was natural language programming, and Lisp [75, 97] had its mark set on artificial-intelligence applications. To some extent even more modern systems like Smalltalk [47] and Java [49] started out with a specific focus, though with general-purpose applications in mind. The former was envisioned for use in education, the latter for targeting embedded systems.

Notable examples among DSLs in modern use are Graphviz's dot [1], POV-Ray's SDL [2], SQL [28], and the R language [77]. Interestingly, due to various feature extensions the latter two are already reaching the arguably fuzzy boundary between domain specialization and general-purpose programming. On the other hand, we find general-purpose languages increasingly allowing for more domain specialization with libraries in combination with clever uses of language features. This philosophy has been explored since the days of Lisp and its dialects [52, 72]. A more modern example for this case is Python [3] with its extensive libraries for numerical and symbolic computation for scientific applications [4].

Looking at programming libraries from a DSL perspective leads to the concept of *embedded domain-specific languages* (EDSLs[1]). One may try to discover and discuss language-like properties in existing libraries but arguably a more intriguing endeavor is to contemplate the novel design or redesign of libraries as EDSLs, i.e. with a language-like nature in mind. Such library-interface designs lead not only to improved ease-of-use and readability (compared with traditional libraries) but can offer better reliability (e.g. illegal states may be avoided by proper EDSL design) and performance benefits through domain-specific optimizations.

The emergence of EDSLs and their popular dissemination in recent years (e.g. as found in the books by Fowler [38] and Ghosh [44]) has lead to a sort of democratization of programming-language design. After all, developing a language with its own parser and compiler or interpreter just for the sake of solving a small problem is a much more daunting task than extending a language's feature in the form of a library. The latter has been common practice and received attention in the form of linguistic support (e.g. module systems, interfaces, encapsulation, information hiding, etc.) for a long time. It is thus only logical to consider language embedding as a means both to simplify DSL implementation as well as to improve library interfaces.

Treating EDSLs similar to traditional libraries helps entice programmers to engage in lightweight language design and usage. While this is a welcome development in the field of software engineering and programming languages it comes with a flaw: EDSLs ought to provide a high-level abstraction, yet the foundation for

[1] Or sometimes abbreviated as DSELs [59].

their implementation is not supporting this. Concretely, what looks like a language is not bound to actually behave like a language or offer the same advantages. This is only natural and inherent to the idea since EDSLs or the concept of a language-like library are often a matter of (subjective) perspective.

This thesis is about improving the foundation of EDSL development by means of targeted, pragmatic host-language extensions for EDSL abstraction to help their authors and users (see Figure 1.1) alike.

1.1 Motivating Problems

Language embedding entails the reuse of a general-purpose language's constructs and abstraction mechanisms. Depending on the choice of host language the author of an EDSL may choose between a small selection of various implementation and design approaches. The easiest and most widely available approach is to directly match the syntax elements, i.e. the tokens of the EDSL, with the semantics of the host language. This requires no special language support whatsoever and is virtually indistinguishable from a traditional library when ignoring that snippets of its usage code may look like snippets of a small language.

LISTING 1.1: Haskell's **`map`** and **`filter`** functions for list processing

```
1 map :: (a -> b) -> [a] -> [b]
2 map f []       = []
3 map f (x : xs) = f x : map f xs
4
5 filter :: (a -> Bool) -> [a] -> [a]
6 filter p []                   = []
7 filter p (x : xs) | p x       = x : filter p xs
8                   | otherwise = filter p xs
```

A good examples for such an embedding is the type of list processing commonly found in functional programming languages [62, Chapter 7] using immutable lists and higher-order functions. Listing 1.1 shows the two central list processing functions in the lazy and pure (here referring to the absence or restriction of side effects) functional programming language Haskell [60].

Modern Java also supports higher-order functions in the form of lambda expressions for so-called functional interfaces. This allows me to introduce an illustrating example in Java. The use case is a database of direct (travel) connections between locations and information on pricing and employed vessels. The connections are stored in an immutable list (of type `ImmutableList` from the Guava library [5]).

The method in Listing 1.2 uses my own definition of the `map` and `filter` functions (highlighted by underlining) to query the database for connections with one little twist: The type of vessel (e.g. ship, train, plane) is only considered if there are more than ten results fulfilling the other criteria. The result is a list of strings containing only part of the information per connection.

While this is easy to use, readable, and concise, it will yield subpar performance due to the creation of many intermediate results. In addition, admittedly the EDSL nature is not immediately obvious here. However, it does appear when considering

LISTING 1.2: Connection query (implicit operation chains)

```
import static ….ImmList.map;
import static ….ImmList.filter;
…
public ImmutableList<String> query(String from, String to,
                                   double maxPrice,
                                   Vessel.Type vesselType) {
  ImmutableList<Connection> l = this.connections;
  l = filter(l, c -> c.getFrom().equals(from));
  l = filter(l, c -> c.getTo().equals(to));
  l = filter(l, c -> c.getVessel().getCapacity() > c.getBooked());
  l = filter(l, c -> c.getPrice() <= maxPrice);

  int count = l.size();
  if (count > 10) {
    l = filter(l, c -> c.getVessel().getType() == vesselType);
  }

  return map(l, c -> "(" + c.getId() + ", "
                         + c.getVessel().getType() + ", "
                         + c.getPrice() + "$)");
}
```

what happens in this query when limiting our view to the behavior of the two EDSL tokens: Chains of several `filter` and `map` operations.

This leads us to a more indirect style of language embedding where this chaining of operations, or in general the composition of operations, is made explicit. The syntax of the EDSL is not directly linked to its semantics anymore and the actual computation is delayed until the constructed chain of operations is evaluated. Consider the new query method in Listing 1.3 which uses two auxiliary tokens: `of`, which lifts input lists to values of type `ImmListE`, and `eval`, which triggers evaluation. Note that `filter` and `map` no longer directly perform filtering or

LISTING 1.3: Explicit, reified operation chains

```
public ImmutableList<String> queryReified(…) {
  ImmListE<Connection> lE = ImmListE.of(this.connections)
    .filter(c -> c.getFrom().equals(from))
    .filter(c -> c.getTo().equals(to))
    .filter(c -> c.getVessel().getCapacity() > c.getBooked())
    .filter(c -> c.getPrice() <= maxPrice);

  ImmutableList<Connection> l = lE.eval();
  lE = ImmListE.of(l);
  int count = l.size();
  if (count > 10) {
    lE = lE.filter(c -> c.getVessel().getType() == vesselType);
  }

  return lE.map(c -> …).eval();
}
```

LISTING 1.4: Manual query implementaion

```
public ImmutableList<String> queryManual(⋯) {
  ArrayList<Connection> temp = new ArrayList<>();
  for (Connection c : this.connections) {
    if (   c.getFrom().equals(from)
        && c.getTo().equals(to)
        && c.getVessel().getCapacity() > c.getBooked()
        && c.getPrice() <= maxPrice) {
      temp.add(c);
    }
  }

  ImmutableList.Builder<String> resBuilder = ImmutableList.builder();
  int count = temp.size();
  if (count > 10) {
    for (Connection c : temp) {
      if (c.getVessel().getType() == vesselType) {
        resBuilder.add(⋯);
      }
    }
  } else {
    for (Connection c : temp) {
      resBuilder.add(⋯);
    }
  }

  return resBuilder.build();
}
```

mapping. Instead, they are constructing the explicit filtering and mapping chain.

The *reification* (as in turning something abstract into something concrete) of these little EDSL subprograms (at run time) as data enables their analysis, domain-specific optimizations, and run-time code generation to speed up performance tremendously, comparable to a manually optimized implementation of this query as shown in Listing 1.4. On the other hand, the implementation of this optimizing EDSL is not as straightforward anymore and can easily lead to issues that are hard to debug on both the EDSL author's as well as the user's side. For instance, if the generated code is not cached there will be processing overhead every time evaluation is triggered or just-in-time compilation might not kick in at all, resulting in subpar performance yet again.

Furthermore, for a user it is unclear at first sight what actually happens in the EDSL-token methods. For instance, do they only contribute to the construction of the chain or do they already perform partial optimizations? Also, is it possible to share and reuse parts of the chain to build a bigger one without problems? These questions can be addressed by diligent design and documentation. However, it would be better if these common EDSL concerns could be handled in a common, uniform way both EDSL authors and users alike can rely on.

In order to overcome the performance limitations of the initial direct language-embedding style we not only have to sacrifice reliability of the used abstraction (i.e. procedural abstraction versus possibly non-uniform program reification) we also

end up negatively affecting usability. Arguably, the EDSL in Listing 1.2 is easier to use than the one in Listing 1.3 for the following reasons:

- It does not require the introduction of an additional type (`ImmListE`).
- It does not require auxiliary tokens.
- It does not require considering program reification.
- It does not require thinking about evaluation triggering.

There are ways out of this dilemma by straying from the described *pure-embedding* techniques [59], i.e. by using various methods of user-program preprocessing. However, this often entails heavyweight, ad-hoc, makeshift solutions or inbuilt metaprogramming support by the host language (and its compiler) such as syntactic macros. But while these address certain parts of the usability concerns such as direct usage, they sacrifice other parts. It is my contention that most existing approaches are either too powerful, i.e. generally not tailored to EDSLs, and thus do not provide reliable abstractions for language embedding, or suffer from their compile-time or pre-compile-time nature.

1.2 Position and Contributions

In this thesis I address the concerns of reliability, performance, and usability encountered in the design and implementation of EDSLs. My goal is to alleviate the tradeoff effect between these concerns, where an ideal solution would allow meeting all these concerns while minimizing negative impacts.

Concretely, I propose the evolution of semi-linguistic extensions for the explicit, high-level support of language embedding in a general-purpose language with focus on controlled EDSL-program reification. By itself this idea appears to be at odds with the very notion of EDSLs as entities that merely inherit host-language features. However, in the past this argument has not stopped general-purpose languages and research projects [89] to add features that while general in appearance find use mostly in language embedding. For instance, *generalized algebraic data types* [29, 116] (GADTs), as found in OCaml [6] and Haskell, most commonly find application in encoding *higher-order abstract syntax* [85] (HOAS) for use in EDSLs [46, 102]. Embedded DSLs have also been considered during the design of the Scala language [80] with liberal rules on method naming (e.g. akin to operator overloading) and the optionality of spelling out syntax like $\cdots . \cdots$ or `(`$\cdots$`)` in many cases. However, the intention of the approach presented in this thesis is to go one step further and consider EDSLs as almost first-class entities.

To further justify my endeavor within the realm of EDSLs, I propose to utilize load-time metaprogramming [30] techniques. This type of metaprogramming is commonly explored and pursued in the context of Java, low-level bytecode instrumentation or transformation, and the Java Virtual Machine (JVM). So do I for the target of my extensions. However, note that the concepts would also be applicable to other environments with a similar design. I posit that load-time metaprogramming makes my approach pragmatic in the following sense: Just

like embedding approaches that do not rely on user-program preprocessing the compilation process remains unaffected. Here, i.e. on the syntax level, the nature of EDSLs as inheriting host-language features is preserved. Meta-level processing of user-programs and in extension that of potentially embedded-DSL subprograms is delayed until code loading (corresponding to Java-class loading) occurs as part of the eventual application execution.

The fact that the proposed extensions are augmentations to the host language and not fully integrated language features (i.e. not a new language yet) ultimately means that my undertaking cannot be considered pure. However, their evolution and development is approached by looking through the lens of pure DSL embedding. The motivation for this is that pure embedding techniques at the very least provide strong behavioral guarantees for the basic building blocks (i.e. lingusitic abstractions) such as function calls, data access, and type-based syntax constraints.

The individual contributions of this thesis do not necessarily correspond one-to-one to chapter names. They are described in the following:

Exploring Concerns and Pitfalls. I identify concerns and deficiencies when designing and implementing EDSLs for the role of rich library interfaces. Concretely, while conceptually EDSLs are often proposed, explored, and presented as high-level abstractions, they may easily fail to live up to their expectations due to potentially leaky concrete implementations. The cause for this is found in the ad-hoc, manual nature of most existing language-embedding approaches. In other words, the high-level concept of a language within another language is manually emulated with lower-level linguistic abstractions.

Explicit EDSL Entities for Automation. To address and eliminate the problems of manual approaches, I propose the consideration of meta-level mechanisms for guiding the implementation and usage of EDSLs, in particular the reification of EDSL programs so as to avoid problems such as unclear boundaries or overlap between various embedded DSLs. The first step towards this is to turn EDSLs themselves into concrete entities with a declaration of token membership and thus delimitation, as well as the encapsulation of their programs' processing phase.

Implicit Staging. I explore the first step toward automation with the so-called *implicit-staging* approach. It was originally motivated by the goal to enable EDSL-program optimization without having to resort to run-time reification, which is seen as an impairment on (seamless) usability and performance. Implicit staging thus attempts to approach what is called *deep embedding* (see Section 2.2.7) while having programs look like ones of a so-called *shallow embedding* (see Section 2.2.6).

In a nutshell, it separates and extracts domain-specific code from general-purpose code in user programs and allows its custom processing by EDSL authors. This requires program analysis before a user program is executed.

EDSL-Program Reification at Load-Time. Most existing DSL-embedding implementations that require static preprocessing choose compile time to do so. This affects the maintainability of programs using such EDSLs: If a bug is detected in

the EDSL implementation every program that uses the EDSL has to be recompiled using an updated, bug-fixed version. Furthermore, some information that could be worthwhile for reification and processing is not yet available at compile time. For implicit staging, load time is a viable alternative to compile time. Hence, as outlined earlier in this section, I concretely explore implicit staging as a pragmatic extension for Java using load-time metaprogramming. The developed prototype framework is tested, evaluated, and compared with other approaches.

Shortcomings of Implicit Staging. The prototype is a good first step toward improving DSL embedding through automation. However, it has two shortcomings. Firstly, the interface for defining explicit EDSL entities is open to abuse and may cause bad surprises for users of an EDSL. Secondly, the purely static nature of implicit staging comes both with advantages as well as drawbacks: On one hand run-time-reification overhead is eliminated, but on the other hand the situation (i.e. ad-hoc implementations) is not improved for EDSLs desiring to provide dynamic program generation for the sake of adaptiveness or run-time configuration.

Almost First-Class Embedding. The lessons learned from the implicit-staging prototype directly lead to a culmination of the previous ideas in my so-called *tame-staging* framework. It is a concrete implementation for guiding the run-time reification of EDSL programs. One major improvement over the implicit-staging prototype is the mechanism for designating Java methods or fields as EDSL tokens. It uses a so-called `@Stage` Java annotation that clearly associates its targets with a statically known explicit EDSL entity represented by a Java class.

Guiding is established by automatically and transparently transforming user code so that it performs dynamic EDSL-program reification when executed. Full seamlessness is only achieved within the local scope of a method body and is called *local carrying*. However, it is still possible to use the other advantages of the framework beyond the method-local scope using so-called *global carrying*, however at the cost of sacrificing some amount of seamlessness. Among these other advantages are for instance the guarantee of predictable EDSL-program-fragment composition as well as the automatic caching and reuse of EDSL-program-processing results.

The framework is evaluated and tested using several example EDSLs and compared with other approaches. The evaluation yields good results for the concerns of reliability and usability, while some issues with performance remain. These may be addressed by more aggressively exploiting static situations in user code or improving other details of the implementation such as the caching mechanisms.

Opportunities for Native Support. Just like the implicit-staging prototype, the tame-staging framework uses load-time metaprogramming behind the scenes. This proves to be a good, pragmatic choice for extending the host language (Java) for the purpose of my research. However, it leaves some room for improvements that could be achieved by a full-fledged, native (as opposed to "almost first-class") language feature for DSL embedding. I discuss some of these opportunities as part of the treatment of the tame-staging framework.

1.3 Organization

The thesis is organized into five chapters (and three appendices). The current chapter combined with Chapter 2 forms the motivation and background-discussion part, in which the main definitions, concerns, approaches for language embedding, together with their advantages and shortcomings, and the basis of comparison are laid out. More concretely, the approaches of compile-time as well as pre-compile-time preprocessing are discussed, followed by more widely available ones that rely only on run-time behavior. Whichever is chosen, the common feature of performance-oriented EDSLs is that their programs' execution is not immediate but delayed to allow for an intermediate optimization step. The difficulty of designing user-friendly DSL embedding stems from this necessity and its realization. Detailed comparisons are presented at the end of Chapter 2 and further supplemented near the ends of the two following chapters.

Chapter 3 introduces the notion of turning EDSLs and their tokens' language membership into explicit entities. This enables an approach for performing controlled, static reification of EDSL programs and customized, author-defined processing. Subsequently, this is concretely explored in the form of a load-time code-transformation framework in Java. Evaluation and discussion indicate shortcomings in terms of uniformness and stresses the approach's limitation to static processing only. This chapter is largely based on my research published in co-authorship with Prof. Chiba at the 28th European Conference on Object-Oriented Programming (ECOOP 2014) [93].

The issues of Chapter 3's approach are addressed in Chapter 4. It takes the idea of token-to-language association to an easier-to-use, restricted interface using Java's annotation mechanism. Furthermore, the scope of EDSL-program reification and composition is extended beyond the expression level and made dynamic. One supported usage style allows for a method-local, transparent form of reification and evaluation triggering, while a second one is more explicit. Both are accomplished and enhanced by static knowledge (internal to the framework). The approach is illustrated and evaluated on several examples. This chapter as well as Section 2.3.3 of Chapter 2 are largely based on my research to be published in co-authorship with Prof. Chiba at the 2015 International Conference on Generative Programming: Concepts and Experiences (GPCE 2015) [92].

Chapter 5 summarizes the presented work and reiterates some of its limitations and potential future directions. Three appendices follow, which contain supplementary data to the experiments conducted in Chapter 3 and 4, as well as minimal documentation for using the framework introduced in Chapter 4.

CHAPTER 2

Background

To embed a domain-specific language has come to mean the focused use of general-purpose features, i.e. (linguistic) abstractions, to *emulate* a language, or at least what feels like one, as a guest within another. While this nomenclature might still be disputed[1] "embedded DSL" is the one chosen here.

Admittedly, the concept can be somewhat elusive. If we were discussing the integration of guest languages with their own syntax [63, 83], semantics [109] and specific compilation process we could perhaps more easily find satisfying answers to the question of what a guest language in this context means and what its delimitations are. However, traditionally EDSLs have decidedly not been about involving that degree of integration and delimitation. Instead, they are rather about adapting one's perspective than they are about clear cuts.

This chapter attempts to establish terminology and introduce approaches to language embedding. For the sake of further discussion some aspects are refined or have their definitions adapted, while others will have to remain somewhat underspecified. The goal is to draw enough of the big picture and focus on selected details to enable the evaluation of this thesis's motivation and results.

2.1 The Essentials

Wide use of language embedding in general emerged first in the Lisp community. Graham [52] provides good examples for how this is accomplished with little effort, including a Lisp-embedded implementation of Prolog. The reason Lisp deserves the honor as the breading ground for embedded languages is found in the language's minimalism. Its syntax, i.e. *S-expressions*, coincides directly with its abstract syntax and opens its users' minds to considering programs as data in an immediate fashion. New language features are introduced via extensions, i.e. syntactic macros, that directly build upon this idea. In this fashion a new language feature looks no different from existing features. It would not be surprising to

[1] Instead of EDSL, some prefer the term *internal DSL* as opposed to *external*, i.e. standalone, ones [38], others use both [44], and others may recognize the term but avoid its usage.

encounter Lisp programmers thinking nothing special of EDSLs because having new language features and combinations thereof just comes natural to them.

Hudak and others [35, 37, 59, 61] took the idea of embedded languages to the world of statically typed functional programming languages. The general idea is to simple use *existing* language features and abstraction mechanisms, e.g. procedural and data abstraction, to implement DSLs. It is argued that one can "inherit the infrastructure of some *other* language [and tailor] it in specific ways to the domain of interest" [59], thereby avoiding the trouble and effort of building a DSL from scratch. Furthermore, there is a stress on the *purity* of DSL embedding, e.g. one that does not rely on a "preprocessor, macro-expander, or generator" [59], but on sound (run-time) abstractions and the type system of the host language.

Of course, whereas EDSLs in Lisp would naturally fit into the already crude S-expression look of the host-language and maybe appear like a first-class language feature, in a language like Haskell having an embedded language be expressed by function calls is potentially less natural. However, it may be argued that retaining the *look-and-feel* of a core (i.e. the host) language specifically helps users approach DSLs more openly as one part of a bigger program as well as enable seamless usage and switching of several DSLs at the same time [59].

In whichever way DSLs are embedded in detail (e.g. with which degree of purity) the notion dictates that large parts of a *general-purpose* host language's syntax, its semantics, and compile-time analyses are reused to simplify both development as well as usage of the DSL. The effect of naturally enabling easy interfacing and intermixing of domain-specific and general-purpose code is also not to be underestimated.

2.1.1 Motivations for Language Embedding

The described form of language piggybacking has been embraced by various programming-language communities ranging from compiled as well as interpreted, functional as well as imperative, statically as well as dynamically typed, niche as well as mainstream languages [5, 7, 8, 38, 39, 44, 45, 101]. One can categorize the focus of research and development interest within the field of DSL embedding into two different motivations:

- ***Embedded DSLs in Lieu of Standalone DSLs.*** Instead of building new dedicated languages from scratch either by hand or supported by parser and compiler generators, a sufficiently powerful host language and supporting frameworks within it can be used to the same or even better effect [73, 87, 88, 89, 100]. In other words, the goal is to simplify prototyping and close-to-standalone DSL implementation. The resulting DSLs are used like standalone DSLs that just happen to be embedded.

- ***Rich High-Level Library Interfaces.*** Traditional programming libraries whose usages fit or can be made to fit into a language-like understanding can be equipped with a richer, high-level interface that exposes and exploits this nature [8, 19, 35, 38, 39, 45, 53, 59, 61, 67, 68]. In other words, the goal is to turn libraries into DSLs that, thanks to embedding, can remain part of the (host) language just like normal libraries.

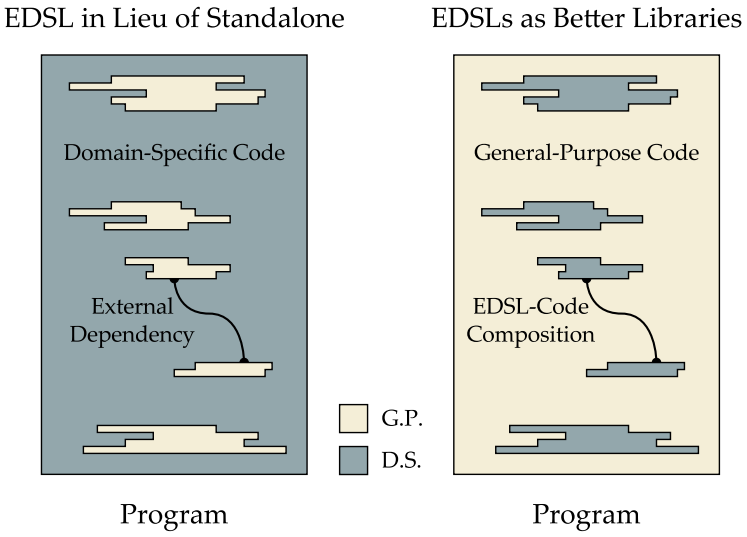


FIGURE 2.1: Duality of language-embedding motivations

As mentioned before, EDSLs are about perspective. The former motivation takes the perspective that standalone DSLs are simply (restricted) languages with their own syntax and semantics, both of which can be abstracted away by cleverly expressing both within a general-purpose language. Likewise, the latter motivation takes the perspective that programming libraries, which after all are already being specific to a domain, sometimes just lack the language aspect of a DSL, or alternatively they have it but it is not exploited well (e.g. for performance or reliability).

Figure 2.1 shows the duality between these two motivations. In the case where focus lies on realizing a single DSLs as an embedding, general-purpose code serves both as a scaffolding to make the EDSL work, as well as allowing for interfacing with the outside world. The expectation is that the amount of EDSL code dominates the amount of general-purpose code.

In the case of libraries turned into rich language-like interfaces the expectation is that the majority of code is general-purpose code with library accesses, i.e. EDSL usages, sprinkled in between, often but not necessarily involving several different EDSLs. Note that this corresponds with traditional views on library usage. With these language-like interfaces, composition of library functionality is expressed as EDSL-code (snippet) composition.

In reality the separation does not have to be as black and white as this extreme depiction makes it seem. Furthermore, the motivations are not necessarily mutually exclusive. After all, support frameworks, proposed for turning planned external DSLs into embedded ones, may certainly be useful for EDSL development in general and thus for library-interface EDSLs as well. Likewise, in principle it is possible to use EDSLs not initially developed with library-interface use in mind as such, though arguably more indirectly than if they had been designed for this purpose.

Yet, the choice of motivation (or its weight) can have an effect on EDSL design. For instance, an EDSL program intended to be run like a dedicated, standalone DSL program, i.e. one that acts as a one-time code generator or compiler, is at liberty to ignore issues such as how to handle the presence of other (non-anticipated) EDSLs'

code or how to avoid redundancies when the same EDSL snippet is executed repeatedly, as might occur with EDSLs used as library interfaces.

2.1.2 The Role of the Host Language

The choice of host language affects the ability to design and use EDSLs. As mentioned in the introduction of this section, the nature of Lisp's syntax has enabled its extensibility and openness to the concept of embedded DSLs. On the other hand its syntax (e.g. prefix operators) is not necessarily inviting to domain-expert newcomers to Lisp programming. To some extent this is inherent to all EDSLs and intentional (to maintain a common look-and-feel). Haskell and Scala have addressed this in their own ways by offering flexibility in expressing function applications as well as the usage of custom operator names.

More important than the precise expression of concrete syntax on the surface is the ability to influence and guide the syntax of EDSL programs. Unfortunately, the term "syntax" is heavily overloaded. In the context of embedded languages in this thesis, "EDSL syntax" refers to the permitted compositions of EDSL tokens and not to the concrete syntax at the host-language front end.

With dynamically typed host languages like Lisp, Python, or Ruby [9], there is hardly anything that enables the guiding of EDSL syntax in an easy fashion. It should be noted that Lisp-style macros can in fact be used to check and report syntax errors[2] in individual EDSL expressions but this requires additional consideration and manual implementation.

LISTING 2.1: Signatures as syntax (by static typing)

```
1  // Integer expressions
2  public static IntV  intVar   (String name)          { ··· }
3  public static IntE  intLit   (int       a)          { ··· }
4  public static IntE  mul      (IntE      a, IntE  b) { ··· }
5  ···
6  // Boolean expressions
7  public static BoolE eq       (IntE      a, IntE  b) { ··· }
8  public static BoolE and      (BoolE     a, BoolE b) { ··· }
9  ···
10 // Statements
11 public static Stmt  intAssign(IntV      v, IntE  e) { ··· }
12 public static Stmt  whileDo  (BoolE  test, Stmt  s) { ··· }
13 ···
```

Static typing provides an easy way to restrict the syntax of a host language's constituent guest EDSLs. Take for instance the simple Java method signatures (including return type) presented in Listing 2.1 for a small imperative EDSL. It is clear and lamented at compile time that an expression such as

```
intAssign(intLit(1), and(···))
```

is not valid syntax. This is a supporting example for Hudak's [59] call to inherit host-language infrastructure, in this case the static-typing infrastructure. So it

[2] Some macro systems even enable the implementation of statically typed self extensions [107].

makes sense to choose a host language (and not to forget an associated community) that already offers as much infrastructure as possible, including tools such as IDEs. The idea of restricting syntax has also been explored in the context of automated generation of typed as well as untyped EDSLs [117]. Note that expressing and checking domain-specific type systems as well as semantics (e.g. whether a variable *within* the EDSL was assigned before usage or not) in this fashion is much harder or impossible. *Dependently typed languages* [23, 24, 79] look like promising candidates for filling this gap. However, for the time being, in mainstream languages such checking is commonly deferred until run time.

A host language's evaluation strategy is another aspect heavily influencing EDSL design. Most commonly found is *call-by-value* or *strict*, *eager* evaluation where argument expressions of a function call are evaluated (usually from left to right) to values to be used as actual arguments within the called function. With *call-by-name*, a *non-strict* strategy, this evaluation does not occur before the call. Instead, the argument expressions are merely substituted within the body of the called function. Lazy evaluation or *call-by-need*, as employed for instance by Haskell, is similar to call-by-name where expression are also substituted but evaluated at most once.

LISTING 2.2: Trivial lazy-evaluation example in Haskell

```
1 ifThenElse :: Bool -> a -> a -> a
2 ifThenElse test trueBr falseBr = if test then trueBr else falseBr
3 ···
4 let x = ···
5     y = ···
6 in  ifThenElse (3 > x) (x + y) (x ^ 2 + y ^ 2)
```

In the example of Listing 2.2 only *one* of `(x + y)` *or* `(x ^ 2 + y ^ 2)` is evaluated depending on the result of `(3 > x)`. In strict languages like Python, C, Java, or OCaml all three argument expressions would be evaluated before the call. It is possible to simulate something akin to call-by-name strategy by requiring explicit (first-class) functions as arguments but this usually results in more verbose expressions. Scala supports call-by-name out of the box, e.g. by specifying a formal parameter as `(x: => A)` (for some type `A`) instead of `(x: A)`. Note that syntactic macro "calls" do not really fall into any of these categories as their expansion and thus semantics is entirely customizable.

A consequence of lazy evaluation on a global scale throughout the programming language is that it requires freedom from unanticipated side effects. This has advantages for language embedding as it avoids unpredictable behavior when intermixing with general-purpose code. On the other hand, it also requires general users to adapt to a specific, potentially unfamiliar style of programming. Without taking sides on which programming paradigm is best, let us merely reiterate here that it affects EDSL design and usage.

2.2 Implementation Techniques

Based on the various implementation techniques and examples already mentioned, by now the reader has hopefully developed a rough intuition for the core concepts

of EDSLs. However, it is still crucial to consider the details of the various existing approaches to gain a better understanding of the design space and the challenges involved. In the following I will first consider what Hudak [59] might have called *impure embedding* techniques and then shine more light on pure embeddings as well as somewhat orthogonal concerns.

2.2.1 Source-Code Preprocessing

If all that is inherited by a host language for EDSL development is the concrete and abstract syntax, one can implement an EDSL by writing a source-code preprocessor (or source-to-source compiler) that needs to be applied to user programs. In this case the degree of inheritance of host-language infrastructure is rather small. What separates this from an approach with entirely custom-syntax DSLs inside host-programs is that at the very least the lexical and syntactic analysis components can be reused if publicly available. Even if unavailable, it is fairly easy to generate parsers from language specifications using parser generators [84] or parser-combinator libraries [74].

The reification of entire user programs with general-purpose as well as domain-specific code portions allows for a great deal of freedom to custom-tailor the EDSL including semantic analyses and optimization strategies for its programs. However, the disadvantages are immediately evident:

- Implementation is heavyweight and hard to maintain. For instance, consider what happens when a new language feature is introduced to the host language.
- Only pre-compile-time information is available. For instance, target architecture of the final compilation cannot automatically be taken into consideration for optimizations.
- Language interactions between various preprocessor-implemented EDSLs are entirely unclear.
- The connection between the original program and the processed program is lost, hampering the debugging of the original (unprocessed) programs.
- The preprocessing and its effects are a global whole-program black box.

In effect, such preprocessor-implemented EDSLs are closer to a new general-purpose language and too custom to be considered a general abstraction pattern, let alone (linguistic-) abstraction mechanism.

2.2.2 Active Libraries

Standalone source-code preprocessors become more viable when they provide general abstraction mechanisms for EDSLs. One such example is CodeBoost [19] which enables the definition of domain-specific rewrite rules for optimizing combinations of specific operators and functions (belonging to DSLs) embedded in C++ [99]. It is reusable and the black-box issue is mostly eliminated by documenting and limiting the scope of rewriting.

Another example for such a framework is the source-to-source Broadway compiler for C [53, 69] which allows library authors to define their own static analyses (using the concept of *typestate* [98]) and optimizations based on them via conditional rewriting. It is more wide-reaching than CodeBoost (and in fact many other traditional approaches) due to the custom data-flow analysis aspect and thus its departure from syntactic-only approaches.

Both CodeBoost and the Broadway compiler essentially incorporate the idea of (and enable) so-called *active libraries* (i.e. "libraries which take an active role in generating code and interacting with programming tools" [112] as part of the compilation process) but stop short of making the realization, verbalization, or step toward the concept of libraries as EDSLs. In both frameworks focus lies more on small optimization steps than customizing EDSL subprograms as a whole. Likewise, it is important for us to keep in mind the connection between active libraries and the motivation of EDSLs as richer library abstractions or interfaces.

One example for a lightweight active-library mechanism in active use can be found in modern versions of the popular Glasgow Haskell Compiler (GHC) [54] in the form of language-integrated rewrite rules. Listing 2.3 shows an example taken directly from the official documentation [10].

LISTING 2.3: Rewrite rule for **map** fusion in Haskell (GHC)

```
1 {-# RULES
2       "map/map"    forall f g xs.  map f (map g xs) = map (f . g) xs
3   #-}
```

2.2.3 Compiler Plugins

Customizable preprocessing integrated into the compiler itself is a step up from standalone preprocessors. This integrated preprocessing is usually found in the form of a compiler-plugin feature as for instance provided by the Scala compiler.

This feature is not always explicitly given the name "compiler plugin". Some languages like Racket [11, 107] and (experimentally in) Scala [25] offer support for types of macros that have a bigger scope beyond that of only expressions. This makes them closer to compiler plugins than expression-level syntactic macros.

The big difference between compiler plugins and standalone source-code preprocessors is that parts of the host-language infrastructure are indeed directly inherited and can be reused, at least in principal. This may range from basic parsing to type-checking or even further semantic-analysis phases.

2.2.4 Syntactic Macros

Syntactic macros are a means to customize compilation in an expression-local way. Unlike whole-program preprocessors or compiler plugins it is possible for people to reason about their programs that use known linguistic abstractions (macro, procedural, or otherwise) and only where an unknown macro occurs they encounter a form of black-box behavior. As mentioned above, this enables the host language

to gradually expand in a seamless fashion. This flexibility can be seen as the big appeal of Lisp while at the same time has made it prone to fragmentation.

Due to their significant role in the development of the embedded-language concept let us take a closer look. Consider a library for addition (vec-plus) and scaling (vec-times) of vectors represented as lists. We can consider these operations to be tokens of a little language and exploit this nature.

LISTING 2.4: Vector EDSL as macros

```
1  (defun vec-plus-n-ext (e)
2    (cond
3      ((atom e) (list e))
4      ((eq (car e) 'vec-plus-n)
5       (cadr e))
6      (t (list e))))
7
8  (defun vec-opt (e)
9    (cond
10     ((atom e) e)
11     ((eq (car e) 'vec-plus)
12      `(vec-plus-n ,(append (vec-plus-n-ext (vec-opt (cadr e)))
13                            (vec-plus-n-ext (vec-opt (caddr e))))))
14     ((eq (car e) 'vec-times)
15      `(vec-times ,(vec-opt (cadr e)) ,(caddr e)))
16     (t e)))
17
18 (defun vec-opt-emit (e)
19   (cond
20     ((atom e) e)
21     ((eq (car e) 'vec-plus-n)
22      `(mapcar '+ ,@(mapcar 'vec-opt-emit (cadr e))))
23     ((eq (car e) 'vec-times)
24      `(mapcar (lambda (x) (* x ,(caddr e))) ,(vec-opt-emit (cadr e))))
25     (t e)))
26
27 (defmacro vec-plus (v1 v2)
28   (vec-opt-emit (vec-opt `(vec-plus ,v1 ,v2))))
29
30 (defmacro vec-times (v s)
31   (vec-opt-emit (vec-opt `(vec-times ,v ,s))))
```

Listing 2.4 shows the implementation of this language and a simple optimization: Binary additions are flattened into one application of **mapcar**. Furthermore, expression semantics are inlined at the macro's call (or expansion) site. For instance,

```
(vec-plus (vec-plus v1 (vec-times v2 2)) v3))
```

would, at compile time, be turned into:

```
(mapcar '+ v1 (mapcar (lambda (x) (* x 2)) v2) v3)
```

The restriction to the expression level is not always a real hindrance as even more complex programs can be expressed as expressions. Scala's implementation of syntactic macros can also work on the block level. In Scala it is also possible to

retrieve some contextual information such as typing and location (line numbers). Individual implementations have different properties and features but the concept remains the same: Subtrees of a program's *abstract syntax tree* (AST) are entirely reified at compile time and at the command and mercy of the macro author.

While great for EDSL authors in terms of customizability and for users in terms of performance and usability, there is a loss in terms of reliability. After all, within these customized subtrees, general assumptions and expectations for the workings of the host language need not necessarily apply anymore, e.g. the evaluation strategy may change. To top it off, writing macros in a way that does not interfere with the rest of the program is not an easy task [52, Chapter 10].

It would be unfair to generally judge macros negatively for all of these byproducts of the (necessary) power bestowed to them by design. However, one may question whether their entire power is required for EDSL implementation, just like custom preprocessors are an excessive solution for achieving what macros provide.

2.2.5 Template Metaprogramming

Template metaprogramming is yet another form of compile-time metaprogramming usable for language embedding. Originally introduced for generic types in C++, this way of metaprogramming encodes compile-time computations within the type system [16, 32]. Unlike other mechanisms like generics in Java that look similar on the surface, template-type-level computations are very powerful and allow precise *specialization* of data-structures as well as operation implementations.

So-called *expression templates* [111] can be used to transform expressions, e.g. for the sake of optimizations. Czarnecki et al. [32] describe this in the context of language embedding of a vector DSL similar to the one used for our syntactic macro example. It involves statically expressing the abstract syntax of an EDSL expression within the type system and building templates that combine and eventually transform it. The process of expressing and handling the computation is clever but rather unintuitive, cumbersome, and limited [32]. It is questionable whether a system essentially intended for parametric polymorphism ought to be shoe-horned (using template-coding tricks [16, 96]) into a type-level abstract syntax processor.

There also exists "template metaprogramming" for Haskell with an extension called Template Haskell [96]. Though similar in name, it is quite different from template metaprogramming in C++. In fact, Template Haskell rather falls somewhere in between the concepts of preprocessors and syntactic macros since it "allows the programming to alter the semantics of a program by transforming it into a different program before it reaches the compiler" [32]. It provides mechanisms for code reification (e.g. of declarations, expressions, etc.), generation, composition, and reflection (i.e. *splicing* in code). As such it can of course be used to implement and transform embedded DSL programs at compile time [95], and it is arguably much easier to do so than with template metaprogramming in C++. However, EDSL authors and users face the same issues as with compiler plugins and syntactic macros, but with additional boilerplate (i.e. the `$(···)` splice operator and `[|···|]` quotation).

2.2.6 Shallow Embedding

Let us now turn to pure embedding techniques. They avoid the type of meta-programming covered in the previous sections and rely on language features and abstraction mechanisms that have a clearly defined behavior at run time, i.e. the tokens of these pure EDSLs work with various kinds of values at run time.

Shallow embedding is the most obvious embodiment of this idea. In this approach an embedded language "is defined directly in terms of its semantics" [46]. In other words, there is a direct mapping between the syntax of the EDSL and its semantics in the host language [102]. For this thesis I find it worthwhile to further divide this embedding style into two subcategories, namely *immediate* and *delayed*, with nuanced differences as described in the following.

2.2.6.1 Immediate: (Non-Computational) Data Composition

The easiest way to understand a shallowly embedded DSL is when it is merely providing functionality for combining (and possibly manipulating) non-computational data [106]. The connection between function call and resulting value is immediate even if the host language's evaluation strategy itself is not strict. In most cases it is indistinguishable from a traditional library that offers utility functions and its language nature merely derives from a (small) selected range of operations that (visually) fit together well.

Take for instance the vector EDSL from Listing 2.4 that was used to showcase syntactic macros. If we implement this EDSL with functions (**defun**) as shown in Listing 2.5 instead of macros (**defmacro**) we get an immediate shallow embedding. The combined values are in fact "just" lists.

LISTING 2.5: Vector DSL shallowly embedded in Lisp

```
1 (defun vec-plus (v1 v2)
2   (mapcar '+ v1 v2))
3
4 (defun vec-times (v s)
5   (mapcar (lambda (e) (* e s)) v))
```

We can further limit the extent of this language by restricting the data types to custom ones. While possible in Lisp it is more useful in a language where static type checking enforces this and limits the range of combinations. Consider what is essentially the same EDSL but with an (externally or publicly) immutable `Vec` data type, implemented in Java as presented in Listing 2.6.

Implementing this kind of embedding is very simple, easy to use (i.e. users work only on values) and reliable (i.e. there are few surprises). Of course, the method bodies themselves are black boxes but the use of procedural abstraction, which follows the common laws of the host language, makes it easy to reason about the programs of such an EDSL. However, it is not possible to apply domain-specific optimizations related to the structure of an EDSL program as we could with macros.

LISTING 2.6: Vector (`Vec`) DSL shallowly embedded in Java

```
public final class Vec {
  final double[] elements;
  private Vec(double[] elements) { this.elements = elements; }

  public static Vec create(double... elements) {
    return new Vec(Arrays.copyOf(elements, elements.length));
  }

  public Vec plus(Vec vec) {
    double[] ds = new double[elements.length];
    for (int i = 0; i < ds.length; i++) {
      ds[i] = elements[i] + vec.elements[i];
    }
    return new Vec(ds);
  }

  public Vec times(double s) {
    double[] ds = new double[elements.length];
    for (int i = 0; i < ds.length; i++) {
      ds[i] = elements[i] * s;
    }
    return new Vec(ds);
  }
}
```

2.2.6.2 Delayed: Computation Composition

In many modern programming languages functions or closures are first-class citizens. These represent computations or computational values that commonly cannot be decomposed (at least not straightforwardly). Despite their somewhat indirect nature, EDSLs that at their core combine these computations are still commonly classified as shallowly embedded [46, 102].

To illustrate this type of embedding let us implement a language for two-dimensional geometric regions in the spirit of the one introduced by Carlson et al. [27], a common choice of example [26, 57, 102]. A region is defined as a function taking two-dimensional coordinates and returning either `true` or `false`, i.e. it is the characteristic function for a set of points. There are primitive regions such as circles and squares of a fixed size and combinators that transform or compose regions, such as scaling and intersection. Listing 2.7 shows an implementation in OCaml.

Programs of this EDSL are mere combinations of function values. So it is true that the mapping to the semantics of the host language is direct. However, the actual computation that is expressed by these regions is delayed until `is_inside` is called. In fact we do not even need this computation-triggering function in this direct representation. It is mere decoration here but it could be useful if we wanted to wrap and hide the internal representation of a region as a function.

This embedding is still easy to implement and use. However usability does suffer slightly from the delayed nature. For instance, it might become hard to debug errors that occur deeply nested within one of the delayed computation combinations.

LISTING 2.7: Region DSL shallowly embedded in OCaml

```
type region = float -> float -> bool

(* Primitives *)
let empty =
  fun x y -> false
let circle =
  fun x y -> sqrt (x *. x +. y *. y) <= 1.0
let square =
  fun x y -> abs_float (x *. x) <= 1.0 && abs_float (y *. y) <= 1.0

(* Combinators *)
let scale (r : region) sx sy =
  fun x y -> r (x /. sx) (y /. sy)
let translate (r : region) tX tY =
  fun x y -> r (x -. tx) (y -. ty)
let outside (r : region) =
  fun x y -> not (r x y)
let intersect (ra : region) (rb : region) =
  fun x y -> (ra x y) && (rb x y)
let union (ra : region) (rb : region) =
  fun x y -> (ra x y) || (rb x y)

(* Checking *)
let is_inside (r : region) =
  fun x y -> r x y
```

Such errors cannot always be avoided like in the regions example. Furthermore, the EDSL author is still not able to implement optimizations based on the structure of EDSL programs. For instance, a simple (desirable but not expressible) optimization would be to transform an expression like

$$\underline{\texttt{intersect}}\ (\underline{\texttt{union}}\ (\cdots)\ (\cdots))\ \underline{\texttt{empty}}$$

into the expression

$$\underline{\texttt{empty}}$$

in order to avoid the computational effort of checking the irrelevant (left) subexpression due to the eager host-language evaluation strategy. The opaque nature of the computational (function) values prevents this.

Another characteristic of shallow embeddings is that they only allow for a fixed number of *interpretations*, usually a single one. In the case of our region language this interpretation is region-member checking. However, it might also be interesting to perform different interpretations such as calculating the total covered area of a region. Gibbons [46] rightly points out that it is possible to add more interpretations by expanding value types to tuples, performing all interpretations, and then projecting the values of interest. This is rather cumbersome and may cause redundant computations (depending on the host language's evaluation strategy).

LISTING 2.8: Region DSL deeply embedded in OCaml

```
(* Abstract syntax *)
type region_e =
  | Empty_e
  | Circle_e
  | Square_e
  | Scale_e     of region_e * float * float
  | Translate_e of region_e * float * float
  | Outside_e   of region_e
  | Intersect_e of region_e * region_e
  | Union_e     of region_e * region_e

type region = region_e

(* Primitives *)
let empty  = Empty_e
let circle = Circle_e
let square = Square_e

(* Combinators *)
let scale     (r : region) sx sy          = Scale_e     (r, sx, sy)
let translate (r : region) tx ty          = Translate_e (r, tx, ty)
let outside   (r : region)                = Outside_e   r
let intersect (ra : region) (rb : region) = Intersect_e (ra, rb)
let union     (ra : region) (rb : region) = Union_e     (ra, rb)

(* Checking *)
let rec is_inside (r : region) = fun x y ->
  match r with
   | Empty_e  -> false
   | Circle_e -> sqrt (x *. x +. y *. y) <= 1.0
   | Square_e -> abs_float (x *. x) <= 1.0 && abs_float (y *. y) <= 1.0
   | Scale_e     (re, sx, sy) -> is_inside re (x /. sx) (y /. sy)
   | Translate_e (re, tx, ty) -> is_inside re (x -. tx) (y -. ty)
   | Outside_e   re         -> not (is_inside re x y)
   | Intersect_e (rea, reb) -> (is_inside rea x y) && (is_inside reb x y)
   | Union_e     (rea, reb) -> (is_inside rea x y) || (is_inside reb x y)
```

2.2.7 Deep Embedding

What prevents optimizations and having a multitude of interpretations is the fact that although computation is in fact reified in the case of delayed shallow embedding, the computation cannot be inspected (i.e. no intensional analysis). The technique of *deep embedding* addresses this by making the EDSL tokens construct and combine custom-defined abstract syntax at run time. This commonly represents computation but is open to different interpretations. The approach can be considered a not-so-distant relative of the so-called *interpreter pattern* [42, Chapter 5].

Listing 2.8 shows a deep embedding of the region language in OCaml with `region_e` being an *algebraic data type* (or *variant*) mirroring the EDSL tokens. The function `is_inside` is an interpreter of the abstract syntax tree generated from the tokens of the EDSL. It is easy to imagine interpreters that take into account aspects of the semantics such as for instance the operands of `Intersect_e` and change

behavior accordingly. One could even go so far as to write a compiler, e.g. to C [36].

It is also rather common to implement optimization at the time of abstract-syntax construction by using so-called *smart constructors* [36]. In our example this refers to the token functions themselves which could (but in our example do not) perform rewriting based on the provided operands' abstract syntax.

Compared to delayed shallow embedding there is not much difference in terms of usability. However, implementation is a little bit more involved and the ability to wrangle abstract syntax could more easily lead to bugs and unreliable behavior on the user side as well.

LISTING 2.9: Vector DSL deeply embedded in Java (interface only)

```
1  // Base value
2  public final class Vec {
3    ...
4    public VecE toVecE()          { ... }
5  }
6
7  // Abstract syntax
8  public class VecE {
9    ...
10   public VecE plus (VecE   vE) { ... }
11   public VecE times(double  s) { ... }
12   public Vec  toVec()          { ... }
13 }
```

The difference to immediate shallow embedding is more significant. Let us show this on a sketched deep embedding of the vector DSL in Java. Listing 2.9 shows the modified skeleton of `Vec` as well as `VecE`, which represents and is to house the language's abstract syntax (not actually shown here). Operations are not immediately performed on the vector values anymore. Instead, the idea is to first lift a value to a literal (with `toVecE`) and then perform operations defined on the abstract syntax (`VecE`). The actual calculation is delayed until `toVec` is invoked.[3]

In general, just by looking at the interface it is not possible to determine whether computation is in fact delayed. However, assuming that this is what the EDSL's documentation says, users need to become aware of this syntax construction and composition as opposed to the simplicity of a shallowly embedded version. Users cannot ignore this even temporarily and might experience a degradation of usability. After all, there is an additional, visible indirection layer in between the desired computation and its actual expression as well as execution.

In some host languages it is possible to hide some of the deep-embedding aspects using *type inference* as well as *implicit conversions* (in Scala). One could also hide deep-embedding behavior behind the `Vec` type itself, but this is only possible when the type is custom-defined (or freely extensible) and when computation is intended to be delayed in a truly lazy fashion, e.g. until vector elements are accessed. It is questionable whether entirely hiding the exact behavior from users is always helpful or an improvement to the overall interface.

[3]The design choice of using instance methods here instead of static methods is not related to the style of embedding. However, it requires the author to be in control of the data type's implementation.

Let us finally compare deep embedding with impure embedding techniques: In a certain sense, deep embedding is an emulation of syntactic macros at run time. EDSL subprograms are reified as data just like any other data in the host language is created and constructed. This happens entirely at run time and while it enables dynamic program composition, it also incurs some overhead as it does not benefit from static preprocessing.

This is counterweighted by the wide availability of the approach as well as the ability to adapt computation and EDSL handling to the actual run-time system or environment. For instance, if a bug is found in a macro's implementation it becomes necessary to recompile all the source files using that macro. With deep embedding, at least in a *dynamically linked* library environment, this is not necessary. Furthermore, deep embedding extends beyond static expressions, i.e. programs can be freely constructed (at run time) using the host language's inbuilt mechanisms. It is also possible to implement auxiliary methods that accept, build, combine, and return abstract syntax. Note that only data types that are designed for EDSL-program composition can be affected, which is a safety improvement over the black-box effects of macros on arbitrary source code within their scope.

2.2.8 Tagless-Final Embedding

So far I have only discussed rather rigid implementation approaches either with or without explicit reification of EDSL programs. However, there are also orthogonal concerns of modularity, flexibility and type safety. To this end, so-called *tagless-final* embedding has been proposed by Carette et al. [26] (inspired by Reynolds [86]).

Recall that in deep embedding flexible interpretations of (reified) EDSL programs are made possible by implementing interpreters that dispatch behavior on the actual abstract syntax elements. The core idea of tagless embedding is that this flexibility in dispatching can be achieved without EDSL-program reification.

LISTING 2.10: Vector EDSL as a type class and example program

```
1  type Vec = [Double]
2  class VecL repr where
3    lit   :: Vec -> repr Vec
4    plus  :: repr Vec -> repr Vec -> repr Vec
5    times :: repr Vec -> Double -> repr Vec
6
7  v1 = [1, 2]
8  v2 = [3, 4]
9  v3 = [5, 6]
10
11 program :: VecL repr => repr Vec
12 program = (lit v1 `plus` (lit v2 `times` 2)) `plus` lit v3
```

This is best explained by example. The Haskell *type class* [113] `VecL` in Listing 2.10 shows a language definition or rather its interface. Programs can be expressed by using only knowledge of the signatures in the type class but have no meaning by themselves. The example `program` can be "run" by providing instances of `VecL` and forcing `program`'s' *tagless interpretation* to occur with these instances.

LISTING 2.11: Tagless evaluator and viewer

```
newtype Val a = Val Vec
instance VecL Val where
  lit          v =
    Val v
  plus  (Val v1) (Val v2) =
    Val (map (\(x, y) -> x + y) (v1 `zip` v2))
  times  (Val v)        s =
    Val (map (\x -> x * s) v)

eval (Val a) = a

newtype Str a = Str String
instance VecL Str where
  lit          v =
    Str (show v)
  plus  (Str v1) (Str v2) =
    Str ("plus(" ++ v1 ++ ", " ++ v2 ++ ")")
  times (Str v)         s =
    Str ("times(" ++ v ++ ", " ++ (show s) ++ ")")

view (Str a) = a
```

Listing 2.11 shows two tagless interpreters. The expression `(eval program)` causes evaluation of the EDSL program, `(view program)` yields a textual representation. Although there is flexibility of interpretation, from an implementation standpoint the two interpreters are simply realizations of shallow embedding. This solves the issue of fixed and rigid shallow-embedding interpretations at the cost of boilerplate code, though arguably in Haskell this is only of minor concern.

However, domain-specific optimizations still require at least some degree of EDSL-program reification. The tagless-final approach allows for any type of interpretation including ones that realize deep embedding. Listing 2.12 shows a simple reifying interpreter that accomplishes this.

Note that our simple examples only expose the main properties of tagless embedding and there is certainly more to explore in terms of expressive strength

LISTING 2.12: Tagless reifier

```
data Exp a = Lit   a
           | Plus  (Exp a) (Exp a)
           | Times (Exp a) Double
  deriving( Show )

instance VecL Exp where
  lit    v    = Lit   v
  plus  v1 v2 = Plus  v1 v2
  times  v  s = Times v s

reif :: Exp Vec -> Exp Vec
reif e = e
```

LISTING 2.13: Tagless-final embedding in Scala

```
trait VecL {
  type Repr[+T]
  def lit  ( v: Vec): Repr[Vec]
  def plus (v1: Repr[Vec], v2: Repr[Vec]): Repr[Vec]
  def times( v: Repr[Vec],  s: Double): Repr[Vec]
}

trait Program { this: VecL =>
  val v1, v2, v3 = ···
  def run() = plus(plus(lit(v1), times(lit(v2), 2)), lit(v3))
}

trait Eval extends VecL {
  type Repr[+T] = Vec
  def lit  ( v: Vec)             = v
  def plus (v1: Vec, v2: Vec)    = ···
  def times( v: Vec,  s: Double) = ···
}
```

and usefulness as a general implementation technique (not only for EDSLs) [26]. The concept has inspired a variant of tagless embedding called *polymorphic embedding* in Scala [57]. A closely related, similar concept (or perspective) in the object-oriented programming (OOP) community is put forward by so-called *object algebras* [51, 82].

In Scala, one way of implementing tagless embedding is by using *mixin composition* [22, 81] as for instance employed by the Lightweight Modular Staging framework [87] using traits (in combination with abstract type members and typed self references). Listing 2.13 presents this. The EDSL program (encoded inside the `Program` trait) can be "run" with an evaluator as follows:

```
new Program with Eval {}.run()
```

In cases where *type-constructor polymorphism* [76] (or *higher-kinded types*) and multiple inheritance of state are not necessary, Java is sufficiently expressive to realize a restricted, first-order variant of this approach. Listing 2.14 presents this. Instead of traits one can use interfaces and default methods. Due to restrictions regarding multiple-interface anonymous classes in Java, "running" `Program` requires the slightly verbose definition of an explicit, named class, e.g. as follows:

```
class P implements Program<Vec>, Eval {} ··· new P().run()
```

As mentioned before, although tagless embedding lends itself to flexible EDSL-program interpretation as well as modular EDSL design, it comes with boilerplate code and associated usage (and usability) limitations. As such it appears to be much more well-suited for the "EDSL in lieu of standalone DSL" line of thinking than for directly enriching traditional library interfaces. After all, despite (or I should rather say because of) its flexibility, tagless-final embedding is not as directly usable and seamless as either straightforwardly implemented shallow or deep embedding as part of a general-purpose program.

LISTING 2.14: Tagless-final embedding in Java

```
interface VecL<R> {
  R lit  (Vec v);
  R plus (R  v1, R      v2);
  R times(R   v, double  s);
}

interface Program<R> extends VecL<R> {
  default R run() {
     Vec v1, v2, v3 = ···;
     return plus(plus(lit(v1), times(lit(v2), 2)), lit(v3));
  }
}

interface Eval extends VecL<Vec> {
  default Vec lit  (Vec  v)            { return v; }
  default Vec plus (Vec v1, Vec    v2) { ··· }
  default Vec times(Vec  v, double  s) { ··· }
}
```

2.2.9 Hybrids

It is of course possible to mix various forms of language embedding. For instance, not all tokens of an EDSL have to correspond to (reified) abstract syntax in deep embedding. Some approaches use the concept of a core language with some EDSL tokens directly constructing corresponding syntax data while others merely abstract over and combine core-language elements [89, 102].

It is also possible to implement EDSLs that dynamically adjust whether behind the scenes of a seemingly deep embedding immediate evaluation occurs or not. In fact, the same may apply to seemingly shallow embeddings.

Another form of hybridization, in fact the direction also taken by this thesis, can be found in the form of mixing impure with pure embeddings, i.e. static with dynamic program reification. For instance, this is provided by Yin-Yang [68], which mixes syntactic macros and deep embedding in order to improve usability.

With this form of hybridization some aspects or partial EDSL programs may be handled statically while others may be delayed until later. They pose a slight challenge to the understanding and notion of the language-embedding perspective the more host-language infrastructure has to be discarded.

2.3 Staged Embedded Languages

All but the simplest EDSLs, i.e. those using immediate shallow embedding, explicitly construct and compose computation.[4] This leads to a separation of EDSL-related computation into distinct stages. For instance, with syntactic macros EDSL-program reification is automated at compile time, the processing of the program happens in one stage, and the execution is a future one (happening at run time). With deep

[4]This even applies when no explicit analysis or transformation is performed as for instance in delayed shallow embedding.

embedding all computation happens at run time but is still staged, just in a more ad-hoc fashion: Procedure calls create abstract syntax, its evaluation is a future stage.

With this concept in mind, I deem it fitting to classify embedded languages that make use of this type of separation as *staged embedded languages*. The following will shed more light on the term "staging" (in the context of *multi-stage programming* (MSP) [104]) and its interpretation for this thesis. This is followed by an overview of the design space and a thorough treatment of various pitfalls that may be encountered with traditional, i.e. manual, implementations of staged EDSLs.

2.3.1 Staging

The term "staging" has come to stand for many concrete notions and implementations that involve stages. Let us trace back its origin in the context of programming languages. Taha simply states that "in essence, staging is altering a program's order of evaluation in order to change the cost of its execution" [105, Chapter 2]. While this may be too general a definition it captures the core idea of multi-stage programming quite well. It is closely related to *partial evaluation* [65], which aims to automatically and transparently find program parts that can be evaluated ahead of (e.g. at compile time) other parts to precompute and transform the program accordingly.

Taha concretizes that a multi-stage program "involves the generation, compilation, and execution of code, all in the same process" [104], while Czarnecki [32] only seems to require delay and does not ask for execution to be part of the same process (since he includes template metaprogramming). Likewise, syntactic macros have also been considered a form of multi-stage programming [43, 104]. The compilation step also does not seem crucial for the concept of staging considering that some implementations of multi-stage programming leave open whether generated computation is interpreted or compiled [87].

2.3.1.1 Syntactic Annotation

To find an interpretation fitting to our concept of staged EDSLs, let us first illustrate Taha's realization of safe, controlled multi-stage programming. The core idea is to allow the explicit construction of general-purpose program fragments at run time with ways to combine these fragments by splicing. Their compilation with subsequent execution must be triggered explicitly. The main contribution is that this is supported by a static type system that ensures already at the initial time of compilation that dynamically composed program fragments are type-safe and their compilation later at run time will not cause type errors [32, 104, 105, 115].

This concept has been explored with several languages as bases, including Java [115] and OCaml [32]. Let us here present the latter, called MetaOCaml. In addition to OCaml's standard features it offers three *syntactic staging annotations*:

- *Brackets* (`.<···>.`) for constructing code of the enclosed expression.
- *Escape* (`.~···`) for splicing, i.e. combining, code fragments.
- *Run* (`.!···`) for compiling and subsequently executing code.

LISTING 2.15: Region DSL implemented with MetaOCaml's explicit annotations

```
type region = float code -> float code -> bool code

(* Primitives *)
let empty =
  fun x y -> .<false>.
let circle =
  fun x y -> .<sqrt (.~(x) *. .~(x) +. .~(y) *. .~(y)) <= 1.0>.
let square =
  fun x y -> .<abs_float (.~(x) *. .~(x)) <= 1.0 && ···>.

(* Combinators *)
let scale (r : region) sx sy =
  fun x y -> r .<.~(x) /. sx>. .<.~(y) /. sy>.
let translate (r : region) tx ty =
  fun x y -> r .<.~(x) -. tx>. .<.~(y) -. ty>.
let outside (r : region) =
  fun x y -> .<not .~(r x y)>.
let intersect (ra : region) (rb : region) =
  fun x y -> .<.~(ra x y) && .~(rb x y)>.
let union (ra : region) (rb : region) =
  fun x y -> .<.~(ra x y) || .~(rb x y)>.

(* Checking (code generation) *)
let is_inside (r : region) =
  (.! .<fun x y -> .~(r .<x>. .<y>.)>.)
```

Note how bracketing and running resemble lambda abstraction and function application. Escaping may be, somewhat crudely, emulated by application or simple value usage. To illustrate multi-stage programming in MetaOCaml let us consider a direct conversion of the delayed shallowly embedded region DSL from Listing 2.7.

The multi-stage version shown in Listing 2.15 does not look so much different. Now, instead of functions, code fragments are combined but in both versions `is_inside` returns a function to check membership of a point in a given region. However, since in the staged version this function is actually compiled it is expected to perform significantly faster than before.

The safety of MetaOCaml derives not from type checking alone but also from the fact that the generated code is not inspectable or transformable by users [32, 87]. This is another similarity with lambda abstraction: Both mechanisms involve the reification of computational fragments as data with the internal representation (commonly) not being accessible or at least not modifiable.

2.3.1.2 Type-Based Annotation

While syntactic annotations have shown to be useful for multi-stage programming they require a custom, heavyweight language implementation with special syntax. Lightweight Modular Staging (LMS) [87] addresses this concern by offering a pure framework implementation of multi-stage programming for Scala. Instead of explicit annotations on expressions it is based on staging expressions and values by "annotating" their types, i.e. wrapping them in the framework's `Rep` type and

LISTING 2.16: Non-staged and staged power function in LMS

```
// Standard power function
def pow(x: Double, y: Int): Double =
  if (y == 0) 1.0 else x * power(x, y - 1)

// Scaffolding for multi-stage programming in LMS
trait Program { this: Arith =>
  // Staged power function
  def pow(x: Rep[Double], y: Int): Rep[Double] =
    if (y == 0) 1.0 else x * pow(x, y - 1)
}
```

relying on *type inference* for the rest of the mechanism.

Listing 2.16 shows a standard implementation as well as a multi-stage version of the power function. In LMS, the fact that `x` is of type `Rep[Double]` means that it represents computation of type `Double` and instead of normal multiplication a version for `Rep[Double]` defined in `Arith` is used. The return value `1.0` is lifted to the proper (return) type by an *implicit conversion* also defined in that trait. Being a form of tagless embedding (see Section 2.2.8) this eventually requires a concrete implementation for `Arith`.

LMS comes with an extensible system for providing and handling an *intermediate representation* (IR) that automatically eliminates (where possible) common subexpressions during construction. It also enables optimizations by rewriting in method implementations and offers machinery to define code emission and compilation. Unlike in MetaOCaml, custom transformation of reified computation is a crucial selling point of this approach (at the potential cost of safety).

Ultimately, LMS provides a sort of generalized framework for deep embedding, which has made it very popular for DSL implementations [7, 88, 100, 101]. It is further improved by Scala-Virtualized [89], a custom branch of the Scala compiler that adds features to customize Scala's inbuilt constructs depending on involved types.

2.3.1.3 A Comprehensive Definition

In both MetaOCaml as well as LMS, multi-stage programming seems to imply some degree of user-level control over which parts are staged and which parts are not. However, this criterion should not exclude other forms of staging from the term's definition. For instance, in simple deeply embedded EDSL library-interface designs it is the author who decides on abstract-syntax construction. However, it is still the user who decides to use the EDSL in the first place and where to use it.

So what is common to syntactic macros, template metaprogramming, delayed shallow embedding, deep embedding, and staging with explicit annotations? For this thesis let us define staging as: The guided and controlled reification of computation as data for the sake of optional custom processing and eventual execution.

The optionality aspect is important or MetaOCaml might not be part of the definition. Likewise, if it were forbidden in general, neither LMS nor template metaprogramming would qualify. Also, note that unlike in the initially presented

definition by Taha, performance is not necessarily the main goal here. There are also other important aspects such as domain-specific analysis and error checking.

2.3.2 The Design Space

In light of the various approaches for staging, let us consider the overall design space for staged EDSLs based on the previous discussions of implementation techniques. Without evaluatively assessing merit it can be roughly outlined on the basis of the following main properties or rather dimensions:

- ***Scope***: Describes how far the employed staging approach can reach within user programs. It may be limited to the *expression* level (as with syntactic macros), *global*, or restricted to some form of *local* scope, e.g. to a function body. One may also understand this dimension as the distance that reified computation may be *carried* across a host-language program.

- ***Delimitation***: Describes the available means for maintaining boundaries between reified computation of various languages, including the host language. This not only relates to the extent of affecting user programs but also to the concern of safety and *information hiding* in regard to intensional analysis at the back end of embedded-language implementations.

- ***Staticity***: Expresses how much of the staging process exploits or enables utilization of static information (inherently or optionally), independent of the run-time environment.

- ***Dynamicity***: Expresses how much of the staging process exploits or enables utilization of dynamic information (inherently or optionally), depending on the run-time environment.

- ***Transparency***: Inversely indicates the extent of how obvious and visible the staged nature of an EDSL is. For instance, syntactic macros lead to a quite hidden form of staging, whereas deep embedding (especially in a statically typed host language) usually makes staging more obvious and explicit.

The aforementioned list will be useful and revisited later for a comparison of approaches. Here it mainly serves the purpose of summarizing important differences and characteristics for our ongoing discourse.

2.3.3 Pitfalls of Manual Implementations

Manual implementations of staged EDSLs always entail the emulation of conceptually higher-level EDSL programs by lower-level language mechanisms such as macros or function calls. This may also apply to approaches that are already specialized for domain-specific optimizations such as rule-based preprocessors (see Section 2.2.2), since even they commonly work in absence of the notion of whole staged-EDSL programs of a given language. Consequently, EDSLs can easily turn into unreliable abstractions, which is illustrated by the following pitfalls.

2.3.3.1 Non-Compositional Staging

With manual deep-embedding implementations the mechanisms in charge of constructing an intermediate representation (IR), i.e. abstract syntax, are free to realize staging in any imaginable 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.

LISTING 2.17: Deeply embedded vector DSL usage example

```
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  }
```

Consider the example in Listing 2.17 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 themselves. As contrived as this may seem, a non-compositional-staging implementation is easily sketched as shown in Listing 2.18.

LISTING 2.18: Non-compositional staging

```
public class VecE {
  private final Vec base;
  private final List<Consumer<double[]>> ops = new ⋯;

  VecE(Vec base) { this.base = base; }

  public VecE plus(VecE vE) {
    ops.add(ds -> {
      double[] elements = vE.toVec().elements;
      for (int i = 0; i < ds.length; i++) { ds[i] += elements[i]; }
    });
    return this;
  }
  public VecE times(double s) {
    ops.add(ds -> {
      for (int i = 0; i < ds.length; i++) { ds[i] *= s; }
    });
    return this;
  }
  public Vec toVec() {
    double[] ds = Arrays.copyOf(base.elements, ⋯);
    for (Op op : ops) { op.accept(ds); }
    return new Vec(ds);
  }
}
```

Note that it is common to implement so-called *builders* [21, Chapter 2] in a some-

what 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 subsequent **`return this`** statement seems to be the current norm or principle for *fluent interface* [38, 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 functions or methods (i.e. the tokens) of the EDSL. 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) [87], as long as compositionality is not impaired.

2.3.3.2 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 wherein program reification, processing, and execution are separate. 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. That is, if documentation and source code are made available and understandable at all.

For instance, from the interface definition (see Listing 2.9) 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 a non-staged embedding. The same applies to execution. For instance, in our vector EDSL in Java, the evaluation of constructed IR could be delayed further, e.g. until one of its elements is accessed. Note that this can even affect seemingly shallowly (but in fact deeply) embedded DSLs that work on custom data types. Ultimately, for users it is not immediately obvious what kind of embedding they are dealing with.

LISTING 2.19: Opaque workflow-phase overlapping example

```
1 public VecE plus(VecE vE) {
2   if ( /* this is the 5th addition in a row */ ) {
3     return this.toVec().toVecE().plus(vE);
4   } else {
5     return /* default IR construction */;
6   }
7 }
```

To illustrate the issue with opaque overlapping of workflow phases, consider Listing 2.19. The staging method for the `plus` token here performs conditional evaluation. From an EDSL user's perspective, any time IR elements are handed to an EDSL-token method (at staging time) it could potentially entail an internal decision to (partially) evaluate early on. 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 such as IR-transforming optimizations within (smart-constructor) EDSL tokens as mentioned in earlier sections, 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 EDSL program in its original form.

2.3.3.3 Fuzzy Language Boundaries

Conceptually, *language boundaries* manifest 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 [68, 102] and is, for instance, easily enabled by syntactic macros. However, even in the case of traditional deep embedding, it is possible to make the staged nature of an EDSL less obvious by not fully exploiting static type checking, by design of the host-language (i.e. one without it) or that of constituent EDSLs (i.e. using an overly general static type like Java's `Object` across the EDSL).

Despite potential advantages for seamless EDSL usage, 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.

To illustrate this pitfall, consider the following expression (`'(···)` stands for `(quote (···))`), sharing some token names with the ones presented in Listing 2.4 but not necessarily following their implementation:

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

One could provide a language labeling of the above expression where H stands for the host language or general-purpose external code. Let us here consider arithmetic operations as part of their own language:

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

Syntactic macros (as well as preprocessors and compiler plugins) are not bound to ignore foreign-language parts. After all, for the `vec-times` macro of L_1 the parts belonging to H and L_2 are just as inspectable as those belonging to L_1.

For instance, the macro implementation might decide that `'(0 0)` is indifferent to scalar multiplication and replace `(vec-times '(0 0) (e))` with `'(0 0)`. This is a problem if `(e)` causes a side effect that might affect the result of `(f)` and the rest of the host 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 be allowed to. An inner-scope expression (s_2) cannot prevent its tampering by an outer-scope macro (s_1), so assuming L_2 here to be another macro-based EDSL implementation, it cannot force L_1 to leave its programs alone.

It is not rare to encounter bugs with macro implementations [52, Chapter 10]. It certainly is not hard to cause them, for instance by missing some cases in macro-implementation conditionals. However, tracking them down is rather difficult.

With deep embedding in dynamically typed host languages, or situations to the same effect in statically typed host languages, one may encounter similar problems. In that case IR elements can be handled by any IR-processing function since boundaries between languages only exist to the extent of ignorance (i.e. one language not knowing another language's IR). However, at the very least it is not possible to affect general-purpose code as is the case with macros. Also, visibility restriction (like **`private`** in Java) on the internal IR representation may prevent inspection even in a dynamically typed setting.

Of course, one may readily blame sloppy macro implementations for causing the described issues. However, this only supports the claim of unreliability of lower-level abstractions for language embedding. Arguably, for both EDSL authors and users it would be beneficial to have clear language-boundary guarantees. It would allow them to rely on the fact that transformations can only occur *within* a language and do not cross over to others. The following informal sketch makes this idea more concrete:

$$
\begin{aligned}
v_1 &= \texttt{(quote (0 0))} = \texttt{'(0 0)} \\
v_2 &= \texttt{(e)} \\
s_2' = v_3 &= \texttt{(+ (f) (* x y))} \\
s_1' &= \texttt{(vec-times (vec-times } v_1 \ v_2\texttt{) } v_3\texttt{)}
\end{aligned}
$$

The contents of the foreign-language expressions would be hidden and at the time of L_1's processing result to take effect (i.e. at run time), the foreign-language subexpressions would have been already evaluated to the values v_1, v_2, and v_3. Whatever the EDSL implementation decides to do internally with the above expression (s_1'), it is not able to depend on or affect the structure, values, and effects of external parts. In the case of compile-time data constants (e.g. v_1) inspection could be allowed without affecting safety.

Diligent EDSL implementations that adhere to the above mitigation scheme can help avoid transgressions but implementing this scheme is a little cumbersome. Even with a support library to simplify this, EDSL users will still not gain general, hard guarantees independent of the macros' implementations.

2.3.3.4 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

LISTING 2.20: Mini factorial

```
1  static int factorial(int x) {
2    IntV n = intVar("n");
3    IntV a = intVar("a");
4
5    return   intAssign(n, intLit(x))
6      .then( intAssign(a, intLit(1)) )
7      .then( whileDo(leq(intLit(1), n),
8               intAssign(a, mul(a, n))
9        .then( intAssign(n, add(n, intLit(-1))) ))
10     ).intRun(a);
11 }
```

within a reified 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.

Consider the code in Listing 2.20. It shows the factorial function implemented in the simple, imperative EDSL introduced in Listing 2.1, which I will call Mini. As mentioned in the very beginning of this chapter in Section 2.1.2, while static typing can impose limitations on the EDSL's syntax, aspects like *definite-assignment analysis* [50, Chapter 16] obviously need to be deferred to run time when dealing with dynamically staged programs.

In a traditional setting `intRun` may perform an analysis on the constructed EDSL program and issue an error message. Consider the situation where line 6 in Listing 2.20 has been omitted. The error message might contain the constructed EDSL program and say where *within* it the error occurs. However, it is not possible to tell the user that the issue can be traced back to line 8 where the unassigned variable `a` would be used first. This example only shows a small instance for the sake of brevity. However, the benefits that retaining static context would provide should become apparent when considering EDSL-program construction that spans several methods and larger code blocks.

Depending on the capabilities of the host language 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 2.3.3.3. Some approaches [89] propose the addition of language features for this very 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.

2.3.3.5 Redundant Processing

It is reasonable to assume that EDSL-program processing involves a non-negligible amount of computational effort. Some programs may undergo significant analysis and optimization or are even compiled and offloaded (e.g. to a GPU). In general,

this is worthwhile when the EDSL-program execution or evaluation is expected to be significantly slower without prior processing, e.g. when dealing with fixed computation to be performed on large data sets as in our motivating example in Chapter 1. A related example of such processing in a general-purpose setting is just-in-time (JIT) compilation [18, 33, 94] that may be triggered heuristically depending on many factors such as overall code size and method-calling frequency.

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. For instance, database queries or vector calculations are dynamically constructed and the library back end dynamically decides how to treat these various issued "commands".

If handled naively this is bound to cause repeated processing overhead for programs that have already been constructed and processed before. For instance, consider frequent calling of the method in Listing 2.21. Only two different EDSL programs (with different vector inputs) are ever generated here, requiring very similar processing every time they are evaluated.

LISTING 2.21: Vector EDSL redundancy example

```
1 static void example(boolean b, Vec v) {
2   VecE e = v.toVecE().plus(v.toVecE());
3   if (b) {
4     e = e.plus(v.toVecE());
5   }
6   out.println(e.toVec());
7 }
```

Furthermore, it is not uncommon to encounter dynamically staged EDSLs used in a static fashion as in Listing 2.20 (or Listing 2.21 if the conditional were omitted). In that case a naive processing unnecessarily has to deal with the same program, excluding its specific input values, over and over. Also, the dynamic nature of the staging phase itself becomes conceptually unnecessary and a cause for overhead.

It is 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 (i.e. when users may shy away from using an EDSL for its redundant processing). While it is certainly possible to devise a manual caching solution, it complicates the EDSL's implementation and design despite its being a common aspect of the higher-level staged-EDSL abstraction itself.

For statically staged EDSLs this is mostly a non-issue, i.e. when processing is limited to one time only. In this sense, only static (or mixed) staging approaches are able to optimally address the case of statically occurring EDSL programs.

Note that this pitfall is not about caching or memoization of results based on input values, but about the caching or memoization of the EDSL program's processing. Consider traditional method calls, which afford various general-purpose optimizations such as *inlining* [17], *devirtualization* [64], or *polymorphic inline caches* [58] to speed up the calling (or dispatch) of subroutines. With EDSLs the granularity is such that a constructed EDSL program itself becomes equivalent to the concept of a subroutine. Now, when method calls are used for staged-EDSL implementations,

optimizations may apply, but only on the lower-than-EDSL-program level. The notion of "calling" a whole EDSL program is not automatically exploited.

2.3.3.6 Summary

Some aspects of the presented pitfalls may be avoidable by EDSL authors with the right combination of host-language selection, discipline to follow strict design and documentation guidelines, and additional implementation effort. However, even so, the pitfalls mean that EDSL user have to check and trust the provided documentation on very basic behaviors of the EDSL.

Like manual implementations of other abstractions, manual staged-EDSL implementations forgo predictability, ease-of-implementation, and automatic optimization opportunities, in favor of potential flexibility and clever manual optimizations. This can be likened to manual procedure calls in low-level languages (e.g. assembly) based on convention (e.g. order of argument placement on a stack). It requires the trust by procedure authors that the code is properly called with arguments placed on the stack in the expected order, and the trust by users that called code performs proper argument retrieval and cleanup before returning.

2.3.3.7 Acknowledgments

This section on embedding pitfalls reproduces in part and extends the motivation section of a paper [92] to be presented at GPCE 2015, of which I am the main author.

2.4 Comparison

Having introduced the pitfalls one might encounter when using and implementing staged EDSLs, let us recapitulate and compare the language-embedding approaches introduced in the first half of this chapter. The difficulty with such a comparison lies in its broadness and the lack of direct correspondences between the properties of one approach and those of another. I have distilled the approaches' core properties using the staged-EDSL design space described in Section 2.3.2. However, note that even this cannot provide a full picture without some additional explanations.

I will accompany this discussion with an evaluative comparison of reliability, performance, and usability from an EDSL user's point of view. For this comparison these criteria are defined as follows:

- Reliability is determined by two factors:
  - ***Safeness*:** Refers to how well a given approach allows for maintaining general-purpose guarantees, such as type safeness, and more importantly domain-specific ones, such as respecting and enforcing language boundaries.
  - ***Uniformness*:** Refers to how uniform a given approach generally behaves. Uniformness allows users to make assumptions and reason about common EDSLs aspects. The pitfalls of non-composable staging as well as opaque workflow are (negatively) related to this. In general, the more black-box-like an approach may act the less likely it is uniform.

- Performance is about run-time execution time and is determined by the following three factors:
  - ***Overhead***: Refers to the amount of overhead incurred. This is related to the pitfall of redundant processing. A "good" or "high" rating for this characteristic means that there is little overhead. Purely static staging approaches generally have little (run-time) overhead.
  - ***Optimization***: Refers to the freedom and extent to which domain-specific optimizations can be implemented, or in other words the expected power of optimizations. Note that this usually inversely related to the ease of expressing these optimizations (by authors).
  - ***Adaptiveness***: Refers to how well an approach allows the adaptation and specialization of EDSL-program execution to the run-time system, configurations, or to a heterogeneous environment. This requires the ability to process whole EDSL programs.
- Although all concerns relate to overall user experience, there are certain characteristics that directly relate to ease-of-use as follows:
  - ***Seamlessness***: Refers to how many obstacles there are for the writing and reading of an EDSL program. The ability to directly interface with an EDSL is one indicator of good usability [68, 87, 89, 102]. Usage of auxiliary data types and scaffolding boilerplate negatively affects this.
  - ***Maintenance***: Refers to how easy it is to maintain applications that contain EDSL programs. For instance, approaches that modify source code at compile time are unlikely to be as maintainable as the pure embedding approaches that simply work on data constructed at run time. When the EDSL evolves (or is bug-fixed), the former requires recompilation and reduces modularity (due to limited ability to perform separate compilation), the latter does not (in a dynamic-linking setting).
  - ***Debugging***: Refers to how the debugging of client code is affected by the presence of EDSLs as well as how easily EDSL programs can be debugged. The latter is (negatively) related to the pitfall of static-context loss. Problems with the former may be encountered with systems that are able to freely transform source code and mangle debug information such that users cannot trace back the source of an error within the original (unprocessed) source code.
  - ***Documentation***: Refers to the self-documentation qualities of an approach, or in other words how easy it is to find out about the behavior (e.g. of staging) of EDSL tokens. This is important for both hidden and obvious staging. This is related to uniformness such that a system that is not uniform in its behavior is unlikely to be good at self-documentation.

The comparison overview is presented in Table 2.1. The symbols △ and ▲ are to be understood as a "good" rating, the symbols ◊ and ⧫ are to be understood as a "fair" rating, and the symbols ▽ and ▼ are to be understood as a "bad" rating. The

former versions stand for cumulative ratings for the three overall criteria, but hide the details and are only meant as a quick summary.

The selection of compared approaches is slightly condensed from the approaches presented in earlier sections. Note that only pure Lightweight Modular Staging (LMS) is considered here. As already mentioned in Section 2.3.1.2, there also exists an understanding of LMS which relies on a custom branch of the Scala compiler, called Scala-Virtualized [89]. This and other hybrid approaches have been omitted from the current comparison but will be revisited in later chapters.

Note that all the ratings pertain to criteria that, for the most part, directly affect EDSL users. It is very possible that a comparison with a different focus or different criteria definitions may lead to a different assessment.

TABLE 2.1: Comparison of implementation approaches

	Manual Preprocessing	Rule-Based Preprocessing	Syntactic Macros	Template Haskell	C++ Templates	Shallow (Immediate)	Shallow (Delayed)	Manual Deep	LMS
G Global, *L* Local, *E* Expression; *N* None, *R* Rule, *O* Opaque, *T* Typing; ⌣/▲ High/Good, ⋇/♦ Fair, ⌢/▼ Low/Bad									
STAGED-EDSL DESIGN									
Scope	*G*	*G*	*E*	*G*	*E*		*G*	*G*	*L*
Delimitation	*N*	*R*	*N*	*N*	*T*		*O*	*T*	*T*
Staticity	⌣	⌣	⌣	⌣	⌣		⌢	⌢	⌢
Dynamicity	⌢	⌢	⌢	⌢	⌢		⌣	⌣	⌣
Transparency	⌣	⌣	⌣	⋇	⌢		⋇	⌢	⋇
EVALUATIVE COMPARISON									
Reliability	▽	◊	▽	▽	◊	△	△	▽	◊
Safeness	▼	▼	▼	▼	♦	▲	▲	♦	♦
Uniformness	▼	▲	♦	♦	♦	▲	♦	▼	♦
Performance	△	◊	△	△	◊	▽	▽	△	△
Overhead	▲	▲	▲	▲	▲	▲	♦	♦	▼
Optimization	▲	♦	▲	▲	♦	▼	▼	▲	▲
Adaptiveness	♦	▼	♦	♦	▼	▼	▼	▲	▲
Usability	▽	△	▽	▽	▽	△	◊	▽	◊
Seamlessness	▲	▲	▲	♦	♦	▲	♦	▼	♦
Maintenance	▼	▼	▼	▼	▼	▲	▲	▲	▲
Debugging	♦	♦	♦	♦	♦	▲	♦	▼	▼
Documentation	▼	▲	▼	▼	▼	♦	▼	▼	♦

Manual Preprocessing. Usually the whole program (containing EDSL code) is available and can be freely transformed. Of course, this depends on the host language and what kind of guarantees an EDSL author wants to maintain, but in general there is no inherent limitation of scope. Delimitation between code of different EDSLs is nonexistent. Preprocessing and "staging" (it is very questionable whether this sort of uncontrolled reification ought to be called staging at all) is static and all its aspects can be hidden.

The evaluative-comparison ratings should be mostly self-explanatory from the previous discussions. Adaptiveness is limited since staging is static and the run-time environment is not known ahead of time. Uniformness is rated "bad" since it is entirely unclear how a preprocessor behaves. In case of a compiler plugin at least the staged IR could be expected to be uniform, but even then being a whole-program black box makes it hard to expect uniform behavior when several EDSLs (and with them several manual preprocessors) get mixed and act together.

Rule-Based Preprocessing. Rule-based preprocessing as in Broadway [53], CodeBoost [19], or GHC may have varying scope up to carrying the effect of optimizations and transformation rules across the whole program (Broadway, GHC). Delimitation may be possible on a rule-by-rule basis, e.g. precedence, but is bound to be very limited and unlike other approaches (like macros) might not even be compensatable or reinforceable with manual-implementation diligence.

The rating for uniformness and documentation are high since rule application commonly follows a fixed routine and it should in principle be easily possible to generate documentation for it. The limited rewriting framework limits optimizations and run-time system specialization since EDSL programs are not processable in their entirety.

Syntactic Macros. Syntactic macros are generally limited to expression scope and provide no mechanism for supporting (embedded) language delimitation.

The rating for uniformness is "fair" as opposed to "bad" since although a macro constitutes a black box, the subject expression is reified uniformly to a common IR. As with other static approaches the overhead is generally low since processing occurs at compile time. Similar to other static approaches, debugging is complicated by the lack of delimitation but aided by locality and static context.

Template Haskell. With Template Haskell it is not only possible to quote expressions. One can also reify existing declarations of types and functions. This makes it have a sort of global scope of staging reach. Some reification (as well as reflection and splicing) can occur in a hidden fashion while reification with quotations is more obvious, which slightly impairs transparency. There is no mechanism to support language delimitation.

As with syntactic macros the rating for uniformness is "fair" since code is reified uniformly to a common IR. Template Haskell's seamlessness rating is "fair" as opposed to "good" due to the use of splicing and quotation.

C++ Templates. C++ templates provide a form of limited staging in expression scope [32] and can use the type system to support language delimitation. Transparency is somewhat impaired due to the use of template types.

Reification occurs rather uniformly and is limited in terms of intensional analysis [32]. However, this limits optimizations and also makes specialization to a run-time environment hard. The mentioned transparency issue directly translates to a "fair" seamlessness rating.

Immediate Shallow Embedding. Immediate shallow embedding is by definition not staged in any way. Its reliability and usability ratings are high because there is only limited potential for behavioral surprises. This embedding style leads to an ideal, direct-use interface but does nothing for improving performance based on (domain-specific) optimizations or specialization.

Delayed Shallow Embedding. Delayed shallow embedding performs a very trivial form of staging at run time. The constructed computation can be carried as far as type signatures allow across the whole program and delimitation is trivially achieved by total opaqueness of anonymous functions or their equivalent (unless advanced reflection features are available and used). Transparency may be impaired slightly since function types are introduced and computation is manually (delayed and) triggered (i.e. function application).

Uniformness is affected due to potential dynamic decisions made by smart constructors but still at a "fair" rating since their decisions cannot be based on intensional analysis of their arguments. On the other hand, this means that neither optimizations nor run-time system adaptations are possible. Yet, slight overhead might be encountered in very dynamic situations. Documentation has a "bad" rating due to an insufficient degree of uniformness and little self-documentation beyond potential argument and return-type information.

Manual Deep Embedding. Like delayed shallow embedding, the scope of manual deep embedding is global: Staged terms of the embedded language can be carried around at will. Delimitation can be supported by static typing if the host language is statically typed. However, this also impairs transparency due to custom types together with wrapping or lifting of values and evaluation triggering.

Uniformness is not guaranteed as illustrated by the pitfall of non-compositional staging, which again negatively affects the documentation rating. Language boundaries can also remain slightly fuzzy, impairing safeness. The overhead of staging depends on whatever is performed in the staging and processing methods but can be somewhat mitigated with some effort.

Lightweight Modular Staging. The scope of LMS can be considered to be (semi-) local. Here this does not mean that staging is limited to a method body but that it can occur throughout the traits and classes involved. Again, delimitation can be achieved with Scala's type system. While LMS is a form of deep embedding (on top of tagless-final embedding), transparency is not as clearly impaired: Scala's type inference and implicit conversions, which make LMS possible in the first place,

do in fact hide staging quite well [87]. On the other hand, having to write trait extensions precludes direct usage [68, 89] and leads to a different kind of usage than normal deep embedding and is not perfectly suited for a library interface.

While the above results in only a "fair" seamlessness rating, the trait system and IR management of LMS does provide for a certain degree of uniformness and documentation. However, things are still manual enough that a "good" rating is not granted. Using several languages in the same code requires combining them to a larger one. Language boundaries then only exist between the host-language, i.e. the "present stage", and the combined-EDSL code, i.e. the "next stage", as a whole. When using a custom tagless interpreter for this combined EDSL, intensional analysis of all staged code becomes possible barring visibility restrictions.

If an EDSL user combines several existing tagless interpreters that exist in isolation the type system can largely ensure that they cannot inspect the internal IR representations of each other. However, in some cases it is still possible to cross these boundaries using casts (if the internal representation is not properly made private). Also, in highly modular situations it can become hard to follow which traits' behaviors prevail. Hence, there is potential for users not to be aware of where language boundaries manifest and what happens there, resulting in only a "fair" safeness rating.

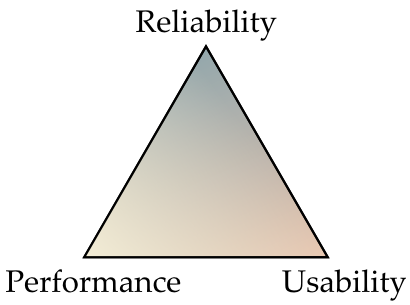
Note that the ability of LMS to combine various EDSLs to form a new one is one of its strengths in the context of modular EDSL construction and prototyping. However, the focus of this comparison is the user's view for which I deem this to be of secondary concern.

The integrated common-subexpression elimination and other such background behavior is bound to produce overhead during staging. Aside from that, LMS allows for ample opportunity to optimize EDSL programs and adapting them to the run-time system or heterogeneous architectures.

Chapter 3

Toward Higher-Level Support

The discussions of Chapter 2 indicate that there is ample room for improvement of language-embedding techniques. In particular for employing EDSLs as rich library interfaces, e.g. to be (boilerplate-) freely intermixed with general-purpose code and other EDSLs, existing approaches fail to simultaneously provide a high degree of reliability, performance, and usability.



While some issues and concerns may be inherent to the idea of language embedding as a whole (and may not escape the influence of the tradeoffs causing this state of affairs), it is still worthwhile to look for better techniques. Barring yet unforeseen developments in the general-purpose expressiveness of mainstream languages, these techniques are likely to emerge as hybrid systems mixing impure-embedding aspects such as compile-time metaprogramming with pure-embedding ones that make general-purpose host-language infrastructure lift some of the heavy work.

In this chapter I will investigate such an approach. It is to be realized as a "middleweight", pragmatic extension to the host language, taking effect during code-loading time when a program is run by end users. This lays the basic groundwork for the higher-level support for EDSLs that will be expanded on in Chapter 4. The key idea is simple enough: Explicate the concept of an embedded language and rely on this to statically extract EDSL programs. Subsequently, these EDSL programs can be processed in a custom fashion by EDSL authors.

The elaboration of a proof-of-concept implementation is followed by its evaluation and summary discussion toward the end.

3.1 Explicating the Embedded Language

In traditional, manual embedding approaches the embedded language itself remains implicit and so does the role of methods as tokens. A language only exists as a collection of (lower-level) host-language constructs that together may or may not be viewed as constituting an EDSL. The user-facing interfaces of these tokens may or may not advertise their membership to such a loose, implicit language.

LISTING 3.1: Matrix EDSL deep-embedding interface

```
1 public static MatE lit(Mat    m)            { return new Lit(m);       }
2 public static MatE add(MatE   l, MatE   r) { return new Add(l, r);    }
3 public static MatE mul(MatE   l, MatE   r) { return new Mul(l, r);    }
4 public static MatE sca(int  dim, double s) { return new Sca(dim, s); }
```

Consider the shallow embedding of a matrix EDSL in Java as shown in Listing 3.2. Its static types alone do not show which methods are tokens and which are not. In fact, it does not matter, for instance, whether we consider `create` a token or not. Expressions written with these tokens are themselves implicit anyway.

Let us turn to a deep embedding as sketched in Listing 3.1. Expressions become explicit to the EDSL author when the `MatE.eval` instance method, not shown here, is called. It triggers evaluation or *materialization*, where the term "materialization" refers to the fact that a staged expression or program represents a value or effects that are yet to materialize.

TABLE 3.1: Implicitness and explicitness

	Shallow Embedding	Deep Embedding	?	?
Language	*implicit*	*implicit*	*explicit*	*explicit*
Interface	*implicit*	*explicit*	*implicit*	*explicit*
Programs	*implicit*	*explicit*	*explicit*	*explicit*

Still, such an interface is only explicit to its human users. A meta-level observer like the compiler remains clueless about its intent. After all, it is only an EDSL program's run-time behavior that causes the reification that makes it explicit. Any approach or concrete system that desires to handle EDSL programs in a special way, e.g. in order to mitigate implementation and usage pitfalls, would need to know that `lit`, `add`, `mul`, `sca`, and `eval` are tokens and belong to the same language.

Only this understanding of language cohesion opens the door to a higher-level view and treatment of (explicit) EDSL programs, whether the interface be of an implicit or explicit nature (see Table 3.1). To this end an embedded language needs to become an explicit entity that can be referred to as such and stands in relationship to its tokens, which then can also be designated explicitly. Furthermore, common concerns such as the entry point for the processing of an (explicit) embedded language's programs can be grouped with and encapsulated in the language itself instead of remaining dispersed in methods like `eval`.

LISTING 3.2: Matrix (`Mat`) DSL shallowly embedded in Java

```
1  public final class Mat {
2    final int dimM;
3    final int dimN;
4    final double[][] elements;
5
6    Mat(int dimM, int dimN, double[][] elements) {
7      this.dimM = dimM;
8      this.dimN = dimN;
9      this.elements = elements;
10   }
11
12   public static Mat create(int dimM, int dimN, double... elements) { ··· }
13
14   public static Mat sca(int dim, double s) {
15     double[][] ds = new double[dim][dim];
16     for (int i = 0; i < dim; i++) {
17       ds[i][i] = s;
18     }
19
20     return new Mat(dim, dim, ds);
21   }
22
23   public static Mat add(Mat l, Mat r) {
24     ···
25     double[][] ds = new double[l.dimM][l.dimN];
26     for (int i = 0; i < l.dimM; i++) {
27       for (int j = 0; j < l.dimN; j++) {
28         ds[i][j] = l.elements[i][j] + r.elements[i][j];
29       }
30     }
31
32     return new Mat(l.dimM, l.dimN, ds);
33   }
34
35   public static Mat mul(Mat l, Mat r) {
36     ···
37     double[][] ds = new double[l.dimM][r.dimN];
38     for (int i = 0; i < l.dimM; i++) {
39       for (int j = 0; j < r.dimN; j++) {
40         for (int k = 0; k < r.dimM; k++) {
41             ds[i][j] += l.elements[i][k] * r.elements[k][j];
42         }
43       }
44     }
45
46     return new Mat(l.dimM, r.dimN, ds);
47   }
48   ···
49 }
```

3.2 Implicit Staging of EDSL Code

Building on the notion of explicit embedded languages, I propose an approach called *implicit staging* that is motivated by the idea of reducing the explicitness of user-facing, language-embedding interfaces to a minimum where it is a hindrance, i.e. the IR construction and materialization triggering, and retaining it where it is desirable, i.e. customized IR processing by EDSL authors [93].

It necessitates an outside, static, meta-level view and transformations on user programs. While it may resemble a syntactic-macro system and other metaprogramming approaches, it aims at providing a more reliable form of (static) staging relying on embedded-language cohesion. In fact, the envisioned kind and degree of reliability is to be equal or even superior to a well-implemented deep embedding in a statically typed setting.

With the exception of languages that allow arbitrary self modification, implicit staging can typically occur only once before the execution of a user program. Figure 3.1 shows the general, conceptual overview of an implicit-staging system for a given program and EDSL:

1. ***Staging*:** Domain-specific parts are automatically extracted, reified, and made available for processing to the EDSL's author in the form of an IR.

2. ***Processing*:** The result of this customized processing forms a so-called *residue* of the domain-specific computation.

3. ***Unstaging*[1]:** The residue is reflected within the original program, yielding a new, transformed program.

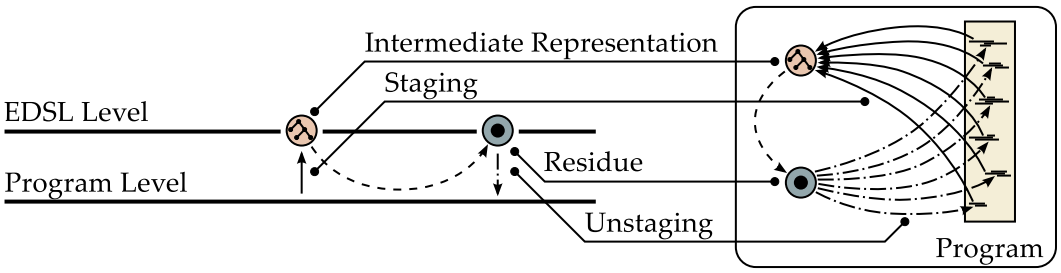


FIGURE 3.1: Implicit-staging overview

The staging step fits our earlier definition of "controlled reification" but with the additional knowledge of language membership. However, at this conceptual level it is not specified how tokens are assigned to the language, how the staging process can be configured, and what the IR and its processing actually encompass. The processing step is to be entirely defined and customized by the EDSL's author to form a desired residue, which is also not specified on the conceptual level, suitable for unstaging. Like the staging step, the unstaging step might also be guided by some form of configuration, which is to be kept simple.

[1]This is not to be confused with the "unstaging" presented by Choi et al. [31], which refers to a sort of inversion of staging (i.e. considering the unstaged version of a multi-stage program) for the purpose of improving static-analysis implementations.

3.2.1 Static Token Reinterpretation

The extraction of domain-specific code can be approached by statically simulating dynamic (run-time) staging behavior to varying degrees. Since EDSL tokens are elements of the host language, they commonly have a defined run-time behavior (i.e. method bodies). However, during implicit staging, tokens may be regarded as mere markers and identifiers for domain-specific computation.

If we assume a representation of the input user program that retains token identifiability, even if it does not retain all the original source-code structure, the tokens can be reinterpreted as performing the construction of a generic IR. However, unlike run-time interpretation or execution, as is the case with pure embedding approaches, this staging reinterpretation is to be performed statically and abstractly.

The IR can be further augmented with data-flow and control-flow information, providing detailed information on the static context in which domain-specific computation occurs.[2] For instance, this might include type information, uses and definitions, or value ranges. During processing of the IR, EDSL authors could use this to improve the residue generation, e.g. perform better optimizations. Some of it might even be necessary for the unstaging step, for instance to perform type conversions for the residue.

It may seem somewhat ironic to frame staging and static token reinterpretation like this, considering that deep embedding itself may be viewed as a run-time emulation of syntactic macros. However, recall that from a safety perspective deep embedding is superior to macros due to delimitation to general-purpose (host-language) code and, if well designed, other EDSLs as well. Also, recall that the scope of deep embedding extends beyond the expression level due to the actual flow of data, so in principal, statically emulating deep embedding seems worthwhile.

3.2.2 The Approach's Potential

Implicit staging provides the basis for exposing a refined view on the structure of domain-specific computation and only that (i.e. excluding irrelevant external aspects). In particular, this means that deep-embedding-style freedom can be approximated for traditionally shallowly embedded DSLs. For instance, in an expression like `add(mul(a, b), mul(a, b))` the common subexpression can be eliminated during processing if the EDSL author chooses to do so. However, implicit staging does not necessarily have to stop there. In fact, it can become an extension to both shallow as well as deep embedding.

The fact that implicit staging may provide contextual information about the input program and contained EDSL subprograms opens the door to optimizations that are not possible with pure embedding approaches on their own. After all, even with deep embedding, what can be inspected during (run-time) IR processing is only what has been dynamically constructed. For instance, in the deeply embedded expression `add(mul(lit(a), lit(b), mul(lit(a), lit(b)))` the fact that there is a common subexpression may eventually be discovered, but this is redundantly and possibly repeatedly done at run time. With implicit staging it could be optimized in the residue to help reduce run-time staging overhead.

[2] Note that on a conceptual level it is not specified what the IR looks like.

If we can extend our view even further, assuming an IR that provides information on dependencies between compound EDSL expressions, further optimization opportunities arise. More generally, implicit staging could not only be used to separate domain-specific computation from general purpose one, but help incorporate the relation between the two levels of computation into the EDSL's design and implementation. After all, unlike for dedicated, standalone DSLs, code snippets live within a general purpose program with its own data flow and control flow. With an appropriate interface for EDSL authors, global optimizations could be applied to EDSL programs that are intermixed and dispersed in user code.

Sometimes, the purpose of dynamically staging an EDSL program at run time is to gather as big a program as makes sense in order to increase the chance of finding redundant code and other optimization opportunities. Recall the deeply embedded matrix DSL from Section 3.1 and consider the user code in Listing 3.3. It might be wise to ever-so-slightly alter the surrounding user program in a restricted, uniform, and safe fashion to maintain as much of the dynamically generated EDSL program as possible until a matrix result needs to be materialized, i.e. when EDSL-external code needs it. For instance, there could be a potential code path in which `cE` and `dE` end up pointing to `aE` and `bE` respectively.

LISTING 3.3: Deep EDSL context example (eager)

```
1 MatE aE, bE, cE, dE;
2 ...
3 Mat e = add(aE, bE).eval();
4 out.println(mul(lit(e), add(cE, dE)).eval());
5 out.println(e);
```

Listing 3.4 shows such a lazier version. However, if it is (statically) known that optimizable situations will not occur, are unlikely, or that `eval` does not perform optimizations in this situation in the first place, it might be worthwhile to stay with the eager version of Listing 3.3 or make different changes.

LISTING 3.4: Deep EDSL context example (lazy)

```
1 MatE aE, bE, cE, dE;
2 ...
3 MatE eE = add(aE, bE);
4 out.println(mul(eE, add(cE, dE)).eval());
5 out.println(eE.eval());
```

It is my vision for implicit staging, with a sufficiently rich IR and powerful unstaging process, to eventually make it possible for EDSL developers to transparently adapt user programs in the described fashion. This would free EDSL users from the burden to consider the implementation details of the EDSL at hand (in our example the `eval` method). The biggest challenge lies in finding a way to achieve this without impairing uniformness and safeness. For now the imagined level of power will remain elusive.

3.2.3 Design Aspects

Designing an actual framework for implicit staging requires careful consideration of the following aspects:

- The choice of host language determines the kind of host-language elements that can be used as EDSL tokens. Furthermore, properties such as dynamic linking and potential self-modification capabilities may limit the extent of implicit staging. This means that not all host languages are equally well suited. Generally speaking, any language that makes static code analysis hard would seem unlikely to be a good candidate.
- The timing of performing IR construction is mainly determined by the type of representation in which user programs can be provided to the framework. While preprocessing of source code is an option, it usually comes with restrictions regarding the deployment and maintenance (as mentioned in Chapter 2) of both the EDSL itself as well as end-user applications, e.g. upgrades require recompilation. Additionally, working entirely at compile time restricts data sharing and encourages a premature code-generation phase.
- The scope of the IR, its contained contextual information, and its construction greatly influence implementation difficulty for both the framework developer as well as EDSL authors. This is one of the main hurdles anticipated for fully realizing the vision outlined in Section 3.2.2.

3.3 Load-Time Metaprogramming

While the concept of implicit staging does not mandate the exact time of program analysis, there are good reasons for choosing a time very close to execution time. The attempt at approximating pure-embedding approaches is one of them. Java's class-loading time serves this exceptionally well, as it is a language environment where compilation, class loading, and run time are closely related. Compilation yields lower-level, stack-machine-based bytecode [48] that retains sufficient language-level information (e.g. class, method, and field names). Its loading occurs on demand at run time, i.e. when a class or an element of a class is first needed.

Java has neither full-fledged compile-time metaprogramming facilities (beyond annotation processors), nor does it allow for simple compiler customization without relying on a custom compiler. However, it does allow for customized bytecode transformation at load time, i.e. when a `.class` file is loaded by the Java Virtual Machine (JVM). Using tool support for load-time metaprogramming by libraries such as Javassist [30], it is possible to build an extension such as (an instance of) implicit staging in a pragmatic fashion. It has the following advantages over compile-time preprocessing:

- ***Seamless Workflow Integration.*** There exists a dedicated mechanism to perform bytecode transformations at load time on the JVM. Hence, setting up an implicit staging implementation is not expected to be harder than using other bytecode instrumentation tools and is not expected to substantially impair software development and usage workflows.

- ***Infrastructure Inheritance.*** The compilation from source code to bytecode remains unaffected. The expression of EDSL tokens follows the tradition of EDSLs as just being common language elements. This means that tools like IDEs and type checkers can mostly be used as usual. However, unfortunately some tools might not always yield correct results and need to be configured to account for the effects of the latter processing. The initial semantics (e.g. method bodies) of tokens is retained up until the bytecode is analyzed, the tokens are reinterpreted (see Section 3.2.1), and changes are reflected.

- ***Run-Time System Adaptation.*** User programs and contained EDSL programs in bytecode remain as is until they are loaded on a specific run-time system, i.e. machine. The processing of their IR can be specialized dynamically to that run-time system. For instance, in presence of specific libraries, drivers, or hardware, EDSL expressions could be compiled to exploit these, and in their absence a fallback implementation could be used.

- ***Shared Environment.*** Loaded user programs share the same environment (including the heap) throughout the staging, IR processing, and unstaging phases. This establishes a form of *cross-stage persistence* [32, 87, 104, 115], which allows for sharing data between processing and actual execution of EDSL programs. This grants both the developers of an implicit-staging framework as well as EDSL authors various freedoms and ease-of-use as compared to systems that would necessitate workarounds such as serialization.

- ***EDSL Deployment, Maintenance, and Evolution.*** Any upgrade or patch of an EDSL's implementation (as well as of an implicit-staging framework itself) can be supplied modularly. There is no need to recompile user programs from Java source files with updated library versions. For instance, this is useful in cases where user programs are only deployed as binaries and cease to be maintained. Implicit staging at load time enables the evolution and improvements of an EDSL to still be reflected in such cases.

Of course there also remain disadvantages and challenges. Working with Java comes with the issues of late binding (i.e. virtual method calls), and dynamic linking itself, which restrict whole-program analysis. However, these issues are shared with other OOP language environments. Being able to work at load time is in so far beneficial as it allows us to consider more information on the actual state of the whole program when it is run than at compile time. However, the main technical challenge lies in having to process low-level (i.e. machine-language-like) bytecode instead of structured source code.

3.4 Proof-of-Concept Implementation

In order to concretely illustrate and evaluate implicit staging at load time, I developed a simple and limited proof-of-concept implicit-staging framework for DSL embedding in Java. It is restricted to the reification of EDSL expressions only, i.e. individual Java expressions that are composed of EDSL tokens either in a nested or chained fashion.

3.4.1 Overview

Aside from providing a language's interface, i.e. skeleton token implementations, an EDSL author is required to provide an implementation of the `TokenDeclaration` interface to specify the set of tokens of the embedded language, as well as an implementation of the `ExpressionCompiler` interface to specify how EDSL expressions are to be translated. The former implicitly configures the staging step, the latter corresponds directly to the custom-processing step mentioned in Section 3.2. Together they form the definition of an explicit embedded language (`EmbeddedLanguage`) as shown in Listing 3.5.

LISTING 3.5: Core interface

```
1  public interface TokenDeclaration {
2    boolean isToken  (CtMethod method);
3    boolean hasTokens(CtClass   clazz);
4  }
5
6  public interface ExpressionCompiler {
7    void compile(ExpressionSite expressionSite);
8  }
9
10 public abstract class EmbeddedLanguage {
11   public abstract TokenDeclaration   getTokenDeclaration();
12   public abstract ExpressionCompiler getExpressionCompiler();
13 }
```

Figure 3.2 shows a simplified, combined workflow for the usage as well as the inner workings of the prototype. EDSL users, e.g. application developers, may write EDSL expressions as they traditionally would with pure embedding approaches. After all, the tokens (i.e. method calls) may exist independently of the implicit staging framework. User programs are compiled as usual and deployed with application startup configured to use the implicit-staging framework.

At the core of the prototype lies a custom *Java agent* (for bytecode instrumentation as described in the `java.lang.instrument` package) that intercepts class

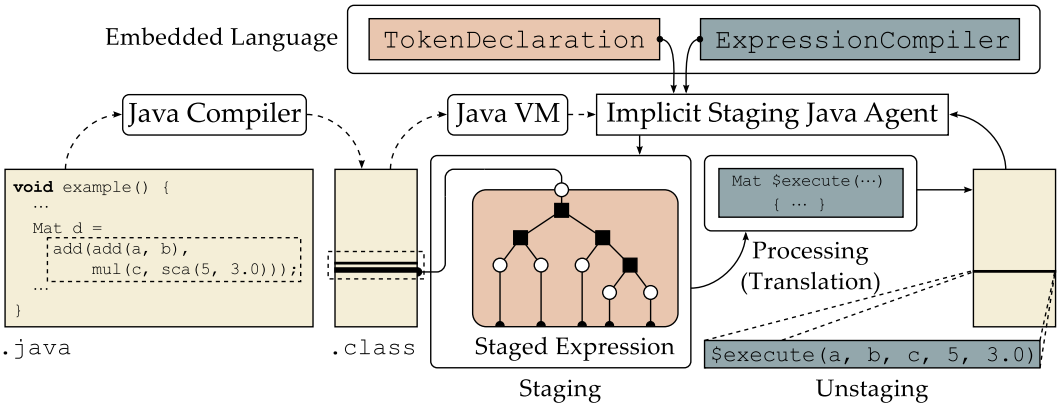


FIGURE 3.2: Implicit-staging prototype overview

loading. When an end user starts the application, the JVM feeds classes to be loaded to the agent and subsequently finalizes the loading of the returned, potentially transformed classes. Staging, processing, and unstaging are all performed within this agent.

During the staging phase, the bytecode in method bodies is analyzed and all contained EDSL expressions are extracted according to the token declaration provided by the EDSL developer. Note that in the interest of simplicity, Figure 3.2 only shows this simplified for a single expression. Then, for the EDSL-specific custom processing all expressions are eventually translated to static methods one by one using the provided expression compiler. Finally, unstaging consists of replacing the original EDSL expressions with calls to the corresponding methods.

3.4.2 Staging: Expression Extraction

The staging process is configured and guided by the token declaration of an EDSL. It specifies which methods[3] belong to the EDSL and is provided as an implementation of the `TokenDeclaration` interface with the following methods:

- **`boolean`** `isToken(CtMethod method)`, a characteristic function for membership of a method in the set of EDSL tokens.
- **`boolean`** `hasTokens(CtClass clazz)`, a method to help quickly exclude classes that do not contain EDSL tokens.

The prototype uses the `hasTokens` method to skip the analysis of classes which do not refer to classes containing EDSL tokens. It only serves optimization purposes. The classes `CtMethod` and `CtClass` are reified method and class types similar to `java.lang.reflect.Method` and `java.lang.Class`, provided by Javassist [30] used in our implementation. Additionally, the prototype offers a default implementation of the `TokenDeclaration` interface, which allows simple registration of tokens and implements the interface by standard semantics for superclass and interface lookup.

Being equipped with the information necessary to distinguish between general-purpose and EDSL-specific parts of a user program, the prototype can perform staging by means of a simple abstract-interpretation, forward-flow data-flow analysis approach [17, 71]. This is similar to the analysis used for backwards compatibility in the JVM's bytecode verifier, which relies on type inference [50, Section 4.10.2]. In fact, the prototype's implementation is based on an existing type analyzer found in Javassist (`javassist.bytecode.analysis`). A trivial linear scan of the input bytecode is not sufficient, since compound EDSL expressions are not guaranteed to be neatly clustered after compilation and depend on the flow of data and control, even with the restriction of only handling expression scope.

Recall that the idea of implicit staging extends beyond mere syntactic extraction. Instead, it attempts to statically interpret tokens as if they were deeply embedded and thus retrieve a static, anticipated shape of EDSL expressions, though limited it may currently be. Furthermore, static analysis is necessary in order to retrieve information on the static context of an extracted expression.

[3] This could be extended to fields but the current implementation is limited to methods.

3.4.2.1 Intermediate Representation

The staged IR in terms of section 3.2 is simply a collection of the contained EDSL expressions' ASTs. In the following I will discuss their representation. Every instance of `Expression`, the prototype's AST data type, represents what would be a value during an actual execution of the bytecode. It holds at least:

- Its *positions*, i.e. the positions of the instructions that caused the original expressions to be placed on the operand stack (before a potential merge).
- Its *type*, i.e. the type of the value the expression would have during actual execution (as specific as this can be determined statically).
- Its *value number*, i.e. a number that can be used to determine whether two expressions would yield the same concrete result during execution.

Type analysis and value-number analysis are performed as part of the same data-flow analysis. The latter is very simple and not to be confused with full-fledged *global value numbering* [91]. It rather resembles an *alias analysis* or *pointer analysis* [55]. It only tracks storing and loading of local variables and some stack operations such as duplication. Type analysis follows the mentioned component already present in Javassist. Hence, for the sake of brevity, a detailed description of these analyses will be omitted for the rest of the description.

Local variables, or `StoredLocal` (*loc*) instances, store the same information with the difference that it holds *stored-by positions* instead of positions, i.e. the positions of the instructions that caused the storing of the local variable.

There is only one concrete subtype of `Expression` that is considered EDSL-specific: `InvocationExpression` (*inv*). In addition to the general information, it holds both the EDSL-token method and its arguments represented as expressions.

A similar expression type is `ConversionExpression` (*cnv*) that wraps a convertee expression. It integrates with the arguments of invocation expressions to bookkeep for potential type conversions. i.e. casting as well as boxing of primitive types and unboxing of their reference-type counterparts).

Instances of the following expression types constitute the terminal leaves of a resulting expression AST and are considered *argument expressions* or *parameter expressions* (`ParameterExpression`) as they stand for (and only occur as) the arguments to domain-specific computation:

- `LocalAccessExpression` (*lac*) holds the stored local variable that is accessed and its potential indices in the (containing method's) local-variable array.
- `StringConstantExpression` (*str*) and `NullExpression` (*nul*) stand for (and hold) constant values. This could be easily extended to other kinds of constants.
- `StandaloneExpression` (*sta*) wraps an expression, including the EDSL-specific *inv*, that is to be treated as "standalone".

- `UnknownExpression` ($\top$) stands for a value resulting from unknown, usually EDSL-external, computation.

The `StandaloneExpression` type requires additional explanation. Consider the expression `add(a, d = mul(b, c))`. The `mul(b, c)` part is required to be considered standalone since it could be shared with EDSL-external code.[4] In the prototype only uniquely used argument expressions are directly exposed to EDSL authors. For the current discussion *sta* can be considered equivalent to $\top$.

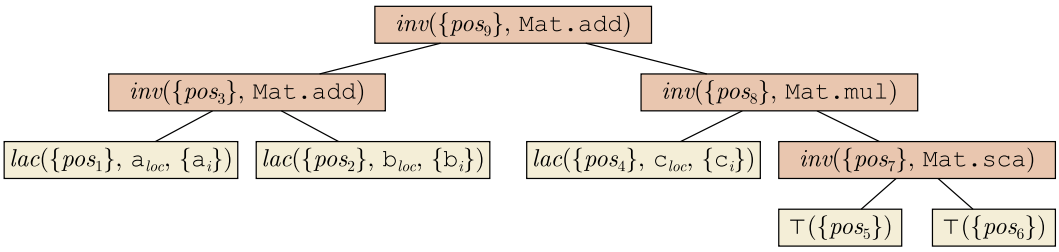


FIGURE 3.3: Extracted expression AST

Figure 3.3 shows the AST that would be yielded by extracting the following expression from the bytecode of a method body:

```
add(add(a, b), mul(c, sca(5, 3.0)))
```

The reason for the two rightmost leaf expressions being $\top$ is that the prototype currently does not handle numeric constants as indicated earlier. We would get the same result if the two values came from non-EDSL method calls or other external code, as for instance in the following expression:

$$\texttt{add(add(a, b), mul(c, sca(}\underbrace{\texttt{i + j}}_{\top}\texttt{, }\underbrace{\texttt{Math.sqrt(9.0)}}_{\top}\texttt{)))}$$

3.4.2.2 Abstract Interpretation (Transferring States)

The abstract interpretation of Java bytecode models *stack-frame* (of the call stack, not to be confused with the operand stack alone) states, i.e. the states of the operand stack and local variables at given program points. As previously mentioned, in the prototype the abstract domain of the modeled "values" are expressions (and stored local variables). As is common in data-flow analysis, a so-called *transfer function* models the effects of bytecode instructions by transferring a subject state before interpreting an instruction to that after it.

The following describes those aspects of the transfer function that are specific to the expression extraction in the prototype:

[4] The reason this comes up in the first place here is due to the translation of the shown idiom in bytecode as stack-top duplication with subsequent storing to a local variable.

(i) If an invocation instruction for a method m is encountered, it is first checked whether m is a token of the EDSL using `isToken`. If so, the arguments for this method are popped from the stack and used to create a new invocation expression (inv) (in combination with the instruction's position and m), which is subsequently added to a (global) collection of extracted expressions. If the method returns a value (i.e. its return type is not **`void`**), the expression is additionally pushed onto the stack. If m is not a token but one of the boxing and unboxing methods, an expression is popped from the stack, wrapped into a conversion expression (cnv) (with the instruction's position), and pushed onto the stack. Checked-cast instructions are handled in a similar fashion.[5]

(ii) If a store instruction is encountered, an expression is popped from the stack and a stored local variable (loc) is created (with the store instruction's position) and placed into the (modeled) local-variable array of the stack frame. In addition, the positions of the popped expression are marked as standalone.

(iii) In case of a load instruction, the associated stored local variable (loc) is retrieved, a new local access expression (lac) is created (containing this loc as well as the load instruction's position), and pushed onto the stack.

(iv) Handling the various constant instructions is trivial.

(v) Any other bytecode instruction or case that causes popping of the stack marks the popped expression's positions as standalone. Any push onto the stack that is not part of the aforementioned cases causes an unknown expression ($\top$) with the instruction's position to be pushed onto the stack.

The abstract interpretation also performs a so-called *occurs check* in cases of expression creation that depends on itself as an argument. In that case the argument will be considered standalone (as per the purpose of sca). This situation is rare, and does not occur with code generated by Java's standard compiler `javac`.

3.4.2.3 Abstract Interpretation (Merging States)

The transfer function is not sufficient for data-flow analysis. The reason for this is that when the abstract interpretation encounters a branching instruction, it needs to explore all the branches. This is usually done by adding the targets of jump instructions to a (global) queue which contains the potential next instructions (or rather their positions) to be handled, i.e. transferred. In the linear, non-jump cases simply the following instruction is considered. This only happens in case of a preceding state change and the overall abstract interpretation continues until this queue is empty.

When the mentioned branches join back together (e.g. after an **`if`** statement), the states of these branches need to be merged. This is achieved by not only adding next-instruction positions to the queue, but also merging the "out" state with the "in" states existing for these target positions. In the prototype the stack frames are

[5] In the prototype primitive conversions are simply considered unknown ($\top$).

merged by element-wise merging of the contained expressions and local variables using a *merge function*. It can be summarized as follows:

$$
\begin{aligned}
merge_{\bar{l}}(loc_1, loc_2) &= loc \text{ with } loc_1^{pos} \cup loc_2^{pos} \\
merge_{\bar{e}}(inv_1, inv_2) &= \begin{cases} inv \text{ with } inv_1^{pos} \cup inv_2^{pos}, & \text{if } inv_1^{method} = inv_2^{method} \\ \quad inv_1^{method}, & \quad \text{and } n = inv_1^{argN} = inv_2^{argN} \\ \quad merge_{\bar{e}}(inv_1^{arg_1}, inv_2^{arg_1}), & \\ \quad merge_{\bar{e}}(inv_1^{arg_2}, inv_2^{arg_2}), & \\ \quad \vdots & \\ \quad merge_{\bar{e}}(inv_1^{arg_n}, inv_2^{arg_n}), & \\ \top \text{ with } inv_1^{pos} \cup inv_2^{pos} & \text{otherwise} \end{cases} \\
merge_{\bar{e}}(str_1, str_2) &= \begin{cases} str \text{ with } str_1^{pos} \cup str_2^{pos}, & \text{if } str_1^{value} = str_2^{value} \\ \quad str_1^{value} & \\ \top \text{ with } str_1^{pos} \cup str_2^{pos} & \text{otherwise} \end{cases} \\
merge_{\bar{e}}(nul_1, nul_2) &= nul \text{ with } nul_1^{pos} \cup nul_2^{pos} \\
merge_{\bar{e}}(\bar{e}_1, \bar{e}_2) &= \top \text{ with } \bar{e}_1^{pos} \cup \bar{e}_2^{pos} \text{ if } e_1 \neq e_2 \text{ or } e_1 = \top \text{ or } e_2 = \top \\
merge_{\bar{e}}(lac_1, lac_2) &= lac \text{ with } lac_1^{pos} \cup lac_2^{pos}, \\
&\quad merge_{\bar{l}}(lac_1^{local}, lac_2^{local}), \\
&\quad lac_1^{indices} \cup lac_2^{indices} \\
merge_{\bar{e}}(cnv_1, cnv_2) &= cnv \text{ with } cnv_1^{pos} \cup cnv_2^{pos}, \\
&\quad merge_{\bar{e}}(cnv_1^{convertee}, cnv_2^{convertee})
\end{aligned}
$$

(i) Positions and local-variable indices are merged using set union.

(ii) Merging stored local variables (loc) yields a stored local variable (loc) with merged elements.

(iii) Merging invocation expressions (inv) yields an invocation expression (inv) with merged elements (arguments, etc.) if they share the same token, otherwise $\top$ with merged positions.

(iv) Merging constant expressions yields the same constant if they share the same value and are of the same kind, otherwise $\top$ with merged positions.

(v) Merging different types of expressions and merging any expression with $\top$ always yields $\top$ with merged positions.

(vi) Merging expressions of the same kind and not of the aforementioned cases yields the same expression with merged elements.

Merging with a yet undefined element of the stack frame is realized simply by overwriting. Merging stack frames of different size should not happen and when detected produces an error.

3.4.2.4 Postprocessing

After a fixed point is reached, i.e. transferring and merging of states do not produce new results, the data-flow analysis stops. In a final postprocessing step the global collection of expressions is then purged of true subexpressions. Also, expressions whose positions have been marked as standalone during transfer are turned into standalone (sta) expressions.

Having introduced the general workings, I can now exemplify the effects of the abstract interpretation. Consider the following expression in source and bytecode:

$$\underline{\texttt{mul}}\texttt{(a, } \underbrace{\texttt{x > 0 ? } \overbrace{\underline{\texttt{add}}\texttt{(b, c)}}^{e_1} \texttt{ : } \overbrace{\underline{\texttt{mul}}\texttt{(b, c)}}^{e_2}}_{merge_{\bar{e}}(e_1, e_2)}\texttt{)}$$

$pos_1 \rightarrow$	o_1	:	`aload`	$\texttt{a}_i$
	o_2	:	`iload`	$\texttt{x}_i$
	o_3	:	**`ifle`**	o_8
$pos_2 \rightarrow$	o_4	:	`aload`	$\texttt{b}_i$
$pos_3 \rightarrow$	o_5	:	`aload`	$\texttt{c}_i$
$pos_4 \rightarrow$	o_6	:	`invokestatic`	`Mat.`$\underline{\texttt{add}}$
	o_7	:	**`goto`**	o_{11}
$pos_5 \rightarrow$	o_8	:	`aload`	$\texttt{b}_i$
$pos_6 \rightarrow$	o_9	:	`aload`	$\texttt{c}_i$
$pos_7 \rightarrow$	o_{10}	:	`invokestatic`	`Mat.`$\underline{\texttt{add}}$
$pos_8 \rightarrow$	o_{11}	:	`invokestatic`	`Mat.`$\underline{\texttt{mul}}$

Java's ternary operator is not reconstructed by the analysis. Instead, the analysis deals with this situation by merging the stack frames at the end of the two conditional branches. For the case that `x` is greater than zero we have the following expression AST at the top of the abstract, modeled operand stack:

$$e_1 = inv\big(\{pos_4\}, \texttt{Mat.add}, \big[lac(\{pos_2\}, \texttt{b}_{loc}, \{\texttt{b}_i\}), lac(\{pos_3\}, \texttt{c}_{loc}, \{\texttt{c}_i\})\big]\big)$$

For the case that `x` is at most zero we get the following AST at the top of the stack:

$$e_2 = inv\big(\{pos_7\}, \texttt{Mat.mul}, \big[lac(\{pos_5\}, \texttt{b}_{loc}, \{\texttt{b}_i\}), lac(\{pos_6\}, \texttt{c}_{loc}, \{\texttt{c}_i\})\big]\big)$$

The data-flow analysis needs to merge these two expressions when control flow merges, yielding $\top(\{pos_4, pos_7\})$. Hence, the outer expression will be:

$$e_3 = inv\big(\{pos_8\}, \texttt{Mat.mul}, \big[lac(\{pos_1\}, \texttt{a}_{loc}, \{\texttt{a}_i\}), \top(\{pos_4, pos_7\})\big]\big)$$

This means that the analysis would yield and eventually list all three expressions e_1, e_2, and e_3 separately. Note that if both e_1 and e_2 were invocations of the same method this would not be the case, since both would merge into a true subexpression of an expression similar to e_3 but with a known second argument.

3.4.3 Processing: Expression Translation

The expressions resulting from staging are wrapped into so-called *expression sites* (`ExpressionSite`) one by one and supplied to the expression compiler provided by the EDSL instance. Expression sites represent the place and context in which an expression was staged and offer methods to support expression translation.

3.4.3.1 Translation to Source Code

Implementing the `ExpressionCompiler` interface directly allows EDSL developers to provide meaning to staged expressions in the form of Java source code. This interface only requires one method to be implemented:

```
void compile(ExpressionSite expressionSite)
```

Connecting parameter expressions with run-time values is accomplished indirectly. Namely, the passed `ExpressionSite` instance offers utility methods to generate source code for value access from `ParameterExpression` nodes.

The translated code for the whole expression is passed back to the given expression site via an instance method called `setCode`. This is a rather low-level way of performing translation.

3.4.3.2 Translation to Live Objects

To simplify translation and to exploit the shared heap at load time the prototype offers a higher-level alternative to the aforementioned to-source-code compilation: Translation to live objects. EDSL authors can implement such a translation by extending the abstract class `ExpressionToCallableCompiler`, which itself implements the low-level `ExpressionCompiler` interface.

It requires a concrete implementation of the `compileToCallable` method, which returns an instance of type `Callable`. Eventually, the prototype framework will replace the original EDSL expression (site) with a call to the `call` method of the returned `Callable` instance. The `Callable` interface is similar to the interface of the same name found in the Java API but its `call` method takes an argument of type `Environment`. During execution time, this environment serves as storage for the actual execution-time arguments passed to the staged EDSL expression.

Environment elements can be accessed through instances of the `Variable` class, which trivially implements the `Callable` interface. Internally, these variables are wrapped indices into the environment and provide access methods. The `ExpressionToCallableCompiler` class provides factory methods to create variables from parameter expressions or fresh ones that can be used as intermediate values. There will only be one entry `Callable` instances per expression site that might be shared by several threads during execution. Hence, an intermediate value should usually not be stored in a field of a `Callable` instance unless it is wrapped into a `ThreadLocal` instance.

Glue code generated by the framework implementation establishes that, during execution time, retrieving the value of a variable created from a parameter expression will yield the value of the associated argument. The high-level compiler internally implements the low-level `compile` method in three steps:

1. The EDSL-author-implemented `compileToCallable` is called.
2. An accessor class is created and the return value from the first step is written to a static field of this accessor class (using Java's run-time reflection facilities).
3. Glue code is generated. It creates an `Environment` instance filled with the expression's arguments and calls the `Callable` instance via its accessor class. This code includes boxing, unboxing, and checked casting if required.

As a concrete illustration, consider an expression representation for our matrix EDSL as a tree with node types `Add`, `Mul`, and `Sca`, which implement the `Callable` interface with semantics close to the shallow-embedding methods of similar names introduced in Section 3.1. We also consider two additional types: `AddN`, representing n-ary matrix addition (using a single accumulator), and `Scale`, representing the scaling of a matrix by a given factor (see Listing 3.7).

Consider again the expression of Figure 3.3):

```
add(add(a, b), mul(c, sca(5, 3.0)))
```

A high-level compiler can be defined to optimize and translate this expression's AST (for instance, using the *visitor pattern* [42]) to the mentioned `Callable` tree, yielding the structure presented in Figure 3.4.

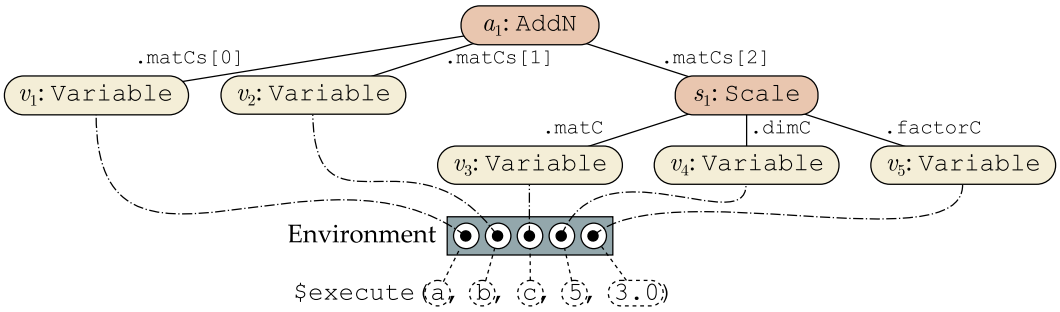


FIGURE 3.4: Translated to tree of `Callables`

As described, the compiler keeps a mapping between parameter expressions and variables for generating the correct glue code to fill the environment with values. Listing 3.6 shows this glue code, assuming the generated accessor class is called `$CallableAccessor`. Glue-code generation happens behind the scenes and can safely be ignored by EDSL authors. All of this allows the custom definition of a staged expression's semantics via an essentially static, computational-object entry point. What this looks like internally is in the hands of the EDSL author.

LISTING 3.6: Glue code in `$execute` method

```
1 private static Mat $execute(Mat u, Mat v, Mat w, int x, double y) {
2   Object[] values = { u, v, w, Integer.valueOf(x), Double.valueOf(y) };
3   Environment env = $CallableAccessor.createEnvironment(values);
4   return (Mat) $CallableAccessor.callable.call(env);
5 }
```

LISTING 3.7: AddN and Scale

```
class AddN implements Callable<Mat> {
  final Callable<Mat>[] matCs;

  public AddN(Callable<Mat>... matCs) {
    this.matCs = matCs;
  }

  public Mat call(Environment env) {
    Mat[] mats = new Mat[this.matCs.length];
    for (int i = 0; i < ms.length; i++) {
      mats[i] = this.matCs[i].call(env);
    }

    int dimM = mats[0].dimM;
    int dimN = mats[0].dimN;
    ...
    double[][] ds = new double[dimM][dimN];
    for (int k = 0; k < mats.length; k++) {
      for (int i = 0; i < dimM; i++) {
        for (int j = 0; j < dimN; j++) {
          ds[i][j] += mats[k].elements[i][j];
        }
      }
    }

    return new Mat(dimM, dimN, ds);
  }
}

class Scale implements Callable<Mat> {
  final Callable<Integer> dimC;
  final Callable<Mat> matC;
  final Callable<Double> factorC;

  public Scale(Callable<Integer> dimC, Callable<Mat> matC,
               Callable<Double> factorC) {
    this.dimC = dimC;
    this.matC = matC;
    this.factorC = factorC;
  }

  public Mat call(Environment env) {
    Mat mat = matC.call(env);
    double factor = factorC.call(env));
    ...
    double[][] ds = new double[mat.dimM][mat.dimN];
    for (int i = 0; i < mat.dimM; i++) {
      for (int j = 0; j < mat.dimN; j++) {
        ds[i][j] = mat.elements[i][j] * factor;
      }
    }

    return new Mat(mat.dimM, mat.dimN, ds);
  }
}
```

3.4.4 Unstaging: Relinking Expression Sites

Having translated all expressions and provided method bodies (e.g. as shown in Listing 3.6) for the expression sites, the framework then needs to establish the appropriate, corresponding links in the user program. This corresponds to the conceptual unstaging phase of implicit staging.

For each expression site a (uniquely named) static method (like `$execute` in Figure 3.3 and Listing 3.6) with the expression site's combined, thus flattened, type signature is added to the surrounding class, and its body is endowed with the provided source code. Javassist comes with an inbuilt, custom compiler that makes this possible. Subsequently, every instruction associated with non-parameter (sub-) expressions of an EDSL expression site are removed from the bytecode (i.e. replaced by `nop` instructions). Finally, a call to the associated method is inserted at the expression site's position.

For the sake of brevity, I omitted the description of some minor details of the implementation here, like the exact method of bytecode editing and the treatment of issues such as a potential exceeding of the maximum number of method parameters[6] (255, as imposed by the JVM).

3.5 Evaluation

The evaluation of the implicit-staging proof-of-concept implementation is split into four parts, the first three of which cover the discussion of reliability, performance, and usability aspects. The last part compares with recent, closely related work.

3.5.1 Reliability

In principle, the described staging process can establish safeness via language cohesion. This means that only tokens of the same embedded language will be reified together and exposed to the author-defined processing, i.e. the expression compiler. Conceptually it realizes the scheme for avoiding fuzzy language boundaries that was proposed in Section 2.3.3.3 of Chapter 2.

This is foiled by the fact that the implicit-staging prototype is not designed to handle (and resolve) several EDSLs at the same time. Instead, it handles different EDSLs in sequence. However, nothing prevents a method to be assigned to several embedded languages. Then the end result depends on the order of handling the various EDSLs supplied to the framework. The problem lies with the mechanism for token declaration, which is also to be blamed for shortcomings in the provision of uniformness guarantees: A `TokenDeclaration` instance is entirely free to (dynamically) change its behavior, i.e. that of the characteristic function. Hence it is a black box that may, though unlikely, arbitrarily redefine the EDSL during staging.

These issues result from an overly liberal interface and could be addressed by simply restricting and fixing the way tokens are defined. It does not seem that the current degree of freedom in declaring tokens is actually necessary. With minimal effort and reasonable diligence the described issues are easily avoided, which raises the reliability ratings (barely) to "fair".

[6] In short, this is tackled by generating a class for passing surplus arguments in its instances.

3.5.2 Performance

The overhead incurred by staging and processing at load time is expected to be quite significant, though specific to the implementation of both the prototype framework as well as the EDSL at hand. However, barring class reloading this overhead is only incurred once when an end-user application is run. On the other hand, there is no execution-time overhead for staging or processing, as would be incurred in the case of deep embedding.

Optimizations, at least on the expression level, can be implemented freely. As is also the case with syntactic macros and other static staging approaches, run-time values (at AST leafs or parameter expressions in the prototype) are not inspectable and EDSL programs cannot be dynamically constructed.

However, load-time metaprogramming brings a different kind of dynamicity. At the time of processing values specific to the run-time system can be accessed and constants of already initialized classes are available. Processing (and possibly offloading code generation) can easily be adapted on the fly.

In order to further investigate the claims regarding overhead and optimizations, experiments were performed, which are described in the following. However, note that in practice results will vary between EDSLs and, as mentioned before, implementation approaches are not always directly comparable. Any experiment will not be able to prove that one approach is generally always better than another.

Three versions of the matrix EDSL were implemented using shallow embedding (S), deep embedding (D), and the described prototype (with compilation to `Callable`) imitating the look-and-feel of the shallowly embedded version (P). The latter two perform optimizations as indicated in Section 3.4.3.2, i.e. fusion of binary additions and turning multiplications with scaling matrices into scaling operations with further fusion when applicable. I made the utmost effort to keep these implementations as comparable as possible to each other. In the deep embedding the optimization are implemented as on-the-fly rewriting. The goal was to be as fair and conservative as possible. Optimizing only during evaluation would have lead to costly re-traversals and unfairly advantaged the implicit-staging version.

The goal of the experiment was to assess the amount of execution time saved in comparison with shallow embedding as well as deep embedding applied to static expressions, while still exposing the same interface as shallow embedding. In order to model a wide variety of situations and opportunists for optimization, randomly generated expressions up to a depth of 5 (`Mat`-typed variables and `sca` expressions are counted as leaves here) were considered. For each depth 30 such expressions were generated, each occurring once in a warm-up loop and once in a loop for which execution time was measured. The warm-up loop is necessary to ensure that before measuring JIT compilation of the staging methods (i.e. optimizing variants of the ones shown in Listing 3.1) and `MatE.eval` has occurred (or not) in the deeply embedded version. For the implicitly staged version the same applies to the `evaluate` method of the various `Callable` implementations. The amount of iterations to be used for warm-up was established by experimenting with various hand-written expressions of the matrix EDSL. Additionally, random 8×8 matrices and scalar values of type **`double`** were generated to serve as arguments for these expressions and were assigned to local variables as literal-generation time was of no

interest. The generated benchmark code was also adapted for the deeply embedded language version. All randomness was only part of the benchmark-code generation.

The benchmark code was run three times for each version with 10,000,000 loop iterations for warm-up and measurement, each on a 3 GHz Intel Core i7 machine with 8 GB of RAM with JRE 7 (Java HotSpot™ 64-Bit Server VM). Figure 3.5 shows the results, aggregated by averaging over all 30 expression execution times per expression depth. See Figure 3.6, 3.7, and 3.8 for per-depth results.

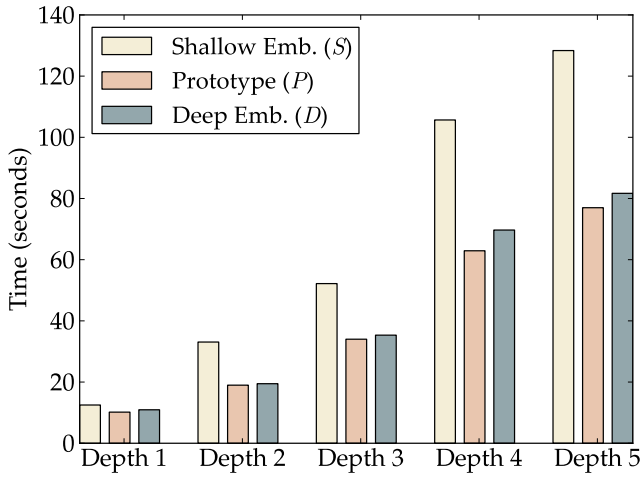


FIGURE 3.5: Aggregated (random) benchmark results

Let us summarize: For expressions at depth 1, the implicit-staging version was faster than shallow embedding for 7 of the 30 expressions and faster than deep embedding for 21 of the 30 expressions. This can be explained by the low rate (or probability) of optimization opportunities for expressions of depth 1 and the added overhead of boxing and `Callable` calling. Still, the maximum slowdown experienced at depth 1 was only by about 6.1% compared to shallow embedding and 4.7% compared to deep embedding. On average, implicit staging was 22.9% faster than shallow embedding and 7.6% faster than deep embedding.

For expressions at depths 2 to 5, implicit staging was faster than shallow embedding for more than 25 of the 30 expressions each. At depth 2, deep embedding was still faster than implicit staging for 17 of the 30 expressions, but for deeper expressions implicit staging was faster than deep embedding for more than 26 of the 30 expressions each. It appears that in the cases where deep embedding was faster, boxing of **`double`** values is to blame for the slowdown. Overall (depth averages), implicit staging sped up execution compared to shallow embedding at minimum by 22.9% and at maximum by 74.3%. Compared to deep embedding, implicit staging sped up execution at minimum by 2.5% and at maximum by 10.8%.

The benchmark on randomly generated expression was supplemented with a similar version wherein the generated expressions had a bias toward containing optimizable structures, e.g. an addition expression is likely to become part of another addition. It is no surprise that shallow embedding did not fare well in this biased experiment. Even deep embedding seems to fare worse than it did in the non-biased expressions experiment. Even so, there are cases, i.e. expressions, where implicit

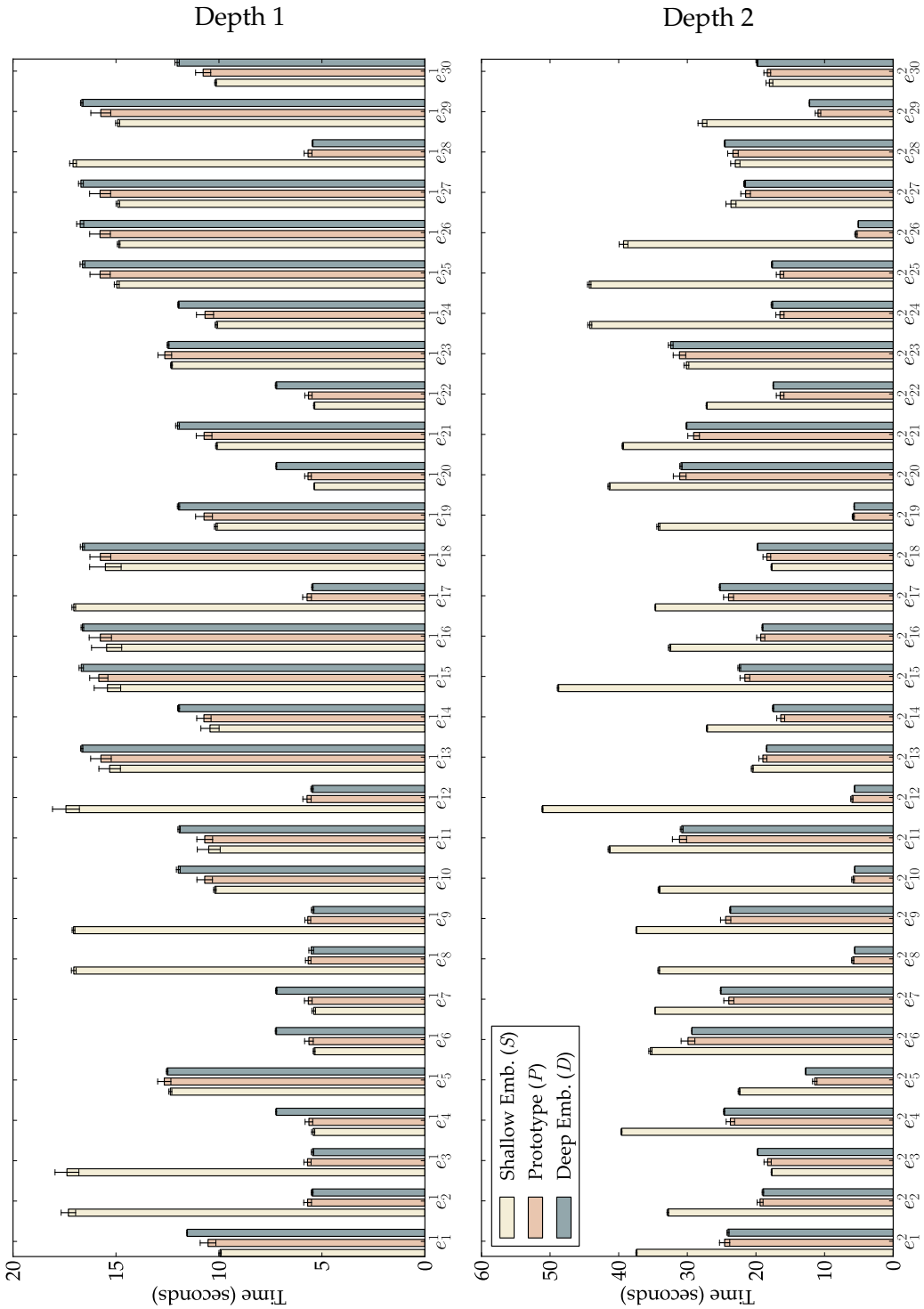


FIGURE 3.6: Depth 1 and 2 (random) benchmark results

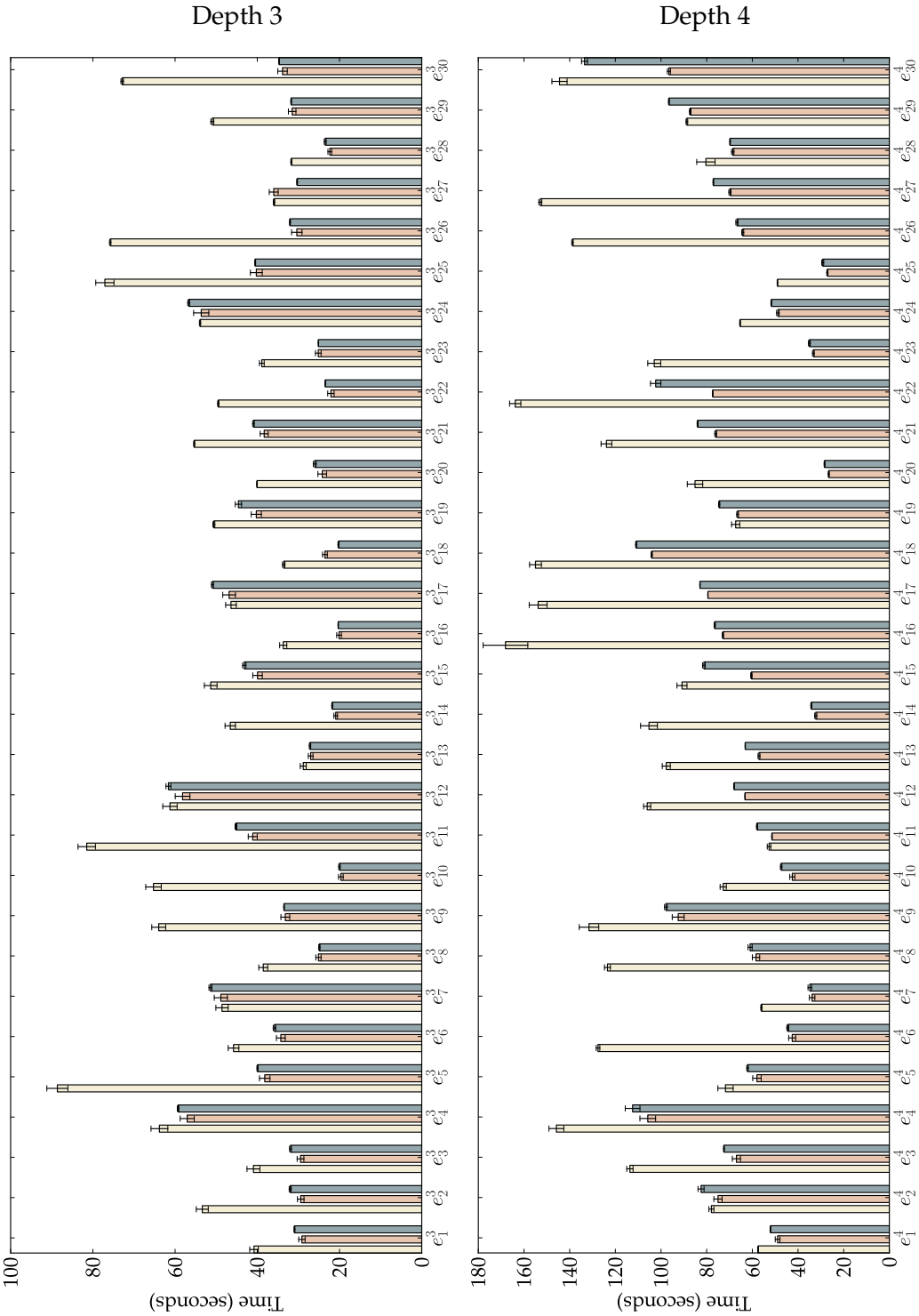


FIGURE 3.7: Depth 3 and 4 (random) benchmark results

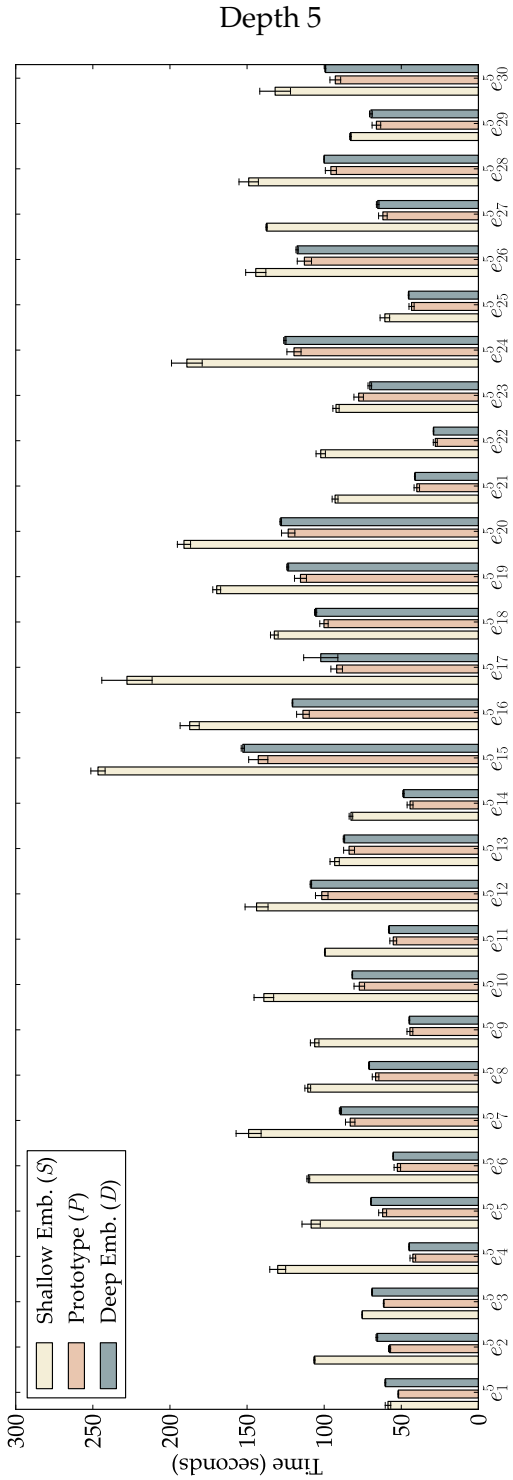


FIGURE 3.8: Depth 5 (random) benchmark results

staging was slower than shallow embedding (at maximum by 7.6%) and slower than deep embedding (at maximum by 7.9%). Again, these cases can most likely be attributed to the aforementioned boxing overhead. Overall (depth averages), implicit staging sped up execution compared to shallow embedding at minimum by 100.4% and at maximum by 257.5%. Compared to deep embedding, implicit staging sped up execution at minimum by 9.4% and at maximum by 29.7%. Figure 3.9 presents the aggregated results for the biased-expressions benchmark version.

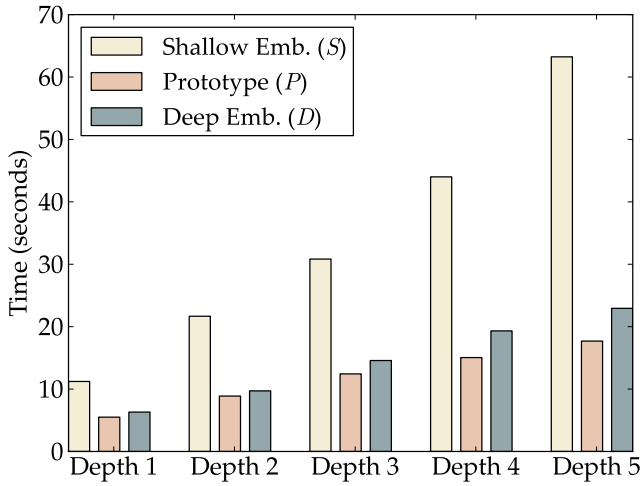


FIGURE 3.9: Aggregated (biased-expressions) benchmark results

Worst-case performance for the three EDSL implementations was also briefly examined by running a benchmark on the expression shown in Listing 3.8. It lends itself as a worst-case specimen due to the many scalar arguments and since no optimizations were implemented specifically for speeding up the addition of scaling matrices.

LISTING 3.8: Worst-case matrix expression

```
1  mul(
2    mul(
3      add(sca(5, 2.0), sca(5, 2.0)),
4      add(sca(5, 2.0), sca(5, 2.0))
5    ),
6    add(sca(5, 2.0), sca(5, 2.0))
7  )
```

Implicit staging was 103.2% slower than shallow embedding. However, deep embedding fared no better with a slowdown by 104.5%. This indicates that the interpretative overhead caused by expression-tree evaluation is significant. In order to further test this case, I implemented a lower-level expression compiler that simply generates Java code identical to the original expression instead of a tree of `Callable` instances. This implementation was only 1.1% slower (for our worst-case expression) than shallow embedding. Hence it seems advisable to move away from the compilation to `Callable`s for final versions of an implicitly staged EDSL implementation.

3.5.3 Usability

The usage of an EDSL that is implicitly staged with the prototype is very seamless, which means that users are not constantly reminded of the staged nature of EDSL expressions. After all, not only is the user interface implicit (i.e. equal to that of a shallow embedding), the framework also only really comes into effect when a user program is run, not when it is compiled. As described in Section 3.3, the load-time nature also positively affects maintenance and long-term evolution concerns.

As is common with static-staging approaches, some limited static context can be retained for use in debugging or avoiding redundant behavior. In the described prototype, bytecode positions of IR elements can be used but translating them to (source-code) line-number information has to be done manually (if detailed error messages are to be emitted during processing).

The aforementioned issues with token declaration can make it hard to really anticipate how far an EDSL extends, which may end up impairing both debugging as well as documentation in a similar fashion as syntactic macros. There are virtually no self-documentation facilities.

3.5.4 Comparison with Related Work

There are two recent developments in supporting EDSL implementations that are closely related to implicit staging and the load-time prototype framework: Yin-Yang [68] for Scala and *JIT macros* in the experimental Java VM of Project Lancet [90] implemented in Scala. Table 3.2 shows a comparison (along the lines of Section 2.4 in Chapter 2), whose details are expanded on in the following.

3.5.4.1 Yin-Yang

Yin-Yang is a preprocessing, generative, and compile-time-metaprogramming approach that is mainly motivated by the reduced ease-of-use of deep embeddings, in particular LMS-style EDSLs, which leak implementation details (such as `Rep` types, traits, etc.) and may lead to cryptic error messages (at compile time as well as at run time). In a nutshell this issue is addressed as follows:

1. The EDSL author defines a *direct* (i.e. shallow) embedding.
2. Yin-Yang is used to generate an LMS-style deep embedding from the shallow interface of this EDSL.
3. The author defines a macro (usually named after the EDSL) from which a Yin-Yang-provided transformation mechanism is invoked. It translates shallow EDSL programs to deep versions. Additionally, it also performs error checking and allows only constructs of the designated EDSL to be used.
4. At run time the deep-embedding code, translated via the macro at compile time, is processed and compiled. In principle, it should be possible to (make the EDSL macro expand to) cache this result to reduce overhead.

Fuzzy language boundaries are avoided by excluding the usage of foreign-language constructs altogether. Optimizations or other adjustments to the EDSL's

semantics can be freely implemented on the generated deep-embedding mirror version of the shallow interface. This freedom may lead to problems of non-compositional staging and opaque workflow, though easy to avoid when following the general patterns of LMS. However, in principle it is possible for EDSL authors to tamper with the internals of the deep embedding as well as the syntactic-macro code. This reduces the reliability ratings of the approach.

TABLE 3.2: Comparison of implementation approaches

		Yin-Yang	JIT Macros (Project Lancet)	Implicit-Staging Prototype
G	Global			
L	Local			
E	Expression			
D	Declaration			
I	EDSL Interface			
T	Typing			
$\smile$/▲	High/Good			
$\asymp$/♦	Fair			
$\frown$/▼	Low/Bad			
STAGED-EDSL DESIGN				
Scope		L	G	E
Delimitation		I	T	D
Staticity		$\smile$	$\smile$	$\smile$
Dynamicity		$\asymp$	$\smile$	$\asymp$
Transparency		$\smile$	$\asymp$	$\smile$
EVALUATIVE COMPARISON				
Reliability		◊	▽	◊
	Safeness	♦	▼	♦
	Uniformness	♦	♦	♦
Performance		△	△	△
	Overhead	♦	▲	▲
	Optimization	▲	▲	▲
	Adaptiveness	♦	▲	♦
Usability		△	▽	△
	Seamlessness	▲	♦	▲
	Maintenance	♦	♦	▲
	Debugging	▲	▼	♦
	Documentation	♦	▼	▼

Maintenance and evolution is possible. After all, the deep embedding can change as long as existing (shallow-to-deep) translations remain valid. However, feature improvements in the overall Yin-Yang framework will not automatically propagate to already compiled code. Debugging is well supported by automatic error messages as well as by retaining static-context information.

Documentation depends on how much of the target deep embedding is shown (at least on the interface level). Users may identify a generated Yin-Yang EDSL macro and then, assuming that tampering on generated code was not too heavy they can infer the target of its translation and from that may make guesses about the behavior (like in LMS). The EDSL-delimited (macro) blocks have implications on seamlessness. It is true that the approach is more seamless than LMS and deep embedding which is why it deserves a "good" rating. However, the named block does take away a little bit from this. Ultimately, this is a necessary design decision that limits staging to static blocks and aids language delimitation.

3.5.4.2 JIT Macros (Project Lancet)

Project Lancet is an ambitious endeavor to implement a Java VM with heavy metaprogramming support, based on and relying on the Graal JIT compiler [12]. The main goal of this approach is to put manual tools into the hands of programmers for precisely guiding the compilation of programs at run time.[7]

For instance, there are directives for inlining (e.g. to disable Project Lancet's default behavior of inlining non-recursive method calls), loop unrolling, and so-called *freezing*, which causes the supplied argument to be evaluated at JIT-compile time. These features are implemented by simple Scala-method syntax and become effective when JIT-compilation is triggered using `compile` (see Listing 3.9).

LISTING 3.9: Project Lancet JIT tools (selection of interfaces)

```
1 def compile[T,U](f: T => U): T => U
2
3 def freeze[T]       ( f: => T): T
4 def unroll[T]       (xs: Iterable[T]): Iterable[T]
5 ...
6 def inlineNever[T]  ( f: => T): T
7 def inlineAlways[T]( f: => T): T
8 def inlineNonRec[T]( f: => T): T
9 ...
```

In fact, these directives are implemented as so-called JIT macros, which are essentially registered callback hooks that are invoked with (arguments of) a form of decompiled-bytecode IR. A LMS-based IR (i.e. `Rep`, `Exp`, `Def`) is used for this since its internals heavily rely on LMS. In essence, the normal execution mode of the VM is implemented as a usual bytecode interpreter while JIT compilation is achieved via LMS-style multi-stage programming as follows:

1. "Decompilation" of bytecode is accomplished by using a staged version of the bytecode interpreter ("staged" here refers to the fact that it will perform staging when run), which performs LMS-style code generation (i.e. the *staged interpreter approach* [32, 103]). In other words, staged interpretation yields the original program represented in the LMS-based IR. In fact, this is essentially a variant of the so-called *first Futamura projection* [40, 41] known from partial

[7] While one may call this JIT compilation, it strays from the customary JIT-compilation concept present in current VMs, which is more transparent, automatic, and heuristical.

evaluation [65, Chapter 4, Section 4.3.3]: "[S]pecializing an interpreter with respect to a source program has the effect of compiling the source program", just that the target of "compilation" here is the LMS-style IR.[8] I will use the following set of equations to illustrate how to derive decompilation of a source program and final compilation to a target back end by turning a normal bytecode interpreter into a staged one:

$$
\begin{aligned}
[\![sourceProg]\!]_{\text{JVM}} &= [\![interp]\!]_{\text{Scala}} \quad [sourceProg] \\
&= [\![([\![spec]\!]_L \quad [interp, sourceProg])]\!]_{\text{Scala}} \\
&= [\![([\![stage^{\text{Scala}}_{\text{LMS}}(interp)]\!]_{\text{Scala}} \quad [sourceProg])]\!]_{\text{LMS}} \\
&= [\![([\![stagedInterp]\!]_{\text{Scala}} \quad [sourceProg])]\!]_{\text{LMS}} \\
&= [\![compile^{\text{LMS}}_{T}([\![stagedInterp]\!]_{\text{Scala}} \quad [sourceProg])]\!]_{T} \\
&= [\![targetProg]\!]_{T}
\end{aligned}
$$

2. Abstract interpretation and optimizations such as inlining could be performed on the resulting IR in a separate step. However, in Project Lancet this is actually interleaved with the first one.

3. The IR is compiled to a desired language and back end, which could for instance range from Java, Scala, and C++ to CUDA [13].

It is rather straightforward to see how this process can enable the implementation of callback hooks to customize processing. Listing 3.10 roughly presents what an optimizing JIT macro for the vector EDSL of Chapter 2 might look like. When the (shallow-interface) `Vec.plus` method is encountered during JIT compilation, the corresponding registered macro implementation method is called. This allows for the deep inspection of IR nodes and on-the-fly transformations.

LISTING 3.10: Vector EDSL JIT macros (rough sketch)

```
// Assume that this is declared in the right context (or scope)
object VecMacros {
  …
  def plus[T](self: Rep[Vec], v: Rep[Vec]): Rep[Vec] = (self, v) match {
    case (  Def(PlusN(vs1)),   Def(PlusN(vs2))) => PlusN(vs1 ++ vs2)
    …
    case (Def(Plus(v1, v2)),                v3) => PlusN(Seq(v1, v2, v3))
    case (               v1, Def(Plus(v2, v3))) => PlusN(Seq(v1, v2, v3))
    case (               v1,                v2) => Plus(v1, v2)
  }
  …
}
…
// Register macro callback hook
Lancet.install(classOf[Vec], VecMacros)
```

[8] Note that in LMS the default framework with built-in staged versions of Scala constructs essentially makes this IR a versatile representation of Scala code.

The freedom bestowed on EDSL authors to identify and modify code with JIT macros is not unlike the one granted by syntactic macros, which leads to a reduced safeness rating. However, while the inspection of host-language, i.e. general-purpose, code appears to be unrestricted, Scala's encapsulation mechanisms can be used to prevent cross-EDSL code inspection. Uniformness is not guaranteed either but is not much different from LMS and is less of a concern if standard LMS-based implementation practices are followed.

While overhead is certainly incurred during Project Lancet's explicit JIT compilation, all overhead when running the compiled code is eliminated. As a JVM implementation in full control of meta-level concerns, there is ample opportunity to exploit dynamic information of the run-time environment. Furthermore, specialization to various back ends that might be available at run time makes EDSL implementations with Project Lancet very adaptive.

On the usability side there are several disadvantages. While the staged nature of EDSL is well hidden from users, the framework of manually triggering and directing JIT compilation, which is required for JIT macros to come into effect, impairs seamlessness. Maintenance issues present themselves in the experimental nature of the custom JVM implementation and the Graal compiler. This may not be a general property of the approach but in the current discussion we have to consider that the approach (in particular its current manifestation) is more heavyweight than even, for instance, compiler plugins.

Although internally static information is heavily used, there does not appear to be any support for employing it in EDSL-program debugging. JIT macros are installed and configured dynamically and JIT compilation may fail often, as much of its workings are (not always safely) in the hands of users. The limited safeness and uniformness of JIT macros arguably further complicate the identification of bugs. The concern of documentation suffers from the same issues.

3.6 Discussion and Summary

By design, the idea of implicit staging is rather general and abstract with a broad vision and potential. Concrete explorations like the implemented prototype are necessary. However, it is nowhere near fully achieving the ideas outlined in Section 3.2.2. One limitation is that only compound expressions are extracted and no further relationship between separate, shared, or referenced ones is fully considered.

There actually were experimentations with allowing the inspection of variable accesses (*lac*) to offer some level of expression-external view. Take for instance the code snippet in Listing 3.11. During expression processing the prototype (experimentally) allows authors to inspect the accesses to `t` in order to optimize all multiplications referring to it, e.g. to perform appropriate scalar scaling instead of matrix multiplication.

However, it is challenging to devise an easy-to-use API that would also allow dealing with the removal of line 1: Whether it can be removed or not depends on the EDSL and whether it is actually inlined by all other expressions and at EDSL-external uses. This would necessitate the provision of more details (for instance as a graph of shared expression usage) to EDSL authors.

LISTING 3.11: Distant and dynamic EDSL-code composition

```
Mat t = sca(5, 3.0);
Mat u = mul(a, t);
if ( ··· ) {
  u = mul(u, t);
} else {
  u = add(u, t);
}
Mat m = add(u, u);
```

Note that with deep embedding the aforementioned case is not an issue as the default behavior is to inline, which shifts the responsibility to users. Furthermore, at the end of the given code snippet, `u` would be a dynamically staged program that depends on the actual flow of control. Of course, in a mostly static setting (as in implicit staging) it is not possible to predict which path will be taken. One way around this would be to explore all possible expressions and specialize the surrounding code accordingly. However, in the general case this is likely to cause intractable code explosion and in the presence of cyclic data flow (i.e. via loops) this fails. One idea to solve this is to somehow switch between dynamic construction (only) were static approximation fails. In fact, Chapter 4 will briefly explore something similar but in the opposite direction.

In the context of this thesis, the most severe limitation of the prototype lies elsewhere: Recall the previous discussion on reliability and usability. It is not enough to perform staging or code modification transparently. Hidden staging ought to be a convenience feature for ease-of-use, along the lines of how type inference allows the omission of type annotations. It should still follow a uniform, predictable pattern. So, any approach that allows a powerful API, e.g. for making expression-inlining decisions, also ought to document and expose the behavior to aid code comprehension when an EDSL user decides to investigate.

The black-box token declaration of the prototype presented in this chapter is an example for the described issue. There are no hard, consistent guarantees as to which EDSL, if at all, a method call belongs. The interface's ad-hoc nature does not allow for a reliable generation of (self-) documentation.

Finally, let us summarize the prototype's worthwhile parts:

- The explication of EDSLs leads to a higher-level treatment of performance-oriented language embedding, leading to rich library interfaces.
- Choosing load-time metaprogramming enables a pragmatic implementation, which can easily be made available to end users. In terms of maintenance it can be seen as a middle ground between compiler plugins and custom VMs.
- While limited, the interface of having a short number of entry points to EDSL-program processing (in fact only one, i.e. `ExpressionCompiler`, in the prototype) is simple and can improve the encapsulation of EDSL concerns.

The bottom line is that the presented prototype constitutes an improvement over traditional embedding techniques, yet is still an overly low-level extension.

3.7 Acknowledgments

The contents of this chapter are, in large part, based on and reproducing the research published in co-authorship with Prof. Chiba at the 28th European Conference on Object-Oriented Programming (ECOOP 2014) under the title "Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding" [93]. I was the first author of this paper.

CHAPTER 4

Almost First-Class Embedding

Chapter 3 laid out a very basic approach for indirectly but pragmatically extending Java for improved language-embedding support. However, the investigated load-time implicit-staging prototype ought to be considered merely an intermediate step toward providing a more refined linguistic abstraction.

The goal of the present chapter is to explore a pragmatic extension that raises the abstraction pattern of language embedding close to the level of a first-class language feature. Of course it would be possible to take the existing implicit-staging prototype and remove its discovered, obvious flaws. Unfortunately, doing so would not readily lead to a more encompassing design: For instance, while the API may be tweaked for more safety and documentation, it would still be limited to static staging. The existing implementation does little to improve the quality of dynamically staged EDSLs.

Why is dynamic staging of interest? Recall the first motivating examples in Chapter 1. Ideally, domain-specific programs such as the one presented in Listing 1.2 are to be directly expressed by users while at the same time allowing EDSL authors to implement their optimization. This requires dynamic staging to be hidden in a safe and comprehensible fashion. As discussed in Chapter 2, manual deep embedding does not provide reliability guarantees, so the program in Listing 1.3 comes with issues beyond merely its verbose explicit-staging nature.

The extension explored in this chapter reuses the ideas of explicating embedded languages and employing static analysis for extracting domain-specific computation. This is complemented by forcing EDSL authors to clearly and visibly mark the tokens of their EDSLs with membership information and other settings.

Secondly, the aspects of dynamic EDSL-program staging and their execution triggering are abstracted away, i.e. they are removed from the direct control of authors, in order to avoid the pitfalls of manual implementations. The author-defined EDSL-program processing occurs by way of indirection to a common interface for IR compilation, similar to `ExpressionToCallableCompiler`.

Like the implicit-staging prototype, the implementation of the present ideas relies on load-time metaprogramming for the sake of pragmatism. Thanks to using static analysis and code transformations, static-context information can be

retained and provided to EDSL authors. Moreover, the design allows for a potential straightforward subsumption of static staging and processing.

I will spend the largest part of this chapter on explaining the user-level workings of the extension followed by a behind-the-scenes discussion, evaluation, and several examples to clarify the usage and supplement the evaluation.

4.1 Support for Dynamically Staged EDSLs

Recall the workflow of using a deeply embedded DSL. It can be divided into the following run-time phases that sometimes overlap in practice:

1. ***(Dynamic) Staging***: IR construction and composition.
2. ***Processing***: Analysis, optimization, run-time compilation, etc.
3. ***Materialization***: Evaluation to a base-value result or side effect.

As discussed multiple times, in the interest of usability, it is desirable to make these phases transparent yet uniform. Implicit staging was designed to achieve this in a static setting. However, in order to provide an all-encompassing linguistic abstraction for language embedding it is important not to curtail the dynamic-staging nature. After all, not only are dynamic situations sometimes unavoidable, they are useful for specializing EDSL programs to run-time conditions.

This means that the above phases should still occur at run time. Additionally, similar to implicit staging, for the sake of reliability, the phases should be properly separated and only the processing phase should be exposed to EDSL authors. My approach is called *tame staging* and its overview is shown in Figure 4.1.

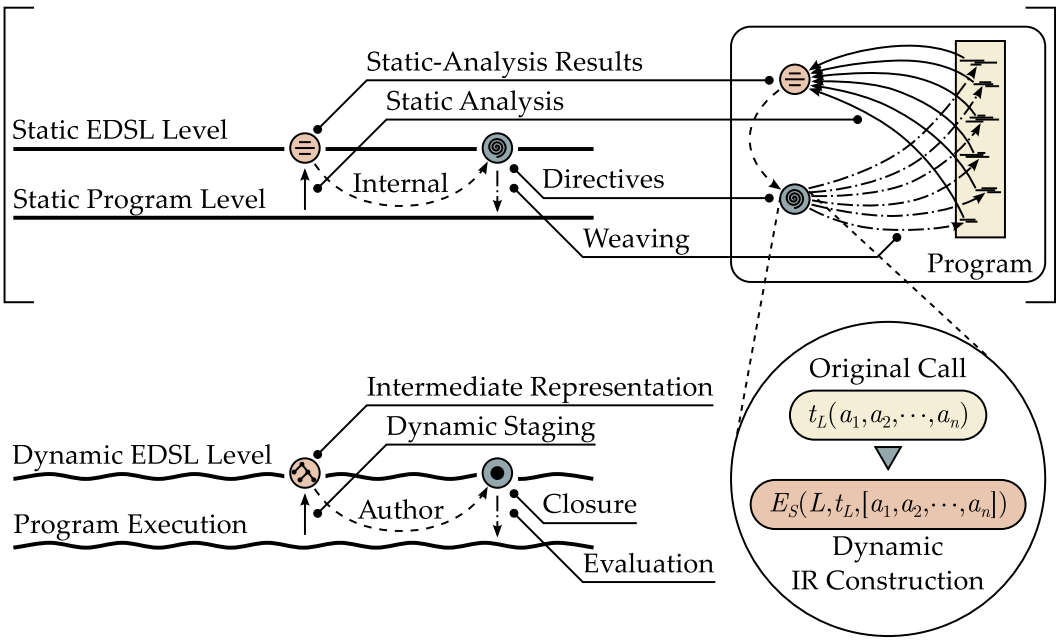


FIGURE 4.1: Tame-staging overview

The upper portion corresponds to the conceptual overview of implicit staging. However, the processing is internal and not exposed to EDSL authors. The result of the internal processing is a collection of transformation directives that, when applied to or *woven* into the program code (as in *aspect-oriented programming* [70]), essentially defer staging to run time. Although the lower right portion appears simple, this is generally more complicated than merely replacing expressions.

The approach was implemented in the form of a prototype framework for Java. However, there is little preventing the adoption of the ideas in other host languages.

4.1.1 The `@Stage` Annotation

Like implicit staging the present framework hinges on the ability to discern between staged and non-staged domain-specific parts of a program. By placing an `@Stage` annotation, whose declaration is shown in Listing 4.1, on a field or method declaration, EDSL authors can designate these to become tokens.

This may be considered another example of inheriting host-language infrastructure. Then again, annotation system like Java's are designed for adding such meta information. It is a more elegant and restricted way to designate language membership than the token-declaration interface of Chapter 3.

LISTING 4.1: `@Stage` annotation type declaration

```
1 @Documented
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target({ ElementType.METHOD, ElementType.FIELD })
4 public @interface Stage {
5   Class<? extends Language<?>> language();
6   boolean                      isStrict()           default false;
7   StaticInfo.Element[]         staticInfoElements() default    {};
8 }
```

An `@Staged` token is always associated with an explicit language, represented by a Java class marked[1] with the `Language` interface. This association is used for language-boundary checks and indirection to IR processing. This corresponds to the `EmbeddedLanguage` instances in the implicit-staging prototype. However, instead of (object) instances, the static type of a class known at compile-time is used. This means that a language must be present and named at the time tokens are declared using `@Stage`, which simplifies documentation. 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. As mentioned before, the framework accomplishes this by transforming user programs accordingly prior to execution.

Composing complex terms within the same embedded language is trivially understood: Argument terms simply become children of a newly constructed term

[1] No abstract methods are declared on the `Language` interface.

LISTING 4.2: Usage example

```
static void sink(Vec v0, Vec v1, Vec v2) { ··· }

static Vec example(int n, Vec a, Vec b, Vec c) {
  a = a.plus(b);

  if (n > 10) {
    a = a.plus(a);
  }
  sink(a, b, a);

  for (int i = 0; i < n; i++) {
    c = c.times(5 + i);
  }
  return c;
}
```

node in a tree-like structure (actually a rooted graph) just like in manual deep embedding. 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.

To illustrate the effect of using `@Staged` language constructs, consider the annotated vector EDSL presented in Figure 4.3, which omits the implementation details of Listing 2.6 in Chapter 2, together with the usage example in Listing 4.2. Note the shallow-embedding-like usage, which in the absence of the annotations (or the framework altogether) would cause standard, immediate method invocation.

However, with the framework and the annotations in place, the `example` method's code is transformed prior to its execution. Listing 4.4 presents a simplified, conceptual description of the transformations and the resulting IR construction that will occur at run time. The type placeholder E stands for the IR data type. Other notation used here is explained as follows:

- 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).
- Double brackets ($[\![\cdots]\!]$) here indicate materialization triggering. This notation was chosen for its frequent use for providing meaning to syntax [65, 78].

LISTING 4.3: Annotated (i.e. `@Staged`) `Vec` methods

```
public final class Vec {
  ···
  @Stage(language = VecL.class)
  public Vec plus(Vec v)      { ··· /* ignored */ }

  @Stage(language = VecL.class)
  public Vec times(double s) { ··· /* ignored */ }
}
```

LISTING 4.4: Transformation example

```
1  static void sink(Vec v0, Vec v1, Vec v2) { ··· }
2
3  static Vec example(int n, Vec a, Vec b, Vec c) {
4  ○ a = a.plus(b);
   ● ⇝ E t_a = E_S(VecL,Vec.plus,[E_V(a),E_V(b)]);
5
6    if (n > 10) {
7    ○ a = a.plus(a);
     ● ⇝ t_a = E_S(VecL,Vec.plus,[t_a,t_a]);
8    }
9  ○ sink(a, b, a);
   ● ⇝ sink([[t_a]], b, [[t_a]]);
10 ○
   ● ⇝ E t_c = E_V(c);
11   for (int i = 0; i < n; i++) {
12   ○ c = c.times(5 + i);
     ● ⇝ t_c = E_S(VecL,Vec.times,[t_c,E_V(5 + i)]);
13   }
14 ○ return c;
   ● ⇝ return [[t_c]];
15 }
```

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. This is different from the implicit-staging prototype that essentially treated variable assignment as external consumption or usage of a staged term. Also note that the current framework was written from scratch and uses a different `Expression`-IR implementation despite sharing the same type name.

Arguably, especially in light of the existence of potential side effects in the general-purpose host-language program, this kind of hidden dynamic staging would seem like a recipe for disaster. The following 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.

- Referencing terms of default-annotated tokens (within the same language) is uniform, i.e. causes inlining. Different behavior might be desirable. Although this is currently not implemented, conceptually there is no reason why a future version could not offer more options within the `@Stage` annotation to configure (and document) this behavior.

- The scope of staging or rather of staged-IR transfer is limited to the body of the method (for now). 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 escape the method's local scope. I will introduce *globally carried* terms in Section 4.1.4.

4.1.2 Materialization Triggers

For locally carried terms, materialization is triggered lazily when encountering a language boundary. At the call `sink(a, b, a)` (line 9) in Listing 4.4, materialization of the first occurrence of `a`'s corresponding term $t_{\mathtt{a}}$ is triggered before the second one. Similar to *thunks* in other on-demand evaluation schemes, in the present 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 consumed (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. Note that there is no need for the framework (or an EDSL author) to specially support an explicit triggering construct for locally carried terms. Any non-`@Staged` identity function can fulfill this role, for instance:

```
public static <T> T materialize(T t) { return t; }
```

Above I 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* (this is further discussed in Section 4.1.5) 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.

LISTING 4.5: Dynamic boundary-decision example ($L_1 \neq L_2$)

```
1  @Stage(language = L1)
2  public static int dec(int i) { return i - 1 }
3
4  @Stage(language = L2)
5  public static int inc(int i) { return i + 1; }
6
7  static void example(int i) {
8    if (i > 0) {
9      i = dec(i);
10   } else {
11     i = inc(i);
12   }
13
14   return inc(i);
15 }
```

In general, the language of a staged-EDSL-term argument cannot always be determined statically. Consequently, the decision on boundary-based materialization triggering is made dynamically. Listing 4.5 illustrates this on a simple, fictitious example: There is no way of statically telling the language of the argument term in line 14. If $t_{\mathtt{i}}$ comes from line 9, it will be materialized and lifted, if it comes from

line 11 it will just be incorporated normally. This decision will have to be made by checking whether the argument term for `inc` in line 14 is a member of L_2 or not.

4.1.3 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[2] similar to the one sketched in Figure 4.6. However, there is quite a bit of scaffolding whose understanding is necessary for this code example to make sense.

LISTING 4.6: Trivial compiler skeleton

```
class VecLCompiler implements Expression.Visitor {
  private final Binder binder;
  private ObjectClosure<Vec> closure;

  VecLCompiler(Environment.Binder binder) { this.binder = binder; }

  public ObjectClosure<Vec> getClosure() { return closure; }

  public void visit(MethodInvocation staged) {
    staged.getArgument(0).accept(this);
    ObjectClosure<Vec> a0 = closure;

    switch (staged.getMember().getName()) {
      case "plus": {
        staged.getArgument(1).accept(this);
        ObjectClosure<Vec> a1 = closure;

        closure = env -> {
          Vec v0 = a0.evaluate(env);
          Vec v1 = a1.evaluate(env);
          double[] ds = new double[v0.elements.length];

          for (int i = 0; i < ds.length; i++) {
            ds[i] = v0.elements[i] + v1.elements[i];
          }

          return new Vec(ds);
        }

        break;
      }
      case "times": ⋯
    }
  }
  ⋯
  public void visit(ObjectValue value) {
    closure = value.bind(binder);
  }
}
```

[2]Just like in the implicit-staging prototype.

Cached Processing. 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. The `Closure`s of the tame-staging framework correspond to the `Callable`s of the implicit-staging prototype.

This indirection allows the framework 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:

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

Its shape ranges over all possible value instantiations for x, y, and z. When the materialization of a staged-EDSL program is triggered, first a cache is probed for an existing mapping from 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 depends on the expected type of the materialized value and follows a naming convention and type signatures as follows:

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

If an author can exclude the case of some type ever being materialized in their language there is no need to provide a processing method for it.

EDSL authors implement these methods as needed at their discretion. However, they need to mind 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 will be the case when the shape of an EDSL program is known statically. With this we see one way in which the framework subsumes aspects implicit staging was also concerned with: Exploiting static knowledge to reduce overhead.

Value-Access Indirection. To accomplish the separation of expression-DAG shape and concrete instances, lifted-value nodes cannot be inspected for their actual contents (without at the same time disabling caching) unless they are constants. Instead, the supplied `Binder` instance is used to create value-access closures from lifted-value nodes (see Listing 4.6, line 37).

Caching and reusing closures requires injecting the actual input-value instances as arguments at materialization time. The methods (`evaluate`) of the closure interfaces take a single `Environment`-typed argument housing the input values found in the expression DAG to be materialized (cf. Section 3.4.3.2 of Chapter 3):

```
double evaluate(Environment environment)
⋮
```

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. However, unlike for the implicit-staging prototype, where such a backing was used, one now has to account for the dynamic nature of the present framework: Extracting and flattening all values every time is costly. I found that making the access closures simply graph-walk to value nodes turns out to be a faster solution in most cases. With this design the environment is simply housing the dynamically staged "input-value" expression DAG. This is an implementation detail worthy of future investigation.

Example. Finally, the reader may revisit Listing 4.6, showing a part of a trivial compiler for the vector EDSL of Listing 4.3 that does not implement optimizations. Listing 4.7 shows the definition of the explicated vector EDSL.

LISTING 4.7: Language class (`VecL`)

```
class VecL implements Language<VecL> {
  public static ObjectClosure makeObjectClosure(
                  Expression.Staged staged,
                  Environment.Binder binder,
                  boolean permCached) {
    VecLCompiler compiler = new VecLCompiler(binder);
    staged.accept(compiler);
    return compiler.getClosure();
  }
}
```

Note that while this shows translation using the visitor pattern, the framework is not concerned with how traversal occurs in detail. For instance, I also created a support package for implementations in Scala using *pattern matching*.

LISTING 4.8: Language class (`VecL`) for Scala compiler

```
class VecL implements Language<VecL> {
  public static ObjectClosure makeObjectClosure(
                  Expression.Staged staged,
                  Environment.Binder binder,
                  boolean permCached) {
    VecLScalaCompiler compiler = new VecLScalaCompiler(binder);
    return (ObjectClosure) compiler.compile(staged);
  }
}
```

Listing 4.9 shows a full Scala implementation of an optimizing compiler that fuses additions and scalar multiplications (Listing 4.8 shows the adapted `VecL` class). The `@Supress` annotation is used to prevent staging so that the shallow-embedding `plus` and `times` methods can be reused in lines 18 and 25 respectively. I will discuss this feature in more detail in Section 4.1.6.

LISTING 4.9: Optimizing compiler in Scala

```
class VecLScalaCompiler(implicit binder: Binder) {
  case class PlusN(vecs: Array[ObjectClosure[Vec]])
             extends ObjectClosure[Vec] {
    override def evaluate(env: Environment): Vec = {
      var v = vecs(0).evaluate(env)
      val ds = util.Arrays.copyOf(v.elements, v.elements.length)
      for (i <- 1 to vecs.length - 1) {
        v = vecs(i).evaluate(env)
        for (j <- 0 to ds.length - 1) { ds(j) = ds(j) + v.elements(j) }
      }
      new Vec(ds)
    }
  }
  class Plus(left: ObjectClosure[Vec], right: ObjectClosure[Vec])
        extends PlusN(Array(left, right)) {
    @Suppress(languages = Array(classOf[VecL]))
    override def evaluate(env: Environment) = {
      left.evaluate(env).plus(right.evaluate(env))
    }
  }
  case class Times(vec: ObjectClosure[Vec], s: DoubleClosure)
             extends ObjectClosure[Vec] {
    @Suppress(languages = Array(classOf[VecL]))
    override def evaluate(env: Environment) = {
      vec.evaluate(env).times(s.evaluate(env))
    }
  }

  def compile[T <: Closure[_]](expression: Expression): T = {
    expression match {
      case MethodInvocation(CtMethod("plus"), _, List(leftE, rightE)) =>
        (compile[ObjectClosure[Vec]](leftE),
         compile[ObjectClosure[Vec]](rightE)) match {
          case (PlusN(lVecs), PlusN(rVecs)) => PlusN(lVecs ++ rVecs)
          case ( PlusN(vecs),       rightC) => PlusN(vecs :+ rightC)
          case (       leftC,  PlusN(vecs)) => PlusN(leftC +: vecs)
          case (       leftC,       rightC) => Plus(leftC, rightC)
        }

      case MethodInvocation(CtMethod("times"), _, List(vecE, sE)) =>
        (compile[ObjectClosure[Vec]](vecE),
         compile[DoubleClosure](sE)) match {
          case (Times(vec, s), sC) => Times(vec, new DoubleClosure {
            override def evaluate(env: Environment): Double = {
              sC.evaluate(env) * s.evaluate(env)
            }
          })
          case (         vecC, sC) => Times(vecC, sC)
        }

      case MethodInvocation(_, _, List(e) => compile(e)
      case ObjectValue(oC) => oC
      case DoubleValue(dC) => dC
    }
  }.asInstanceOf[T]
}
```

4.1.4 Global Carrying

The 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 explicit staging and materialization can be good for users. Yet, sometimes a more obvious and *global* style of staging as traditionally seen with deep embedding is in fact desirable. For instance, this is arguably better suited for EDSLs that deal with mutable input, since users have to decide precisely where to trigger EDSL-program execution. Furthermore, in some embedded languages, not every fragment bears meaning on its own. Local carrying is problematic for these cases since every single term may hit a language boundary and cause its materialization.

Hence, it is desirable for a linguistic-abstraction feature for dynamically staged EDSLs to also provide support for this style. In the following I will describe how a small addition to the framework described so far accomplishes this by permitting terms to be *globally carried*. Figure 4.10 shows global carrying in action using the explicit, deeply embedded version of the vector DSL. 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.

LISTING 4.10: Global carrying example

```
1  static VecE times2(VecE vE) { return vE.times(2.0); }
2
3  static Vec example(int n, Vec a, Vec b, Vec c) {
4    VecE rE = a.toVecE().plus(b);
5    Vec r = times2(rE).toVec();
6    if (n > 5) {
7      return c;
8    }
9    return r;
10 }
```

Recall that with traditional deep embedding EDSL authors define their own types (like `VecE`) and data structures for representing abstract syntax. In the present framework EDSL authors can instead simply declare a type extending the `GlobalCarrier` class. An instance of a *global carrier* can have two meta states:

- An actual value of the data type as defined at the author's discretion.
- Abstract syntax, i.e. it literally *carries* an expression DAG.

Hybrid states are possible but usually not recommended.

Defining an `@Stage`-annotated token with (return) type `GlobalCarrier`, or preferably a subtype thereof, suffices to signal that the returned instance shall carry the constructed term. The code in Listing 4.11 completely defines `VecE` and adds relevant annotations. When an `@Stage`-annotated construct (or rather its term construction) receives a carrier-typed argument this will first be checked for carrying a staged term and if so move on to language-boundary checks. For a foreign-language globally carried term its carrier will be lifted as a value instead of contributing to the complex-term composition.

LISTING 4.11: `VecE` as a global carrier

```
public final class Vec {
  …
  @Stage(language = VecEL.class)
  public VecE toVecE()        { … /* ignored */ }
}

public final class VecE extends GlobalCarrier {
  @Stage(language = VecEL.class)
  public VecE plus(VecE vE)   { … /* ignored */ }

  @Stage(language = VecEL.class)
  public VecE times(double s) { … /* ignored */ }

  @Stage(language = VecEL.class)
  public Vec toVec()          { … /* ignored */ }
}
```

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 token designated to be an explicit trigger (as usual in traditional deep embedding).
- By transitioning to a locally carried token, which may lead to subsequent implicit materialization.

Let us interpret Figure 4.10 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 9 if it is reached during execution. Note that thanks to the way the methods are named and structured in the EDSL, one could simply reuse the compiler of Figure 4.6.

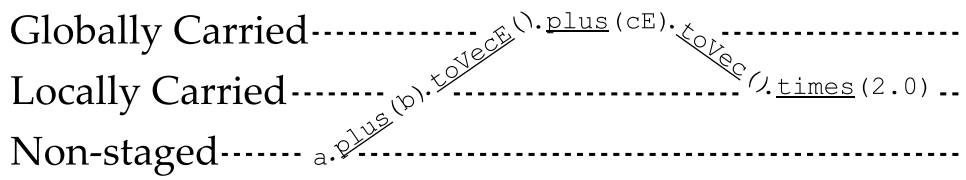


FIGURE 4.2: Carrying-level transitioning

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 carrying (i.e. methods on `Vec`) with global carrying, as shown in Figure 4.2. 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 this may seem contrived, a use case for such an arrangement may arise in EDSLs where some tokens require explicit handling by users while others do not.

4.1.5 Language-Boundary Customization

For the sake of modularity it is desirable to make EDSL syntax composable, i.e. reusing the tokens of another language. This has not been a big focus for the tame-staging framework. However, at the very least basic support for it requires a means for changing the default boundary behavior of terms only accepting language constructs of the same language.

To this end the framework provides the `@Accept` annotation, which is to be placed on the individual parameters of `@Staged` constructs for which the default (self-) acceptance behavior is to be overridden. Its only element (`languages`) is an array of language classes that will be interpreted as the set of languages accepted at the given parameter. Recall the example of Listing 4.5. We could annotate the `inc` token to also accept L_1 as shown in Listing 4.12 (line 2). This would mean that L_2's compiler also needs to handle `dec` tokens.

LISTING 4.12: `@Accept`ing

```
1 @Stage(language = L2)
2 public static int inc(@Accept(languages = { L1, L2 }) int i) { ··· }
3
4 @Stage(language = L2)
5 public static int inc(@Accept(languages = { L1 }) int i) { ··· }
6
7 @Stage(language = L2)
8 public static int inc(@Accept(languages = {}) int i) { ··· }
```

It is also possible to make `inc` not accept its own language (line 5) or not accept any language at all (line 8), i.e. always force materialization of the argument. The latter is especially useful for cutting off staging early: Imagine the familiar vector DSL had a token that returns a **`double`**-typed value, e.g. to retrieve an element. An author might deliberately not want to handle the case where this is fed into `times`'s scalar argument.

Currently, it is not possible to denote acceptance of *any* language. There might be use cases for it and this feature ought to be considered in future work. However, for the time being it seems too welcoming to reliability and documentation issues. For instance, an EDSL author might not want their tokens to be accepted by others. This is possible and will be introduced shortly. For now, consider what would happen if a token signals that it accepts all languages. In fact it would only mean to accept "all languages that do not deny it". Without this feature it is a lot easier to clearly see exactly which languages are accepted where.

The system of languages and their boundaries (together with their customizations) may be viewed as a sort of tacked on, mixed dynamic and static type system: If languages correspond to types then their tokens roughly correspond to values. However, languages exist on a separate level, parallel to the traditional types.

4.1.6 Suppression of Staging Behavior

In most of the examples in this chapter the method bodies of `@Staged` EDSL tokens were omitted. This was done because their implementation was not relevant (i.e.

ignored): Using such a token is not linked to its original implementation anymore. As in implicit staging it is reinterpreted.

However, I 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 or reusing existing implementations as was done in Listing 4.9. 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).

Without the suppression of staging in Listing 4.9, the implementing method bodies would yet again be transformed, in turn leading to staging yet again. The original implementation would never be reached.

4.1.7 Visibility Control

An EDSL author might not want users to suppress staging of their language. Likewise, allowing just anyone (i.e. other authors) to add tokens to an existing language with the `@Stage` annotation is dangerous. After all, the author of the EDSL should not by default be forced to consider additional language constructs (added after the fact) when building their compiler. Similar issues apply to `@Accept` as mentioned in Section 4.1.5.

LISTING 4.13: `@Configure` annotation type declaration

```
1 @Documented
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target({ ElementType.TYPE })
4 public @interface Configure {
5   boolean hasRestrictedAcceptAccessibility()   default false;
6   boolean hasRestrictedStageAccessibility()    default false;
7   boolean hasRestrictedSuppressAccessibility() default false;
8 }
```

A simple solution is to follow the EDSL tradition of inheriting host-language infrastructure by relying on Java's visibility control: Setting the language class non-public solves the issue. However, sometimes this might be too drastic, for instance when suppression should be allowed "publicly" while prohibiting the usage of the other annotations. To this end, yet another annotation, called `@Configure` (see Listing 4.13), was added, which 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 annotation usage outside of the Java package that contains the `@Configure`d language class.

4.1.8 Static `@Stage` Inheritance

The framework's interpretation of a construct as either 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 permitted. 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. This applies to fields as well for consistency. However, currently it is allowed to change languages or `@Accept` options as it might prove useful for specialization on a lower level of a class hierarchy. Future investigations ought to determine whether reliability concerns should outweigh this freedom. It might also be worthwhile to consider providing a setting on `@Stage` for limiting inheritance.

Let us look at an example that showcases how `@Stage` inheritance can be used to provide optimizations on a family of concrete constructs. Consider Haskell's **`Functor`** type class defined as follows [66, Chapter 6]:

```
class Functor f where fmap :: (a -> b) -> f a -> f b
```

Its instances are all to obey the following identities:

```
fmap       id  =  id
fmap  (f . g)  =  (fmap f) . (fmap g)
```

These open up the door to optimizing nested applications of `fmap`, for instance by rewrite rules (cf. Section 2.2.2 in Chapter 2).

LISTING 4.14: Abstract `Functor` and implementation example

```
public abstract class Functor<A> {
  @Stage(language = FunctorL.class)
  public abstract <B> Functor<B> fmap(
      @Accept(languages = {}) Function<? super A, ? extends B> function);
}

public class FunctorList<T> extends Functor<T> {
  private final ArrayList<T> list;

  private FunctorList(ArrayList<T> list) { this.list = list; }

  public FunctorList(List<T> list) { this.list = new ArrayList<>(list); }

  public <U> FunctorList<B> fmap(
      Function<? super T, ? extends U> function) {
    return new FunctorList<>(
      list.parallelStream()
          .map(function)
          .collect(Collectors.toCollection(ArrayList::new))
    );
  }
}
```

In Java we could emulate this type class with an abstract class as shown in Listing 4.14, which also shows an example class (`FunctorList`) extending it. Using the present framework it is possible to implement (dynamic) rewritings for all subclasses of `Functor`. The crucial idea is that the `fmap` method is associated with a staged-EDSL class, here named `FunctorL`.

A compiler for `FunctorL` can be written that for all `Functor` implementations (i.e. subclasses) turns a method chain of `fmap` calls into a closure that performs a single `fmap` call with the composition of all the argument functions. Note that we have to make use of `@Suppress(languages = FunctorL.class)` in that compiler implementation (i.e. for the generated closure) because we need to be able to actually use the original, late-bound implementation of `fmap`.

4.1.9 Static Information

The 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, there only is support for collecting two types of information:

- `StaticInfo.Element.ORIGIN`, the origin, i.e. method, line number, and bytecode position of a constructed term.[3]
- `StaticInfo.Element.INFERRED_TYPES`, the inferred types of a token's arguments and (return) type.

The actual information can be retrieved from `Expression.Staged` (subtype) instances as `Optional<StaticInfo>` values.

One might wonder why the framework does not simply collect all available information all the time. One reason is that obviously it costs memory and time to add additional information to every constructed term. A more pressing reason is that static information needs to be considered part of an expression DAG's shape. After all, an EDSL author can inspect it and thus it may affect the cached result of expression-DAG processing. Now, if every EDSL term were to contain static information, the amount of redundant processing would be bound to increase since fewer expression graphs would end up being isomorphic to each other.

For an example of using `staticInfoElements`, recall the Mini EDSL of Section 2.1.2 and Section 2.3.3 in Chapter 2. Listing 4.15 shows the full definition of its tokens using the tame-staging framework. The language class `MiniL` is simple and omitted here together with its compiler. What is important here is that before compilation we can perform a simple definite-assignment analysis directly on the abstract syntax. Its important bits are shown in Listing 4.16. It is very crude and simply throws a run-time exception when a non-assigned variable is read.

Alas, when the faulty factorial program in Listing 4.17 is run, the analyzer will complain with the error message: `Variable not initialized near:` `ORIGIN(⋯.factorial(int),` pos, $start + 5$), where pos stands for bytecode position and $start$ denotes the first line of the method. This could be made more accurate if more tokens were set to collect static information.

[3] It serves a similar purpose as Scala-Virtualized's `SourceContext` [89].

LISTING 4.15: Mini tokens

```
public final class Mini { Mini() {}
  public abstract static class BoolE extends GlobalCarrier { BoolE() {} }
  public abstract static class BoolV extends BoolE         { BoolV() {} }
  public abstract static class IntE  extends GlobalCarrier { IntE() {} }
  public abstract static class IntV  extends IntE          { IntV() {} }
  public abstract static class Stmt  extends GlobalCarrier { Stmt() {}
    @Stage(language = MiniL.class, staticInfoElements = ORIGIN)
    public Stmt then(Stmt v) { ··· }
    @Stage(language = MiniL.class, isStrict = true,
           staticInfoElements = ORIGIN)
    public int intRun(IntV e) { ··· }
    @Stage(language = MiniL.class, isStrict = true,
           staticInfoElements = Element.ORIGIN)
    public boolean boolRun(BoolV e) { ··· }
  }

  @Stage(language = MiniL.class)
  public static IntE add(IntE a, IntE b) { ··· }
  @Stage(language = MiniL.class)
  public static IntE mul(IntE a, IntE b) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE eq(IntE a, IntE b) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE leq(IntE a, IntE b) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE and(BoolE a, BoolE b) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE or(BoolE a, BoolE b) { ··· }

  @Stage(language = MiniL.class)
  public static IntE neg(IntE a) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE not(BoolE a) { ··· }

  @Stage(language = MiniL.class, staticInfoElements = ORIGIN)
  public static IntV intVar(String name) { ··· }
  @Stage(language = MiniL.class, staticInfoElements = ORIGIN)
  public static BoolV boolVar(String name) { ··· }

  @Stage(language = MiniL.class)
  public static Stmt intAssign(IntV v, IntE e) { ··· }
  @Stage(language = MiniL.class)
  public static Stmt boolAssign(BoolV v, BoolE e) { ··· }

  @Stage(language = MiniL.class)
  public static IntE intLit(@Accept(languages = {}) int a) { ··· }
  @Stage(language = MiniL.class)
  public static BoolE boolLit(@Accept(languages = {}) boolean a) { ··· }

  @Stage(language = MiniL.class)
  public static Stmt whileDo(BoolE test, Stmt s) { ··· }
}
```

LISTING 4.16: Mini analyzer

```
class MiniLAnalyzer implements Expression.Visitor {
  private final Environment.Binder binder;
  private final HashSet<Expression> initVars = new HashSet<>();
  private StaticInfo lastStaticInfo;

  MiniLAnalyzer(Environment.Binder binder) { this.binder = binder; }

  public void visit(Expression.MethodInvocation staged) {
    CtMethod m = staged.getMember();
    switch (m.getName()) {
      case "add": case "mul": case "eq": case "leq":
      case "and": case "or": {
        staged.getArgument(0).accept(this);
        staged.getArgument(1).accept(this);
        break;
      }
      case "neg": case "not": {
        staged.getArgument(0).accept(this);
        break;
      }
      case "intVar": case "boolVar": {
        if (!initVars.contains(staged)) {
          Expression.ObjectValue<String> name =
            (Expression.ObjectValue<String>) staged.getArgument(0);
          throw new RuntimeException("Variable not initialized near: "
            + lastStaticInfo.getOrigin().get());
        }
        break;
      }
      case "intLit": case "boolLit": {
        staged.getArgument(0).accept(this);
        break;
      }
      case "intAssign": case "boolAssign": {
        staged.getArgument(1).accept(this);
        initVars.add(staged.getArgument(0));
        break;
      }
      case "whileDo": {
        HashSet<Expression> oldInitVars = new HashSet<>(initVars);
        staged.getArgument(0).accept(this);
        staged.getArgument(1).accept(this);
        initVars.retainAll(oldInitVars);
        break;
      }
      case "then": case "intRun": case "boolRun": {
        staged.getArgument(0).accept(this);
        lastStaticInfo = staged.getStaticInfo().get();
        staged.getArgument(1).accept(this);
        break;
      }
    }
  }
  ...
}
```

LISTING 4.17: Faulty Mini factorial

```
static int factorial(int x) {
  IntV n = intVar("n");
  IntV a = intVar("a");

  return   intAssign(n, intLit(x))
    .then( whileDo(leq(intLit(1), n),
             intAssign(a, mul(a, n))
      .then( intAssign(n, add(n, intLit(-1))) ))
    ).intRun(a);
}
```

4.2 Behind the Scenes

As introduced in Section 4.1, behind the scenes of the tame-staging framework there is an on-the-fly code-transformation mechanism inspired by implicit staging and its simple proof-of-concept implementation. Again, through the use of load-time metaprogramming and by reusing host-language infrastructure, now including annotations, EDSL-abstraction becomes an *almost* first-class feature.

Ideally, it would be supported natively by the compiler and runtime (e.g. JVM) as an *actual* first-class feature. However, an entirely new, heavyweight language just for exploring and prototyping this framework is is not pragmatic due to development and maintenance concerns as well as the initial hurdle (for users) of switching to a novel language implementation.

4.2.1 The Case for Load Time

The advantages of using load time have already been discussed in Section 3.3 of Chapter 3. Let us here make the case for it again with special regard to the issues at hand in the tame-staging framework's implementation:

- The framework implementation needs to hide certain methods during compile time but establish that they are public after user-code transformation (i.e. when the transformed code is executed).
- Classes need to be adapted 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.
- By transforming bytecode as it is loaded, processing can be delayed as long as possible. This fits very nicely into the rest of the design rationale, e.g. on-demand materialization of locally carried terms.

4.2.2 System Overview

When a new class is loaded into the system its methods are quickly and superficially scanned for containing usages of `@Staged` constructs. Only if present the system performs several proper static data-flow analyses on bytecode in the following order:

1. ***Type-Inference Analysis*:** Yields type approximations.
2. ***Value-Flow Analysis*:** Relates the creations and consumptions of values.
3. ***Constant Analysis*:** Marks values as constant (or not).
4. ***Stage Analysis*:** Builds a *stage graph* (see Section 4.2.4) from `@Staged` tokens.
5. ***Weave Analysis*:** Determines necessary directives for bytecode adaptations.

The first three analyses are independent of language-embedding concerns but relied on by the latter two. All of them are iterative forward-flow data-flow analyses based on the same basic principals as the one used in the implicit-staging prototype. However, here phases are separated more clearly instead of having everything combined into one process. Moreover, they are largely written from scratch using a slightly different traversal order.

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.

4.2.3 Constant Analysis

Constant analysis is interesting as it can actively exploits the load-time aspect of the implementation: Accesses to **`final static`** fields of classes are considered constant (at compile time we do not always know yet whether a field access will be constant or not), and if the classes in question have already been loaded and initialized, the available actual values can be used to improve the constant analysis.

LISTING 4.18: Constant analysis

```
1  public class A {
2    public static int f = 3;
3
4    public static void main(String[] args) {
5      out.println(B.f);
6      C.m(args.length);
7    }
8  }
9
10 public class B {
11   public static final int f = A.f + 2;
12 }
13
14 public class C {
15   public static void m(int i) {
16     int x = 5;
17     if (i > 2) {
18       x = B.f;
19     }
20     out.println(x);
21   }
22 }
```

Let us look at the example in Listing 4.18 with method `A.main` as the program's entry point. By the time class `C` will be loaded by the JVM, class `B` will have been initialized and its static field `B.f` constantly set to `5`. For the sake of argument assume that the constant analysis is run on method `C.m`. It could detect that at line 20 where `x` is consumed, its value will be `5`. Of course this is only the case here but might change depending on the situation at load time. This is why at compile time this analysis is not possible. For instance, due to dynamic linking there is no guarantee that all classes use the same version as where seen during compilation.

Note that this analysis may become problematic in a setting where classes can be reloaded dynamically. Supporting this would require invalidation of previous analyses and transformations and subsequent recursive reloading. Run-time reflection may also affect this analysis negatively. It might be necessary to limit this analysis or make it configurable in the future. In case of a native language feature, the concerns above could be addressed on the VM level.

4.2.4 The Stage Graph

Together mainly with the results of value-flow analysis and annotation lookup, the stage analysis yields a graph that represents both the data and control flow between the EDSL tokens contained in a method and also external value generations (or sources) and consumptions (or uses).

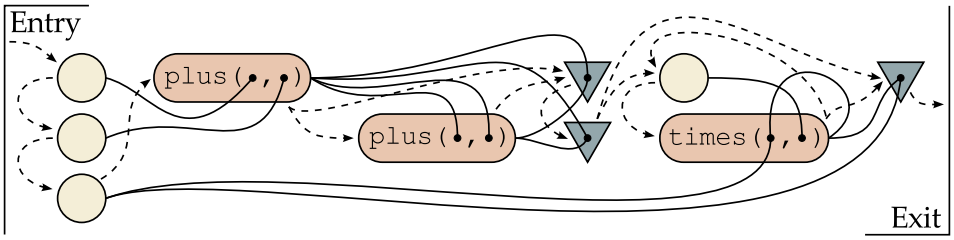


FIGURE 4.3: Stage graph for Figure 4.2

Figure 4.3 shows an example graph for the code in Listing 4.2 of Section 4.1.1. 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. This graph can be used to analyze interactions among only those primitives relevant for the contained staged programs (with control-flow dependent shapes) and ignore everything else that goes on externally.

For instance, one analysis on the stage graph determines which parts will never take different shapes and can thus be permanently cached (or potentially preprocessed). Simplified, these are the terms stemming from constructs that have no dynamic dependencies and are strict or at least used once externally. In the graph of Figure 4.3 this applies to the `plus` nodes but not the `times` one.

Note that this comes close to a rich (static) IR as envisioned in Section 3.2.2 of Chapter 3 though it is only used framework-internally. It is still too crude and does not isolate separate EDSLs, which is only achieved by the framework's further processing. The actual graph contains more detailed information on the nodes than the simplified depiction above.

4.2.5 Token-Representation Generation

Every `@Staged` token is represented by its own class that is generated on the fly before weaving. Figure 4.4 shows the `Expression` hierarchy in which token representations are made to extend one of the three `Expression.Staged` classes.[4]

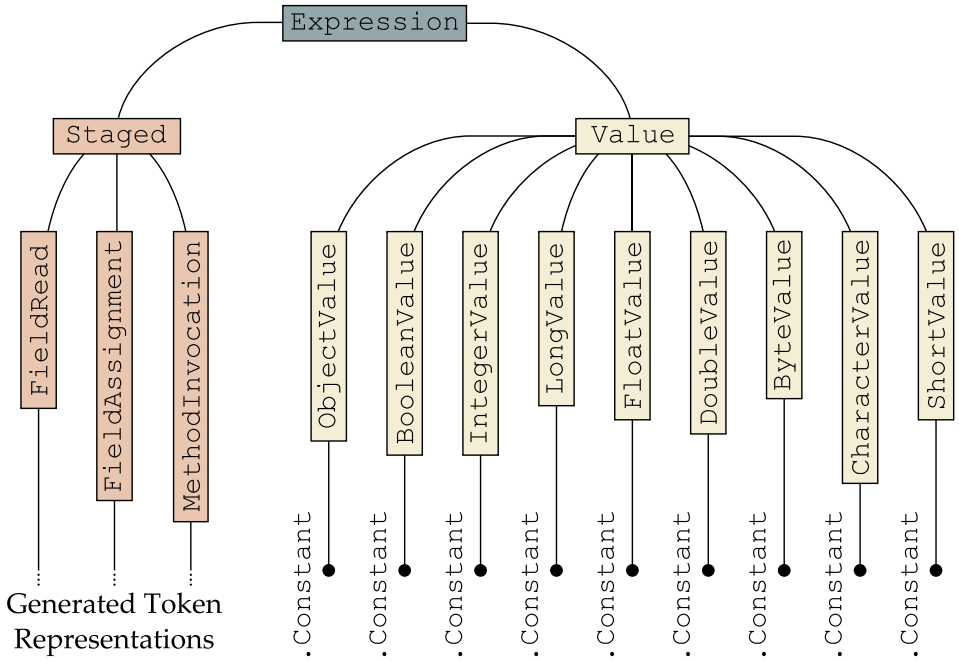


FIGURE 4.4: `Expression` hierarchy

Token-representation-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:

- Unloading of expression DAGs from `GlobalCarrier` and `LocalCarrier` instances. The latter is mainly used internally to locally carry reference-typed terms, but can also be used by authors like `GlobalCarrier`.
- Enforcing language boundaries. The accepted languages per argument are statically known and stored as static fields on the token-representation class.
- Extracting raw versions of conversion-wrapped expression roots. This is necessary because sometimes, in order to maintain soundness externally, terms may be wrapped in internal-use-only conversion expressions.

Without this code generation one would need to provide additional information for every IR-node instantiation (i.e. factory-method call site) and use many conditionals or rely on run-time reflection.

[4]There is no (visible) conversion expression like the implicit-staging prototype had. In the current implementation downcasts force materialization while upcasts, boxing, and unboxing have no effect.

4.2.6 Weave Analysis and Weaving

The weaving step entails injecting bytecode for lifting, materialization, term (i.e. token-representation) construction, and carrier instantiation and loading. Here again, working on the bytecode level is beneficial as it is much less restrictive than source code. Yet, for good reasons, bytecode is still type-checked and restrictive enough that adjustments are necessary: For instance, suppose one has 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. The following shows a similar situation and weaving result for the code in Listing 4.5:

0	:	`iload`	0		0	:	`iload`	0
1	:	**`ifle`**	6	$\rightsquigarrow$	1	:	**`ifle`**	7
2	:	`iload`	0		2	:	`iload`	0
				$\rightsquigarrow$	3	:	`invokestatic`	$\mathit{lift}_{\texttt{int}}$
3	:	`invokestatic`	<u>dec</u>	$\rightsquigarrow$	4	:	`invokestatic`	$\mathit{factory}_{\texttt{dec}}$
4	:	`istore`	0	$\rightsquigarrow$	5	:	`astore`	0
5	:	**`goto`**	9	$\rightsquigarrow$	6	:	**`goto`**	11
6	:	`iload`	0		7	:	`iload`	0
				$\rightsquigarrow$	8	:	`invokestatic`	$\mathit{lift}_{\texttt{int}}$
7	:	`invokestatic`	<u>inc</u>	$\rightsquigarrow$	9	:	`invokestatic`	$\mathit{factory}_{\texttt{inc}}$
8	:	`istore`	0	$\rightsquigarrow$	10	:	`astore`	0
9	:	`iload`	0	$\rightsquigarrow$	11	:	`aload`	0
10	:	`invokestatic`	<u>inc</u>	$\rightsquigarrow$	12	:	`invokestatic`	$\mathit{factory}_{\texttt{inc}}$
				$\rightsquigarrow$	13	:	`invokestatic`	$\mathit{mater}_{\texttt{int}}$
11	:	**`ireturn`**			14	:	**`ireturn`**	

Similar changes have to be performed all over to satisfy the JVM's bytecode verifier, e.g. value lifting before control-flow merges as in line 10 of Listing 4.4. The main task of the prior weave analysis is to determine where these adaptations are needed and direct the weaving pass accordingly.

The implementations of the analysis and weaving itself are somewhat involved. After all, there are more aspects in need of handling as for instance compared with the implicit-staging prototype's unstaging step. However, the exact details and edge cases are not particularly interesting here and beyond the scope of this thesis's discussions.

4.2.7 Run-Time Support

The framework's run-time support consists mainly of a collection of static methods in a hidden, i.e. non-public, class called `Dispatcher`, which is made public only at the start of the Java agent.

Carrier subclasses are modified in such a way that only the internals of the framework implementation can construct "empty" instances to carry abstract syntax. This is ensured by adding a new constructor (on the fly) accepting a special parameter type that itself is generated at load time. Carriers contain a `payload` field of type

`Expression` that is written (loaded) and read (unloaded) using hidden methods in the dispatcher class.

The dispatcher class also contains methods to persist data between the time of transforming (the methods in) a class and its actual initialization. It is used to temporarily store and retrieve static information (e.g. line numbers etc.) to be added to expression-DAG nodes in client code without relying on run-time reflection (and causing potential class-loading issues). During transformation of a method this static information is stored using a persistence method which returns an integer ID. Static fields are then added to the declaring class of the method that is transformed together with field-initialization code that retrieves the data from the persistent storage (and frees its alloted spot) using the previously generated ID.

Permanent caching works by using static (closure-holder) fields on the declaring class of a woven method. Their contents are provided to token-representations instances when they are created. For default, non-permanent caching a size-limiting cache implementation (provided by Guava [5]) is used.

4.3 Examples

Tame staging was introduced with several small code examples for the sake of simple illustration. In this section I will show slightly more meaningful examples. However, keep in mind that these are still specifically constructed with the framework in mind. Considering even larger examples would be of interest. However, the goal of the present work is not to provide functionality for a specific EDSL but for language embedding in general. As such, no single example alone, big or small, would be able to convince of its usefulness.

4.3.1 Centroid Calculation (Vector EDSL)

Consider a collection of high-dimensional, immutable weighted vectors (see Listing 4.19), or a point cloud, for which we want to determine its centroid. This is a simple calculation in which all the weighted vectors are added and divided by their total number. Listing 4.20 shows an implementation of this as a method taking a variable amount of weighted vectors.

LISTING 4.19: Weighted vector (temporary container)

```
1  public final static class WeightedVec {
2    private final Vec vec;
3    private final double weight;
4
5    public WeightedVec(Vec vec, double weight) {
6      this.vec    = vec;
7      this.weight = weight;
8    }
9
10   public Vec    getVec()    { return vec; }
11   public double getWeight() { return weight; }
12 }
```

LISTING 4.20: Centroid-calculation method

```
public static Vec centroid(WeightedVec... weightedVecs) {
  if (weightedVecs.length == 0) {
    throw new IllegalArgumentException();
  }

  double w = weightedVecs[0].getWeight();
  Vec v = weightedVecs[0].getVec();

  if (w != 1.0) {
    v.times(w);
  }

  for (int i = 1; i < weightedVecs.length; i++) {
    w = weightedVecs[i].getWeight();

    if (w != 0.0) {
      if (w == 1.0) {
        v = v.plus(weightedVecs[i].getVec());
      } else {
        v = v.plus(weightedVecs[i].getVec().times(w));
      }
    }
  }

  if (weightedVecs.length > 1) {
    return v.times(1.0 / (double) weightedVecs.length);
  } else {
    return v;
  }
}
```

The execution time was benchmarked on a 3 GHz Intel Core i7 machine with 8 GB of RAM with JRE 8 (Java HotSpot™ 64-Bit Server VM) for 20 randomly generated arrays of weighted vectors of size 100,000. Warm-up and measurement were performed in a 1,000 iterations loop, respectively, where one of the 20 weighted-vector arrays was chosen in a cyclic fashion.

Figure 4.5 shows the results when using shallow embedding (S), tame-staging local carrying (L), manual deep embedding (D), and tame-staging global carrying (G). The latter two were run on a slightly adapted version of the code using `VecE` instead of `Vec`. The optimizations implemented by all staged-EDSL (i.e. excluding shallow embedding) follow the ones shown in Listing 4.9. For each version ten measurements were collected and aggregated.

It is no surprise that the results show shorter execution times for the staged, optimized versions as compared to the shallow embedding. For smaller amounts of data (i.e. vector dimensions) this might not hold, as discussed in Section 3.5.2 of Chapter 3. Comparing the averaged results, both L and G were about 36% faster than S. They were also marginally (0.2%) faster than D, however this is unlikely to be significant. At least in this benchmark the framework-assisted embeddings were not slower than the manual ones.

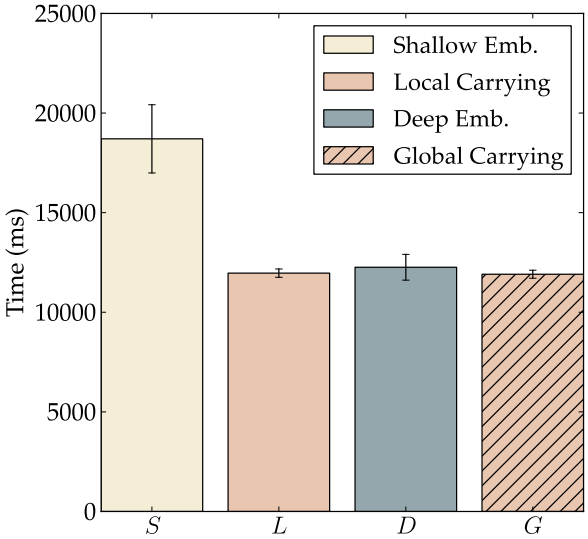


FIGURE 4.5: Centroid-calculation benchmark results

In all staged versions materialization is delayed until exiting the method body. Unfortunately this is not without problems. If the list of weighted vectors is very large, the reified programs (i.e. expression DAGs) will also be very large. In fact, they might become so large that isomorphism checks (let alone author-defined processing) fail due to limited call-stack size when deeply traversing the constructed programs.

There is no easy way out. One could try to impose a size limit and, for the locally carried case, automatically trigger materialization. This has bad effects on uniformness, since potential points of materialization become less predictable. Allowing this through an optional setting on tokens might be a good compromise in the future. Alternatively, users who anticipate the creation of large programs can manually trigger materialization with an identity function, which however is detrimental to seamlessness.

4.3.2 Radar (Region EDSL)

The region EDSL introduced in Section 2.2.6.2 of Chapter 2 can be defined using the tame-staging framework and its `@Stage` annotation as shown in Listing 4.21. The (delayed-) shallow-embedding implementations are omitted for the sake of brevity and avoiding redundancy. They can be easily derived from the earlier descriptions.

For this implementation, global carrying is used together with a hybrid-state carrier. While this is usually not recommended it can sometimes be helpful: The core EDSL here is defined without an `@Stage`d token for region-membership checking. Instead, the materialization can only be triggered by `Region.asFunction`. The `Region.isInside` method is an auxiliary EDSL construct, which only materializes "itself", i.e. the carried abstract syntax, if necessary, before performing the region-membership check.

The fact that it is not annotated can be seen by EDSL users to whom it communicates standard method behavior, requiring reading the author's documentation.

LISTING 4.21: `@Staged` region EDSL

```
public final class Region extends GlobalCarrier {
  private BiFunction<Float, Float, Boolean> function;

  private Region(BiFunction<Float, Float, Boolean> function) {
    this.function = function;
  }

  @Stage(language = RegionL.class)
  public static Region empty() { ··· }
  @Stage(language = RegionL.class)
  public static Region circle() { ··· }
  @Stage(language = RegionL.class)
  public static Region square() { ··· }

  @Stage(language = RegionL.class)
  public static Region scale(Region region, float scaleX,
                             float scaleY) { ··· }
  @Stage(language = RegionL.class)
  public static Region translate(Region region, float transX,
                                 float transY) { ··· }

  @Stage(language = RegionL.class)
  public static Region outside(Region region) { ··· }
  @Stage(language = RegionL.class)
  public static Region intersect(Region regionA, Region regionB) { ··· }
  @Stage(language = RegionL.class)
  public static Region union(Region regionA, Region regionB) { ··· }

  @Stage(language = RegionL.class, isStrict = true)
  public BiFunction<Float, Float, Boolean> asFunction() { ··· }

  // Not annotated!
  public boolean isInside(float x, float y) {
    if (function == null) {
      function = this.asFunction();
    }

    return function.apply(x, y);
  }
}
```

On the other hand, the annotated tokens have a clearly defined behavior, while of course the result of materialization depends on the EDSL author's processing.

The compiler for this language is not shown here. As an obvious, simple optimization it detects redundant unions and intersections and uses constant information where available. Furthermore, instead of creating a composition of `Closure` instances, it actually generates Java source code that is compiled using Javassist's inbuilt compilation feature [14], i.e. it avoids interpretation overhead like the MetaOCaml implementation in Section 2.3.1.1.

The original use case for this EDSL as described by Carlson et al. [27] was sensing and tracking real-world objects. Let us explore this again here by defining a very simple system that checks a defined set of regions for contained objects

LISTING 4.22: Simulated "radar"

```
public class Radar {
  private final List<NamedRegion>   namedRegions;
  private final List<TrackedObject> trackedObjects;

  public Radar(List<NamedRegion> namedRegions,
               List<TrackedObject> trackedObjects) {
    this.namedRegions   = namedRegions;
    this.trackedObjects = trackedObjects;
  }

  public List<TrackingData> track(long duration, int width, int height) {
    List<TrackingData> res = new LinkedList<>();
    Random r = new Random(42);

    Region combined = empty();
    for (NamedRegion n : namedRegions) {
      combined = combined.union(n.getRegion());
    }

    long time = 0;
    while (time < duration) {
      for (TrackedObject t : trackedObjects) {
        if (r.nextBoolean()) {
          t.setX(r.nextInt(width) - width / 2);
          t.setY(r.nextInt(height) - height / 2);
        }

        if (combined.isInside(t.getX(), t.getY())) {
          for (NamedRegion n : namedRegions) {
            if (n.getRegion().isInside(t.getX(), t.getY())) {
              res.add(new TrackingData(time, n.getName(), t.getId()));
            }
          }
        }
      }

      time += 10;
    }

    return res;
  }
}
```

and logs this information. Listing 4.22 shows its implementation, omitting the implementations of `NamedRegion`, `TrackedObject`, and `TrackingData`. The sensing and tracking functionality is only simulated by making a random decision to update a tracked objects coordinates (randomly) and by a simple, incremental progression of time up until a time limit is reached. The collected tracking data is returned as a list.

This implementation was also benchmarked. Listing 4.23 excerpts the kind of named regions used for the benchmark. In total there are 14 regions, outlining (fictitious) zones and structures at a small civilian airport. Figure 4.6 visualizes all

LISTING 4.23: Named regions (excerpt)

```
NamedRegion r1 = new NamedRegion( "Airfield A",
  translate(scale(square(), 60.0f, 4.0f), -20.0f, 60.0f)
);

NamedRegion r2 = new NamedRegion( "Airfield A Danger Zone",
  translate(
    intersect(
      outside(scale(square(), 64.0f, 6.0f)),
      scale(square(), 80.0f, 20.0f)
    ),
    -20.0f, 60.0f
  )
);

NamedRegion r3 = new NamedRegion( "Airfield A Inner Proximity Zone",
  translate(annulus(64.0f, 90.0f), -20.0f, 60.0f)
);
...
NamedRegion r13 = new NamedRegion( "Hangar E",
  translate(scale(square(), 9.0f, 6.0f), -20.0f, -220.0f)
);

NamedRegion r14 = new NamedRegion( "Area Proximity Zone",
  translate(annulus(300.0f, 400.0f), 80.0f, -50.0f)
);
```

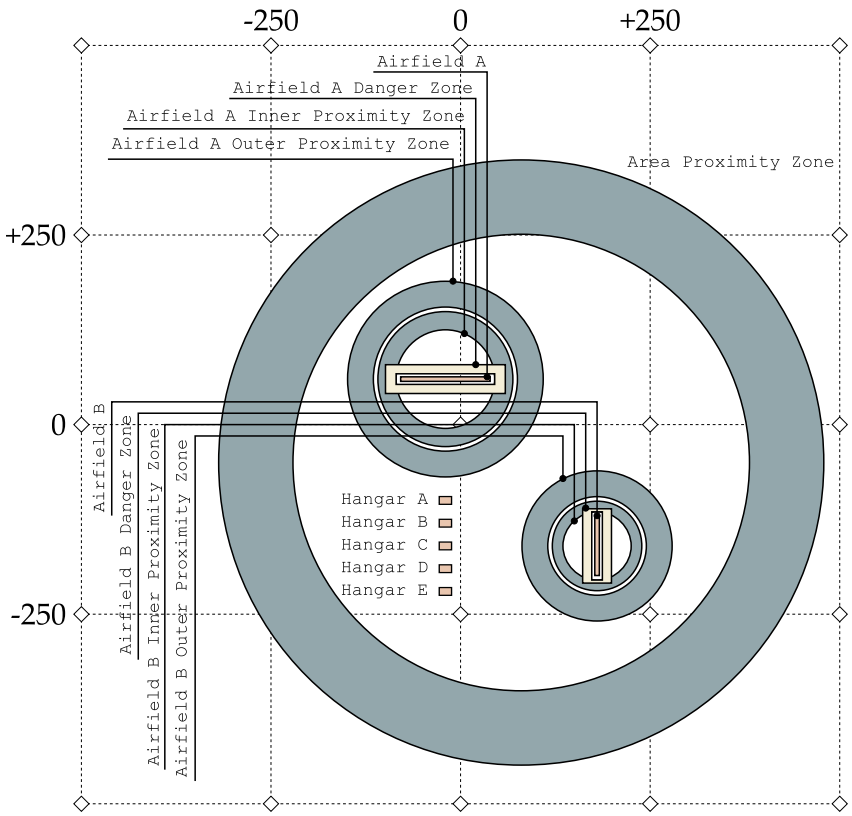


FIGURE 4.6: Named-regions visualization

the subject regions combined in a single grid. The definitions also contain calls to a yet undefined method called `annulus`. Due to global carrying, like `times2` in Section 4.1.4, this method can be defined as shown in Listing 4.24. In the work by Carlson et al. [27] this is used as an example for a *derived region*.

LISTING 4.24: Custom expansion (derived region)

```
1 public static Region annulus(float innerRadius, float outerRadius) {
2   return intersect(
3            outside(
4              circle(innerRadius)
5            ),
6            circle(outerRadius)
7          );
8 }
```

For the benchmark, a `Radar` instance was created with the list of named regions and 100,000 randomly generated objects to be tracked with random positions. Tracking was done for 1,000 simulated time units. Only a delayed shallow embedding (S) and the annotated, tame-staging implementation (A) were compared, with warm-up (S_W, A_W) and cold (S_C, A_C), where in the former case a single tracking pass was done before the measured one.

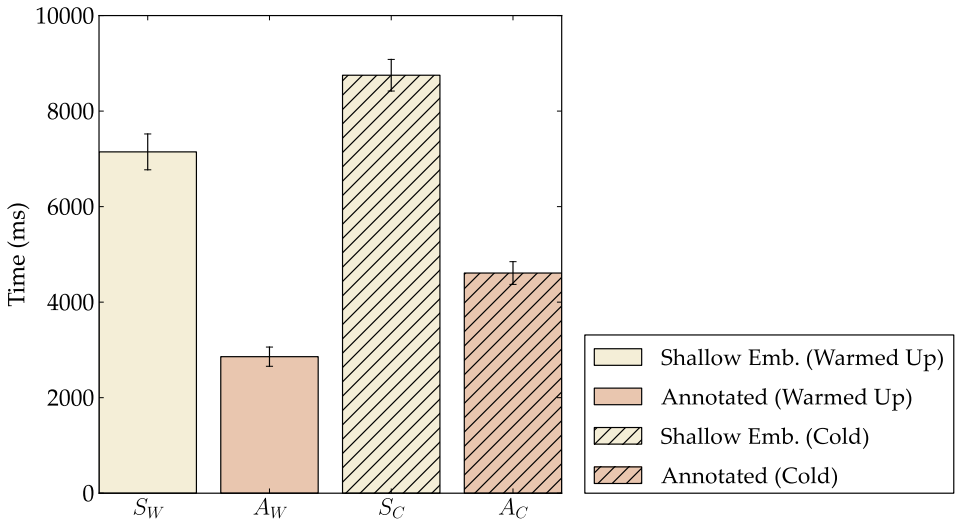


FIGURE 4.7: Radar benchmark results

Figure 4.7 shows the aggregated results of ten runs per version.[5] Removing the overhead of function application (i.e. "interpretation" in the context of delayed shallow embedding) by compilation had the desired (and not unexpected) result. The annotated version was 60% faster than the shallowly embedded one when comparing the warmed-up benchmark results, and 47.3% faster in the cold-startup benchmark.

[5]The experimental hardware and software setup remains unchanged from that of the other experiments in this section.

Note that similar results can surely be achieved with manual deep embedding. However, there will not be any guarantees for staging behavior, no constant input can be detected and used, and redundant processing is not automatically avoided. Moreover, the compiler for `RegionL` was implemented as a simple traversal and code emission from the auto-generated IR. With manual deep embedding, EDSL authors would need to define their own IR data types and correctly implement IR construction in the EDSL tokens.

4.3.3 Connections (`ImmutableList`-Processing EDSL)

For the final example let us revisit the scenario and EDSL for processing immutable lists presented in Section 1.1 of Chapter 1. Listing 4.25 defines the syntax of a locally carried (hiddenly staged) EDSL with tokens `map` and `filter`, which directly work on Guava's `ImmutableList` data type.

LISTING 4.25: `ImmList` definition

```
1  public final class ImmList { private ImmList() { }
2    @Stage(language = ImmListL.class)
3    public static <F, T> ImmutableList<T> map(
4        ImmutableList<F> list,
5        @Accept(languages = {}) Function<? super F, T> function) {
6      ImmutableList.Builder<T> b = ImmutableList.builder();
7      for (F f : list) { b.add(function.apply(f)); }
8      return b.build();
9    }
10   @Stage(language = ImmListL.class)
11   public static <T> ImmutableList<T> filter(
12       ImmutableList<T> list,
13       @Accept(languages = {}) Predicate<? super T> predicate) {
14     ImmutableList.Builder<T> b = ImmutableList.builder();
15     for (T t : list) { if (predicate.apply(t)) { b.add(t); } }
16     return b.build();
17   }
18 }
```

The reason the function and predicate arguments are annotated with `@Accept` has to do with the fact that since their types are interfaces, there is no guarantee that they are not global carriers. Making them hard language boundaries avoids the need for dynamic checking.

Again, the processing of `ImmListL` comes in the form of compilation to Java code with its own overhead as well of that of additional class loading. However, scenarios like that of Section 1.1 are ideal for it, since we can expect the list of connections[6] to be large enough for the initial effort to (likely) pay off.

Likewise, very similar queries may be encountered repeatedly. Hence, it is imperative that compilation results are cached in some fashion or other. Fortunately, the tame-staging framework takes care of this. In fact, the implementation of both syntax and meaning of `ImmListL` is about 20% shorter than the roughly corresponding deep embedding.

[6] Admittedly, in practice a more suitable data structure than lists (or arrays) would be used.

An experiment was conducted for the query methods introduced in Section 1.1. The "database" was randomly initialized with 2,000,000 entries. Plane connections from Tokyo to Frankfurt costing up to 2,500$ were queried 100 times in succession. The shallow-embedding (S) and tame-staging (A_D) versions of the EDSL were run on the code of Listing 1.2. Additionally, a slightly modified query method was run with tame staging, where materialization is explicitly forced (A_F) right before line 13 (using an identity function, i.e. `l = materialize(l)`). The compiled deep-embedding (D_C) implementation was run on the query method of Listing 1.3 and a version (D_I) using Guava's `FluentIterable` EDSL was run on a similar-looking method. Finally, the handwritten search (M) of Listing 1.4 was also benchmarked.

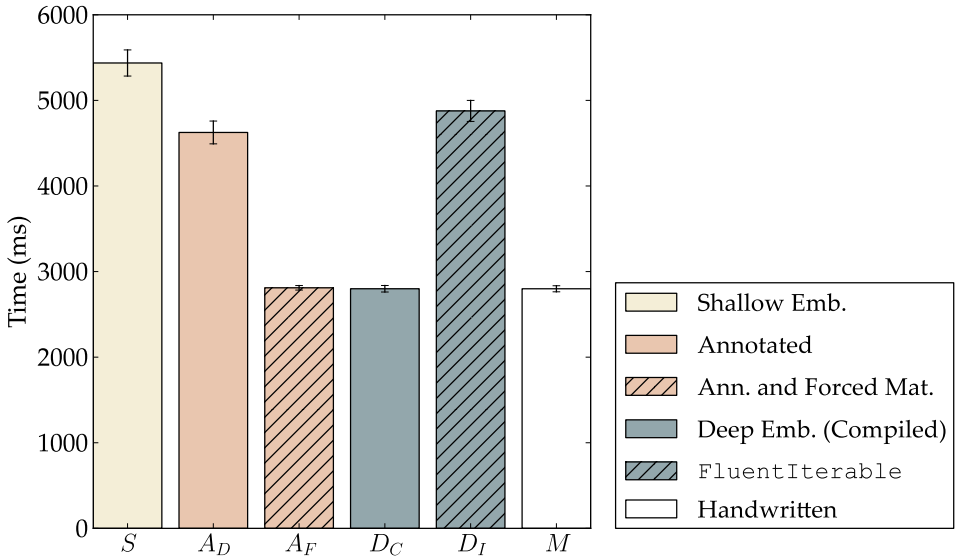


FIGURE 4.8: Connections-query benchmark results

Figure 4.8 shows the aggregated results of ten runs per EDSL (and query) implementation. Switching to the tame-staging version (A_D) for running the shallow-embedding query sped it up by about 14.9%. However, the compiled deep-embedding version fared much better. It was about 48% faster than shallow embedding. Similar is true for the handwritten M. The reason why A_D is not on par with the others is because of the inlining behavior of staged-term composition. After evaluating the first part of the result for checking its size, the other implementations all explicitly reuse the already generated list, while A_D reruns the operations, then with an added filtering. As mentioned above this is easily fixed by adding one line, so A_F is on par with D_C and M. Despite looking nearly identical to the query used for D_C, the (non-compiling) `FluentIterable` version (D_I) was even slower than A_D, which illustrates again how much difference compilation can make.

4.4 Evaluation

With the introduction of the tame-staging framework and some of its use cases we are now well equipped to evaluate its merits and shortcomings along the lines of the previous chapters.

4.4.1 Reliability

The `@Stage` annotation and the other framework features enable hidden and deep-embedding style dynamic staging in a safe and uniform way. Strict language boundaries (and `@Accept`) are maintained and the behavior when they manifest is clearly defined. Furthermore, the system is in full control of how staging occurs and what IR is used. As the EDSL author has no say in these matters, it is easy to establishing the compositionality of staging.

Black-box behavior is largely avoided through the same means, making the approach very uniform. Why not completely uniform? This is a matter of perspective: For locally carried terms this is always a hard guarantee. However, allowing global carrying means that for better or worse some, i.e. auxiliary, tokens do not necessarily have uniform, predictable behavior, e.g. cause internal materialization. This is an inherent property of global carrying and cannot be addressed in a general fashion (i.e. while providing the same freedom). However, note that even in this setting EDSL users can still rely on the "basic" `@Staged` tokens' behaviors.

Many cases of auxiliary or derived tokens are meant for the purpose of expansion or rewriting. In the future it may be worthwhile to add an `@Rewrite` annotation which acts similarly to `@Stage` for rewriting while staging and follows fixed rules (cf. Section 2.2.2 in Chapter 2). While this is again reminiscent of the opaque-workflow pitfall, at least it would document and restrict rewriting behavior.

4.4.2 Performance

The freedom of performing optimizations is on the same level as deep embedding. However, the introduced indirection is slightly limiting. For instance, with manual deep embedding it is possible to inspect some values during staging and make further staging decisions internally, thus exploiting more information than just the EDSL-program shape. Essentially, programs can be specialized to very specific input values. Of course, as mentioned in Section 2.3.3 of Chapter 2 this is detrimental when attempting to avoid redundant processing.

Tame staging strikes a balance where only the shape and actually constant values (but not variable values) ever become subject to author-defined processing. While this indirection is limiting and causes overhead it also opens up the door to avoiding dynamic staging where applicable. Currently the result of processing statically known programs is permanently cached, but the IR construction still occurs dynamically. In the future, such cases could be handed over to the author-defined processor at load-time to remove the dynamic-staging overhead similar to implicit staging.

The benchmarks in Section 4.3 showed favorable results. However, it would be disingenuous to sweep under the rug the run-time overhead of language-boundary enforcement, carrier unlaoding, and other checks during staging, combined with the indirection of processing, i.e. isomorphism checks and cache lookup, and closure execution, i.e. environment creation and value access.

Benchmarking the simple, initial code example of Listing 4.2 (and an adapted version) in this chapter illustrates this issue. Here, five EDSL implementations were considered: A non-staged, shallow embedding (S), an `@Staged`-annotated

version (see Listing 4.3) with both the trivial compiler (A_N) of Listing 4.6 as well an optimizing one (A_O) (cf. Section 4.3.1). Also, both a non-optimizing (D_N) as well as an optimizing (D_O) (manually) deeply embedded version were implemented. The non-optimizing A_N and D_N simply delay evaluation.

The code was called with `example(i % 20, a, b, c)`, where `i` incrementally ranges from 0 to 99,999. The variables `a`, `b`, and `c` contain randomly initialized vectors of sizes 100, 1,000, or 10,000. For each implementation the execution time was measured ten times per vector size in the same hardware and software environment as the other experiments in this chapter, after a single warm-up run on the largest vector size.

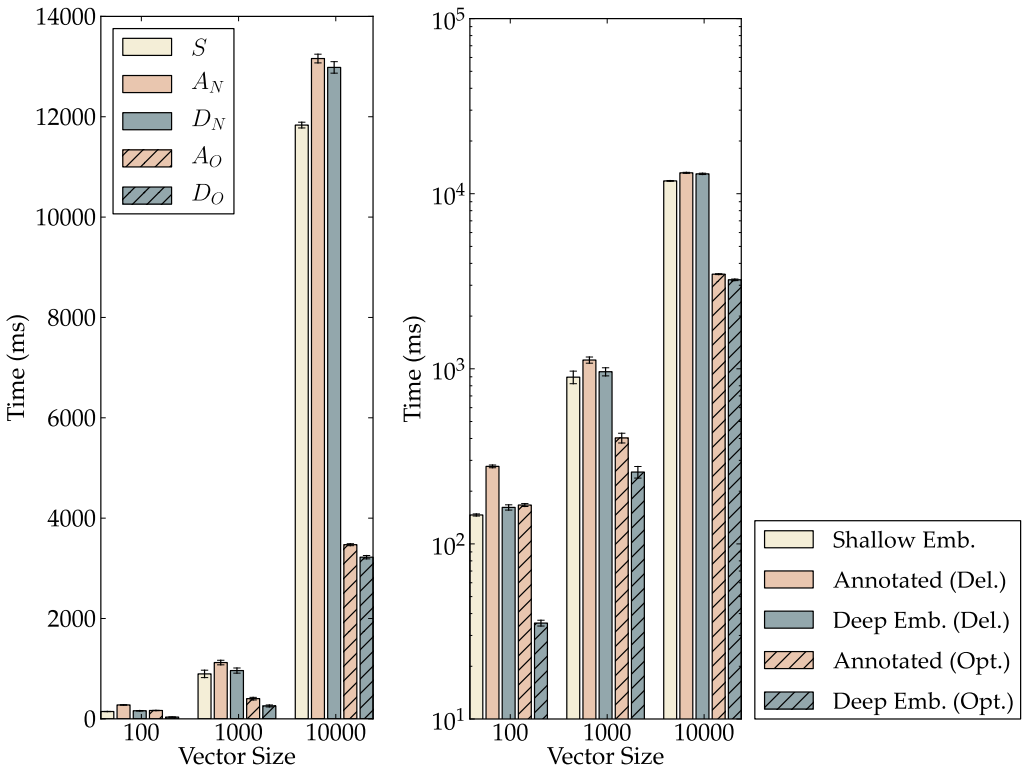


FIGURE 4.9: Benchmark results (linear and logarithmic scale)

Figure 4.9 shows the averaged results of the benchmarks for the different vector sizes. In terms of performance, using the `@Stage`-based implementations (A_N, A_O) became only worthwhile for vectors with more than 1,000 elements. Below that, the performance is subpar: D_O was about 78.8% faster than A_O. Even above, the handcrafted deep embedding (D_N, D_O) yielded slightly better results: D_O was 36.1% faster than A_O for vectors of size 1,000 and still 7.2% faster for vectors with 10,000 elements.

It comes as no surprise that there is a cost associated with this novel linguistic-abstraction mechanism for language embedding. However, it also comes with a potential for future improvement unavailable otherwise: Without changing the interface it is possible to change and optimize the back end.

4.4.3 Usability

Tame staging excels on the usability front. Dynamic-staging concerns can be hidden from EDSL users and no restrictions are imposed on where EDSL-program fragments may be composed. Together with the behind-the-scenes load-time transformation on the fly, a high degree of seamlessness is achieved. At the same time, thanks to using Java's annotations and their sort of self-documentation, it is very easy for EDSL users to figure out where staging occurs and to which language a token belongs. This could be further improved by tool support, for instance by an IDE plugin that highlights staging constructs and materialization points using EDSL-dependent color coding.

Debugging is helped in several ways. For one, the static context of a token can be used to create debug messages as illustrated in Section 4.1.9. Furthermore, the uniformness of staging leads to predictability, also desirable for debugging. Finally, The weaving step maintains debug symbols as much as possible, which makes it possible to use existing tools for debugging. For instance, EDSL user can exploit this to inspect staged terms and to track down issues with conditional EDSL-program composition.

Finally, the maintenance and evolution of EDSLs and EDSL-reliant user code benefits from the load-time nature of the system. However, it is important to point out one flaw of exclusively relying on load-time metaprogramming: Some errors will be detected somewhat late. For instance, in the current implementation, visibility restrictions (of `@Configure`) are only checked at load time and then lead to a message warning of the ineffectiveness of an annotation. An IDE plugin could be provided to allow users to detect such issues early on. Alternatively, an annotation preprocessor or checker [15] could do the trick.

4.4.4 Comparison with Related Work

The idea of providing language support for integrating DSLs into host languages has been investigated before. First and foremost, there is the dynamically typed Python-inspired Converge [109]. In this language, DSLs or sublanguages can be defined both in terms of syntax and semantics and are used in explicitly language-delimited blocks (and expressions).

Examples of statically typed languages with a sole focus on custom syntax are Wyvern [83] and ProteaJ [63]. Both use type-associated syntax or operators to integrate sublanguages into their respective host languages. However, these sublanguages as a whole are still implicit.

All these approaches deliberately argue for a departure from the host language's look-and-feel and attempt to solve nontrivial parsing issues. Hence, I claim that none of these approaches really fit into the DSL-embedding style put forward by Hudak [59]. Moreover, for these approaches 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.

Table 4.1 augments the comparison table of Chapter 3. Besides the related work presented in Chapter 2 and 3, there is to my knowledge only one (major) other

work that explores extending the host-language to accommodate Hudak-style DSL embedding: Scala-Virtualized [89]. It will be briefly discussed in the following.

TABLE 4.1: Comparison of implementation approaches

		Yin-Yang	JIT Macros (Project Lancet)	LMS with Scala-Virtualized	Implicit-Staging Prototype	Tame Staging (`@stage`)
G	Global					
L	Local					
E	Expression					
A	Annotation					
D	Declaration					
I	EDSL Interface					
T	Typing					
◡/▲	High/Good					
⋇/♦	Fair					
◠/▼	Low/Bad					
STAGED-EDSL DESIGN						
Scope		*L*	*G*	*L*	*E*	*G*
Delimitation		*I*	*T*	*T*	*D*	*A*
Staticity		◡	◡	⋇	◡	◡
Dynamicity		⋇	◡	◡	⋇	◡
Transparency		◡	⋇	⋇	◡	◡
EVALUATIVE COMPARISON						
Reliability		◇	▽	◇	◇	△
	Safeness	♦	▼	♦	♦	▲
	Uniformness	♦	♦	♦	♦	▲
Performance		△	△	△	△	△
	Overhead	♦	▲	▼	▲	♦
	Optimization	▲	▲	▲	▲	▲
	Adaptiveness	♦	▲	▲	♦	▲
Usability		△	▽	△	△	△
	Seamlessness	▲	♦	▲	▲	▲
	Maintenance	♦	♦	♦	▲	▲
	Debugging	▲	▼	▲	♦	▲
	Documentation	♦	▼	♦	▼	▲

4.4.5 Scala-Virtualized

Scala-Virtualized [89] is a custom branch of the Scala compiler. It explores various general-purpose tweaks for improving Scala as a meta language for EDSL hosting. The main idea is to allow extensive customization of host-language behavior, or *language virtualization*, way beyond simple operator overloading.

It is possible to redefine the meaning of inbuilt control structures and even variable assignment by defining special methods (and singleton objects). For instance, the Scala expression

$$\texttt{\textbf{if}}\ (\mathit{condition})\ \mathit{trueBranch}\ \texttt{\textbf{else}}\ \mathit{falseBranch}$$

is *desugared* into the following method invocation:

$$\texttt{\_\_ifThenElse(}\mathit{condition}\texttt{,}\ \mathit{trueBranch}\texttt{,}\ \mathit{falseBranch}\texttt{)}$$

It is possible to define various versions (with different type signatures) so that depending on the involved types (e.g. of the condition) a specific implementation is called that is different from the default one.

This can be combined with LMS [87] to the effect that EDSL code looks like regular Scala code with staged semantics. Without Scala-Virtualized, the illusion is not entirely perfect, e.g. conditionals in pure LMS have to be expressed as standard method calls.

Additionally, yet another form of desugaring is proposed to address the issue that "[e]xtending traits and creating objects in a certain way just to define a little DSL program may be asking too much." [87], i.e. to remove the boilerplate associated with LMS-style embedding: So-called *DSL scopes*, which essentially are DSL-named blocks that expand to boilerplate, LMS-style EDSL-program definitions.

Scala-Virtualized also adds `SourceContext`, an implicit parameter that allows for better error reporting, thus addressing an important aspect of debugging EDSLs. However, static context is not used much further, e.g. for constant detection, caching, or preprocessing.

Thanks to these additions, the ratings for LMS-usability increase slightly. However, while maintenance of existing EDSLs is not curtailed, the overall maintenance is negatively affected by the fact that now a custom compiler (branch) is required.

Why is Scala-Virtualized related and relevant for comparison with the tame-staging framework? After all it does not explicate embedded languages or treat its programs on a (meta or) higher level. This is precisely why it is so related: It showcases a different school of thought. One in which a new language feature should be as general as possible. It is true that generality is commonly a good direction to take. However, I believe that when a host language is extended for the specific purpose of language embedding, yet without actually giving embedded languages special treatment, improvement opportunities (on various fronts) are missed (while still adding potentially complex features).

4.5 Discussion and Summary

The tame-staging framework presented in this chapter provides a high-level, dedicated abstraction mechanism for embedded languages. 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.

For the time being, solely relying on the load-time bytecode transformation functionality of the JVM makes the approach a pragmatic host-language extension

to Java. Hence, with such an extension language embedding becomes (only) "almost" a first-class feature. This serves well for experimentation and smoothing out the rough edges (e.g. `@Stage` inheritance, inlining-behavior configuration, handling of large-program staging, etc.). It can be easily adopted by users and may aid in testing the waters to garner acceptance for first-class, native support.

I do believe that it is worthwhile to eventually consider tightly integrating the presented ideas into the design of future languages and virtual machines (or updated versions thereof). The annotations could be replaced by keywords and (explicit) language classes could receive special treatment. Additional virtual-machine support sounds promising in particular to tackle the cost of abstraction and to improve general performance, for instance by integrating the feature into JIT compilation. After all, in the case of the JVM, a lower-level, internal implementation of staging and caching can be expected to perform much better than the additional indirection on top of existing bytecode.

Yet another issue is the crude way of exposing and traversing the staged IR as a generic data structure. An approach like that of Yin-Yang [68], in which the static type system limits the actual types of IR nodes, looks appealing, yet comes with the additional step of generation and macro-directed translation. However, in a system that integrates first-class language embedding natively, a `@Staged` (or maybe **`stage`**`d`) token could be exposed as a special data type, for instance

```
token class Vec::times { ··· }
```

with (constant) instance fields, e.g. `Vec::times.v` and `Vec::times.s`, for access to encapsulated data that is statically known to be stored on a token representation.

Alas, this and other native support is still far off and to be considered for future work. Finally, let us recapitulate the central achievements of the developed pragmatic-extension tame-staging framework:

- As in implicit staging, tame staging explicates EDSLs for detecting domain-specific code and guiding EDSL-program handling. These explicit EDSLs may be considered akin to a parallel (language-) type system.

- Dynamic staging is accomplished through a combination of static and dynamic techniques, which essentially provide selective, i.e. EDSL-specific, lazy-evaluation with custom EDSL-program processing.

- The (definition-site) `@Stage` annotation serves as a unique (i.e. only one language per token) and visible (i.e. self-documented) language-association mechanism for EDSL tokens and houses token-specific settings.

4.6 Acknowledgments

The contents of this chapter are, in large part, based on and reproducing the research to be published in co-authorship with Prof. Chiba at the 2015 International Conference on Generative Programming: Concepts and Experiences (GPCE 2015) under the title "Almost First-Class Language Embedding: Taming Staged Embedded DSLs" [92]. I was the first author of this paper.

Chapter 5

Conclusions

This thesis introduced a progression from the low-level, manual embedding of (staged) domain-specific languages to a novel framework that is grounded in turning embedded languages from implicit to explicit entities, which in turn serves as the basis for an almost first-class, self-documented, linguistic abstraction for dynamic EDSL-program composition. Its novelty lies not with the idea of providing meta-level transformations. For that, existing approaches like syntactic macros or rewrite-rule-based preprocessing arguably suffice. Instead, the novelty lies in making (seemingly) meta-level features in the context of DSL embedding palatable by EDSL users and still sufficiently customizable by EDSL authors.

Palatability is also a concern when it comes to the creation and deployment of novel functionality. By relying on the JVM's Java-agent feature and performing language extension at load time, the proposed features can be deployed in a pragmatic fashion. Not only that, but also the so-realized EDSLs themselves can be deployed, bug-fixed, and evolved without diminishing the advantages of separate compilation and dynamic linking. Implementing language extensions in this fashion has been explored before, e.g. in the context of implementing AspectJ [56, 70], but, to the best of my knowledge, not in the context of simultaneously improving the reliability, performance, and usability of EDSLs through a dedicated linguistic abstraction.

One could argue that attempting to integrate[1] DSLs as an inbuilt feature should naturally be taken further to a level where sublanguages (syntactically) coexist with the host-language (as for instance in the case of LINQ [108]) some times even on equal terms [34, 109]. However, while this is yet another related and interesting direction of research, the notion of full-on syntactic and semantic language composition eschews the advantages of language embedding such as the reuse of infrastructure, familiarity with the host language, type safety, and dynamic program construction.

The ideas put forward by this thesis are surprisingly simple. In fact, in the tame-staging framework of Chapter 4, global carrying is on its surface virtually indistinguishable from traditional deep embedding, i.e. method calls generating

[1] I am avoiding the term "embed" for this because of its (Hudak-style embedding) connotations.

terms, yet with guarantees for compositional staging, caching, language-boundary enforcement, and other advantages afforded by the nature of a dedicated linguistic abstraction. The (mental) switch to the more transparent local-carrying style merely requires a passing knowledge and understanding of the method-local staging behavior and lazy-evaluation-like materialization triggering.

While the gains are not immediately obvious, the language-embedding pitfalls of Section 2.3.3 in Chapter 2 are addressed as best as possible in the scope of my research. Let us once more quickly and collectively consider the achievements of this thesis in order:

(i) The advantages of traditional EDSL implementation approaches were identified but so was the fact that these are counterweighted by their ad-hoc nature.

(ii) Implicit staging was proposed to illustrate the benefit of considering EDSLs on a higher level, and serves as a general framework for assisting the implementation of reliable, statically staged EDSLs.

(iii) A concrete investigation was conducted by way of a proof-of-concept implementation of a load-time (metaprogramming) implicit-staging framework for expression-level (static) staging.

(iv) Its strengths are in reducing the overhead of dynamic staging as well as retaining simple EDSL interfaces that hide their staged nature.

(v) Its shortcomings were identified to lie in the crude way of specifying language membership and its problematic implications on reliability and documentation aspects.

(vi) Tame staging was proposed as a concrete refinement of the ideas of implicit staging (and its prototype), enabling dynamic staging and introducing clarity to the API by means of a language-membership annotation (`@Stage`).

(vii) Out of the box, tame staging offers support for mixing the usage of several EDSLs. Safeness and uniformness are established by enforcing language boundaries with predictable behavior where they occur.

(viii) Both indirection of processing (i.e. closure caching) and static-staging aspects (i.e. stage-graph analyses, constants, and static-context retention) are used to address the concerns of redundant processing as well as debugging in the latter case.

(ix) Both implicit-staging as well as tame-staging implementations were designed and built as pragmatic extensions to Java, by leaving the user-code compilation process untouched and deferring to load-time adaptations.

I believe that there is merit in considering embedded (domain-specific) languages as first-class citizens from the beginning in future languages. The presented pragmatic extensions illustrate this and can serve as a testbed to thoroughly explore and promote this proposition.

5.1 Limitations

The previous discussions and the comparison of tame staging, as the final result of the present investigation, with traditional and recently emerged approaches indicate that it is a very user-friendly solution for providing EDSLs in the role of rich high-level library interfaces. Figure 5.1 shows a three-dimensional placement of the approaches introduced and compared in the previous chapters.

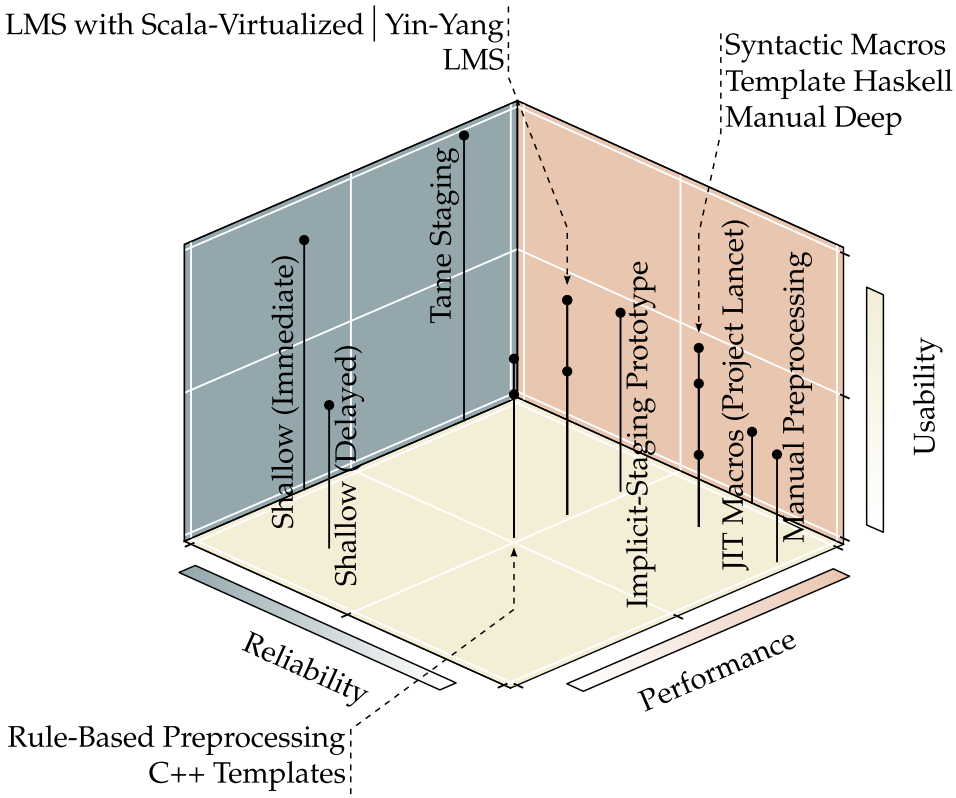


FIGURE 5.1: Three-dimensional placement of embedding approaches

By now, the reader should be aware that the focus of this thesis lies on the overall experience for EDSL users, i.e. programmers employing an EDSL as part of their software-development process, in terms of the three criteria or dimensions of reliability, performance, and (direct) usability as discussed in Chapter 2. The comparison of approaches is to be understood in light of this.

As part of this focus some concerns were left behind or were not sufficiently addressed in my research. For instance, composing several embedded languages in terms of both syntax and semantics to form a new one was only marginally considered during the design of the presented pragmatic extensions. Lightweight Modular Staging (LMS) and tagless-final embedding in general are much better suited for that. Properly designed, both EDSL interfaces (i.e. syntax) as well as tagless interpreters (i.e. semantics) can be combined, reused, and extended. This form of modularity is a desirable property for EDSL authors as it helps reduce implementation effort.

Of course, this may be considered a back-end problem that could be addressed in tame staging by an additional mechanism or methodology for implementing EDSL-program processors, i.e. compilers. On the front end, `@Stage` and `@Accept` annotations could be used to provide and reuse EDSL tokens. However, there is not yet a general plan or concept to consolidate and refine these ideas.

EDSL authors do still profit from using tame staging over manual deep embedding since the automation of staging and caching take off some of the burden usually encountered, especially in a verbose host language such as Java. However, this is not entirely without drawbacks. The expression-DAG data structure may be considered by some to be overly generic. For instance, in some cases expression traversal can be more cumbersome than in a manual, specific deep-embedding implementation, not to forget that the requirement to implement input-value-independent compilers could be seen by authors as an unnecessary complication (despite its argued necessity for reducing redundant processing). This is exacerbated by the potential run-time overhead experienced in some use cases (with the current framework implementation).

One major contribution of this thesis is the provision of staged-EDSL support through pragmatic extensions that rely on load-time metaprogramming. Load-time metaprogramming depends on the features of the host-language runtime (e.g. JVM). This is better than relying on compiler extensions not only in the interest of software-evolution concerns. After all, the bytecode format is commonly not undergoing changes as frequently as the language front end.

However, the presented pragmatic extensions demand a close relationship between the compiler and the language runtime. For instance, while in principle the tame-staging framework also works (as is) in Scala, using it effectively may sometimes necessitate the understanding and consideration of Scala's compilation strategies (e.g. seemingly local variables may become heap references when escaping to anonymous-function bodies in the same scope) especially in the presence of advanced features such as user-defined implicit conversions.

The mentioned limitations might call into question the adoption prospective of an approach like tame staging in its present form. However, I am hopeful that the advantages and future potential of a linguistic abstraction far outweigh the current shortcomings and issues found in the prototypical nature of my exploration.

5.2 Future Work

There is certainly room for further improvement. For one, abstraction overhead deserves strong attention. In the short term, efforts should be concentrated on this in order to highlight more clearly the advantages of being able to exploit the meta-level ability to detect and handle domain-specific code. The present tame-staging framework implementation does not yet make full use of this. For instance, although in cases where an EDSL program's shape is statically known permanent caching reduces some of the redundant processing, staging still occurs dynamically as usual and the input-value environment is also not specialized. These are two main contributors to run-time overhead that could be tackled by a more elaborate back-end implementation. The implicit-staging prototype of Chapter 3 trivially

addresses this concern (by design). However bringing this functionality to the dynamic-staging realm of tame staging is somewhat involved.

It is also worthwhile to identify supplementary features and refinements to the interface of the tame-staging framework. For instance, it would be interesting to investigate what kind of options for the inlining of staged terms or (instead) the reuse of already materialized values (if present) could be provided. This may prove to be a useful tool to improve performance when locally carrying, e.g. the `ImmutableList`-processing EDSL of Chapter 1 and Section 4.3.3 in Chapter 4.

There may also be cases where staging could reuse term-representation nodes for common-subexpression elimination. Likewise, an author may want to decide that materialization ought to be automatically triggered due to external circumstances (i.e. ones not relating to value consumption) such as the exceeding of a threshold for expression-DAG size. These and likely many more such options could be considered in the future. However, at the same time it is crucial not to unnecessarily impair the ease of understanding and predicting behaviors. An excess amount of options may turn into a burden on the mental load of EDSL users and authors alike. Controlled user studies as well as observing the potential adoption and usage of the tame-staging framework in production may give further hints on how useful the linguistic abstraction is and what use cases could warrant further extension.

As elaborated at the end of Chapter 4, the next big leap outside the scope of this thesis would be to go beyond pragmatic extensions, in other words, to turn the current *almost* first-class feature into an *actual*, native first-class feature in future versions of general-purpose host languages. After all, dedicated support for staged EDSLs with appropriate usage guarantees and inbuilt checks and warnings could be a good alternative to more unrestricted meta-programming approaches like syntactic macros, which language designers may shy away from due to reliability concerns.

APPENDIX A

Matrix EDSL Experiment Expressions

Matrix values are contained in variables starting with `m` and **`double`** values are contained in variables starting with `d`. The `dim` variable contains the dimension value `8`. Duplicates were not excluded from the benchmark (nor from the lists here).

LISTING A.1: Matrix EDSL benchmark loops

```
Matrix res = null;
long start;
long end;
…
for (int i = 0; i < WARMUP_REPS; i++) {
  res = e_i^d;
}
System.gc();
try { Thread.sleep(2000); } catch (… e) { e.printStackTrace(); }

start = System.nanoTime();
for (int i = 0; i < BENCHMARK_REPS; i++) {
  res = e_i^d;
}
end = System.nanoTime();

resultArray[…][…] = res;
timeArray[…][…]   = end - start;
expArray[…][…]    = "e_i^d";
…
```

Note that the expressions are merely listed but not shown within their respective warm-up and measurement loops here. The template for such loops is shown in Listing A.1 with e_i^d denoting the i-th expression of depth d.

A.1 Randomly Generated Expression

A.1.1 Shallow Embedding and Implicit Staging

A.1.1.1 Depth 1 (Random, Shallow)

```
1  add(m1, sca(dim, d1))
2  mul(sca(dim, d1), m1)
3  mul(sca(dim, d1), m0)
4  add(m0, m1)
5  mul(m0, m1)
6  add(m1, m0)
7  add(m1, m0)
8  mul(m0, sca(dim, d0))
9  mul(sca(dim, d0), m1)
10 add(sca(dim, d1), m1)
11 add(m1, sca(dim, d1))
12 mul(sca(dim, d0), m0)
13 add(sca(dim, d0), sca(dim, d1))
14 add(sca(dim, d1), m0)
15 add(sca(dim, d1), sca(dim, d1))
16 add(sca(dim, d0), sca(dim, d0))
17 mul(m0, sca(dim, d1))
18 add(sca(dim, d1), sca(dim, d0))
19 add(m0, sca(dim, d0))
20 add(m0, m1)
21 add(sca(dim, d1), m1)
22 add(m1, m1)
23 mul(m1, m1)
24 add(sca(dim, d1), m1)
25 add(sca(dim, d1), sca(dim, d0))
26 add(sca(dim, d0), sca(dim, d0))
27 add(sca(dim, d1), sca(dim, d0))
28 mul(m0, sca(dim, d1))
29 add(sca(dim, d1), sca(dim, d1))
30 add(m0, sca(dim, d0))
```

A.1.1.2 Depth 2 (Random, Shallow)

```
1  add(add(sca(dim, d3), sca(dim, d4)), mul(m2, sca(dim, d3)))
2  add(add(sca(dim, d2), m2), mul(m3, sca(dim, d4)))
3  mul(add(m4, m1), m0)
4  add(mul(sca(dim, d2), sca(dim, d4)), mul(m3, m0))
5  add(mul(m1, sca(dim, d4)), m3)
6  add(add(sca(dim, d1), sca(dim, d1)), add(sca(dim, d1), sca(dim, d4)))
7  mul(mul(sca(dim, d4), m4), add(m3, m1))
8  mul(mul(sca(dim, d2), m4), sca(dim, d1))
9  add(mul(m2, sca(dim, d0)), add(sca(dim, d3), sca(dim, d1)))
10 mul(mul(sca(dim, d2), m0), sca(dim, d2))
11 mul(mul(sca(dim, d2), m3), mul(m0, m0))
12 mul(mul(sca(dim, d3), m0), mul(sca(dim, d1), sca(dim, d4)))
13 add(add(m1, sca(dim, d0)), sca(dim, d0))
14 mul(add(sca(dim, d2), m4), sca(dim, d0))
15 mul(mul(sca(dim, d2), sca(dim, d1)), add(sca(dim, d4), sca(dim, d0)))
16 add(add(sca(dim, d2), m0), mul(m1, sca(dim, d0)))
17 mul(add(m1, m1), mul(m2, sca(dim, d0)))
18 mul(add(m4, m3), m3)
```

```
19 mul(mul(sca(dim, d1), sca(dim, d3)), m4)
20 mul(mul(m3, m4), mul(m1, sca(dim, d4)))
21 mul(mul(m2, sca(dim, d0)), add(m2, sca(dim, d0)))
22 add(mul(sca(dim, d2), m1), sca(dim, d3))
23 mul(add(m3, m1), mul(m1, m4))
24 mul(mul(sca(dim, d1), sca(dim, d0)), add(m2, sca(dim, d0)))
25 mul(mul(sca(dim, d3), sca(dim, d4)), add(m4, sca(dim, d2)))
26 mul(mul(sca(dim, d1), sca(dim, d0)), sca(dim, d4))
27 add(mul(m4, m2), add(m1, m1))
28 mul(add(sca(dim, d0), m2), m2)
29 add(mul(sca(dim, d2), sca(dim, d1)), m2)
30 mul(add(m0, m3), m3)
```

A.1.1.3 Depth 3 (Random, Shallow)

```
1  add(mul(mul(sca(dim, d2), m7), m8), sca(dim, d4))
2  mul(add(mul(m2, m8), mul(sca(dim, d5), m2)), sca(dim, d7))
3  add(add(mul(m5, m4), m4), add(m0, add(m8, sca(dim, d1))))
4  add(mul(add(sca(dim, d1), m5), mul(m9, m5)), add(add(sca(dim, d7),
       ↪ sca(dim, d3)), m5))
5  add(mul(mul(m6, m6), m8), mul(mul(sca(dim, d5), sca(dim, d6)),
       ↪ mul(sca(dim, d3), sca(dim, d1))))
6  mul(mul(add(sca(dim, d2), sca(dim, d0)), m7), sca(dim, d3))
7  mul(mul(mul(m4, m0), add(m5, sca(dim, d2))), m2)
8  add(add(mul(m2, sca(dim, d9)), sca(dim, d1)), sca(dim, d8))
9  add(mul(mul(m4, sca(dim, d0)), mul(m7, sca(dim, d7))), add(sca(dim, d5),
       ↪ m3))
10 mul(mul(mul(m4, sca(dim, d0)), mul(sca(dim, d7), sca(dim, d2))), m5)
11 add(mul(mul(sca(dim, d2), m7), add(sca(dim, d6), m8)), mul(sca(dim, d9),
       ↪ mul(sca(dim, d1), m0)))
12 add(mul(add(m4, sca(dim, d2)), mul(m2, m7)), add(mul(m1, m3), m0))
13 mul(add(add(sca(dim, d7), m9), m3), m7)
14 add(add(mul(m9, sca(dim, d2)), mul(sca(dim, d1), m1)), m5)
15 mul(mul(add(sca(dim, d9), m2), add(sca(dim, d0), m7)), sca(dim, d6))
16 mul(add(add(m2, m7), sca(dim, d7)), sca(dim, d4))
17 mul(mul(add(sca(dim, d8), sca(dim, d4)), add(m1, m3)), m5)
18 mul(add(add(m0, m2), sca(dim, d3)), sca(dim, d2))
19 add(mul(add(sca(dim, d1), sca(dim, d2)), mul(m4, sca(dim, d8))), m2)
20 mul(add(mul(sca(dim, d8), sca(dim, d9)), m6), m3)
21 add(add(add(sca(dim, d7), sca(dim, d1)), add(m3, m0)), add(add(m0, m5),
       ↪ mul(m5, m1)))
22 mul(mul(add(sca(dim, d5), sca(dim, d7)), sca(dim, d7)), sca(dim, d0))
23 mul(add(add(sca(dim, d5), sca(dim, d4)), m6), sca(dim, d1))
24 add(mul(add(sca(dim, d7), sca(dim, d9)), add(m0, m0)), add(sca(dim, d4),
       ↪ sca(dim, d9)))
25 add(add(mul(sca(dim, d7), sca(dim, d4)), add(m3, sca(dim, d7))),
       ↪ mul(mul(sca(dim, d9), m4), m2))
26 add(mul(mul(sca(dim, d7), sca(dim, d1)), sca(dim, d5)), mul(mul(m6, m0),
       ↪ sca(dim, d7)))
27 add(add(add(sca(dim, d9), sca(dim, d5)), sca(dim, d9)), sca(dim, d0))
28 add(add(add(sca(dim, d5), m1), add(sca(dim, d6), m8)), m4)
29 mul(add(add(sca(dim, d6), sca(dim, d1)), add(sca(dim, d7), m0)),
       ↪ sca(dim, d9))
30 add(add(add(m7, sca(dim, d3)), add(sca(dim, d6), sca(dim, d2))),
       ↪ mul(mul(sca(dim, d4), sca(dim, d2)), m3))
```

A.1.1.4 Depth 4 (Random, Shallow)

```
1  mul(add(add(mul(m12, m8), m4), m7), add(add(m15, add(m2, sca(dim, d3))),
       ↪ m5))
2  add(mul(mul(add(m0, m10), add(m9, sca(dim, d0))), add(sca(dim, d0),
       ↪ sca(dim, d10))), add(sca(dim, d14), sca(dim, d11)))
3  add(add(mul(add(m15, sca(dim, d9)), mul(sca(dim, d9), sca(dim, d12))),
       ↪ m4), mul(mul(add(m15, sca(dim, d1)), mul(sca(dim, d8), m1)), m3))
4  add(mul(mul(add(sca(dim, d10), m8), add(sca(dim, d2), sca(dim, d1))),
       ↪ add(add(sca(dim, d16), m5), sca(dim, d13))), add(add(sca(dim,
       ↪ d0), add(m5, sca(dim, d15))), add(sca(dim, d3), mul(m0, sca(dim,
       ↪ d7)))))
5  add(add(mul(add(m13, sca(dim, d0)), mul(m12, m11)), add(m10, sca(dim,
       ↪ d14))), add(sca(dim, d0), m11))
6  add(add(add(mul(sca(dim, d13), sca(dim, d1)), mul(m2, sca(dim, d6))),
       ↪ add(add(m15, sca(dim, d12)), mul(sca(dim, d10), m6))),
       ↪ mul(mul(sca(dim, d11), sca(dim, d10)), m1))
7  mul(add(add(add(sca(dim, d11), m12), mul(sca(dim, d15), sca(dim, d5))),
       ↪ m14), m7)
8  add(mul(add(add(m5, sca(dim, d11)), mul(m0, sca(dim, d12))), m10),
       ↪ mul(mul(mul(sca(dim, d4), sca(dim, d9)), mul(sca(dim, d5), m9)),
       ↪ m7))
9  mul(mul(mul(mul(m0, sca(dim, d10)), sca(dim, d5)), add(mul(m0, m5),
       ↪ add(sca(dim, d3), m10))), add(sca(dim, d16), mul(add(m6, sca(dim,
       ↪ d14)), m5)))
10 mul(mul(mul(add(m2, m9), mul(m3, sca(dim, d0))), sca(dim, d11)), add(m5,
       ↪ m13))
11 add(mul(mul(add(m5, m14), add(sca(dim, d10), sca(dim, d7))), m6), m3)
12 mul(add(add(mul(sca(dim, d12), sca(dim, d5)), add(m0, m6)),
       ↪ mul(mul(sca(dim, d10), m7), add(sca(dim, d9), m3))), add(m5, m8))
13 add(add(add(add(sca(dim, d6), m5), mul(sca(dim, d5), sca(dim, d4))),
       ↪ m7), add(m2, mul(add(sca(dim, d12), m11), add(m10, sca(dim,
       ↪ d5)))))
14 add(add(mul(mul(m0, sca(dim, d8)), sca(dim, d1)), mul(mul(sca(dim, d2),
       ↪ m13), sca(dim, d16))), add(sca(dim, d11), sca(dim, d10)))
15 add(mul(add(mul(sca(dim, d0), m12), sca(dim, d13)), m4), add(sca(dim,
       ↪ d1), mul(m6, mul(m9, sca(dim, d12)))))
16 add(add(add(mul(m3, m10), mul(sca(dim, d2), m2)), sca(dim, d10)),
       ↪ add(mul(mul(sca(dim, d8), m0), mul(sca(dim, d1), sca(dim, d9))),
       ↪ mul(mul(m3, m0), mul(sca(dim, d15), m5))))
17 mul(add(add(mul(sca(dim, d14), sca(dim, d12)), mul(sca(dim, d6), m12)),
       ↪ mul(mul(m15, m5), m9)), mul(mul(sca(dim, d8), add(sca(dim, d12),
       ↪ sca(dim, d13))), sca(dim, d7)))
18 add(mul(add(mul(m8, sca(dim, d3)), add(sca(dim, d10), m14)),
       ↪ mul(add(m12, sca(dim, d5)), mul(m9, m7))), add(mul(add(sca(dim,
       ↪ d11), m16), sca(dim, d13)), add(sca(dim, d8), sca(dim, d5))))
19 add(mul(mul(add(sca(dim, d11), sca(dim, d6)), add(m3, sca(dim, d5))),
       ↪ add(m7, sca(dim, d3))), m7)
20 add(add(mul(mul(sca(dim, d4), m2), sca(dim, d11)), add(m16, mul(sca(dim,
       ↪ d9), sca(dim, d7)))), sca(dim, d14))
21 mul(add(mul(mul(sca(dim, d1), sca(dim, d7)), sca(dim, d1)), sca(dim,
       ↪ d10)), mul(add(m2, sca(dim, d6)), mul(add(m16, m9), add(m15,
       ↪ m12))))
22 mul(mul(mul(mul(m4, sca(dim, d9)), m16), mul(add(sca(dim, d0), m4),
       ↪ mul(sca(dim, d15), sca(dim, d1)))), mul(mul(add(sca(dim, d11),
       ↪ m6), mul(sca(dim, d1), sca(dim, d13))), m15))
23 add(add(mul(mul(m8, sca(dim, d9)), sca(dim, d2)), sca(dim, d4)),
       ↪ add(add(sca(dim, d15), m2), mul(sca(dim, d10), sca(dim, d0))))
```

```
24 mul(mul(add(mul(sca(dim, d12), sca(dim, d6)), mul(m14, m8)), m11), m2)
25 mul(add(add(add(m9, sca(dim, d16)), add(m10, sca(dim, d7))), m15),
       ↪ sca(dim, d7))
26 mul(mul(add(add(m3, sca(dim, d5)), mul(m5, sca(dim, d14))), sca(dim,
       ↪ d7)), add(mul(mul(sca(dim, d9), m2), sca(dim, d11)), add(m16,
       ↪ add(sca(dim, d3), sca(dim, d9)))))
27 add(mul(mul(add(m4, m3), add(sca(dim, d12), sca(dim, d16))),
       ↪ mul(sca(dim, d8), sca(dim, d3))), add(add(add(m6, m7), mul(m4,
       ↪ sca(dim, d5))), add(sca(dim, d16), mul(m4, sca(dim, d13)))))
28 add(mul(add(add(sca(dim, d5), sca(dim, d13)), sca(dim, d10)),
       ↪ add(add(m11, sca(dim, d0)), sca(dim, d12))), add(m11, sca(dim,
       ↪ d3)))
29 add(mul(mul(mul(m12, m4), add(m11, m12)), mul(add(sca(dim, d15), m6),
       ↪ add(m5, sca(dim, d4)))), sca(dim, d16))
30 mul(mul(add(mul(m4, m6), sca(dim, d6)), add(add(m9, m13), sca(dim,
       ↪ d16))), add(mul(add(m2, m6), sca(dim, d13)), mul(mul(sca(dim,
       ↪ d8), sca(dim, d7)), add(sca(dim, d7), m4))))
```

A.1.1.5 Depth 5 (Random, Shallow)

```
1  mul(add(mul(add(add(m18, m0), m21), m9), sca(dim, d0)), add(m11,
       ↪ sca(dim, d17)))
2  mul(add(add(add(add(m4, sca(dim, d14)), m8), add(add(m5, m19), sca(dim,
       ↪ d2))), add(m22, mul(add(m0, sca(dim, d16)), sca(dim, d13)))),
       ↪ mul(m10, sca(dim, d5)))
3  mul(mul(mul(mul(add(m24, sca(dim, d22)), add(m16, sca(dim, d3))), m17),
       ↪ sca(dim, d24)), m14)
4  add(mul(mul(add(add(sca(dim, d12), sca(dim, d14)), mul(m0, sca(dim,
       ↪ d1))), mul(mul(sca(dim, d13), sca(dim, d22)), sca(dim, d22))),
       ↪ sca(dim, d19)), sca(dim, d24))
5  add(mul(add(mul(mul(m17, sca(dim, d5)), sca(dim, d25)), sca(dim, d1)),
       ↪ add(sca(dim, d10), m23)), mul(sca(dim, d1), add(sca(dim, d6),
       ↪ m9)))
6  add(add(mul(add(add(m21, sca(dim, d14)), m17), mul(mul(sca(dim, d20),
       ↪ m4), sca(dim, d21))), mul(m4, sca(dim, d6))), mul(sca(dim, d19),
       ↪ m16))
7  add(add(add(add(add(m14, sca(dim, d1)), add(sca(dim, d14), m23)),
       ↪ add(add(m12, m13), mul(sca(dim, d6), m25))), sca(dim, d1)),
       ↪ add(add(sca(dim, d12), mul(add(sca(dim, d19), m2), mul(m22,
       ↪ m7))), m14))
8  add(mul(mul(mul(mul(m1, m22), m3), m11), sca(dim, d22)), mul(sca(dim,
       ↪ d3), mul(add(sca(dim, d23), m2), sca(dim, d2))))
9  mul(add(mul(mul(mul(sca(dim, d5), sca(dim, d23)), mul(sca(dim, d15),
       ↪ m5)), m15), add(sca(dim, d17), sca(dim, d14))), sca(dim, d20))
10 add(mul(add(mul(mul(sca(dim, d23), sca(dim, d25)), mul(m11, sca(dim,
       ↪ d1))), sca(dim, d22)), mul(m8, add(mul(m5, m7), sca(dim, d15)))),
       ↪ sca(dim, d7))
11 mul(mul(mul(mul(mul(m0, sca(dim, d19)), add(m25, sca(dim, d7))),
       ↪ mul(m17, sca(dim, d23))), m12), sca(dim, d2))
12 add(mul(mul(mul(mul(m16, m6), add(sca(dim, d14), sca(dim, d23))),
       ↪ sca(dim, d23)), add(add(mul(sca(dim, d16), m21), m18), sca(dim,
       ↪ d12))), mul(add(sca(dim, d7), m25), sca(dim, d13)))
13 add(add(mul(mul(mul(m21, m1), add(sca(dim, d25), sca(dim, d19))),
       ↪ add(m17, sca(dim, d11))), sca(dim, d8)), sca(dim, d25))
14 add(mul(add(add(mul(m23, sca(dim, d22)), mul(m25, m3)), mul(m9, sca(dim,
       ↪ d12))), sca(dim, d20)), m11)
```

```
15 add(add(add(mul(mul(m14, m22), sca(dim, d21)), mul(sca(dim, d13),
       ↪ add(sca(dim, d19), m1))), mul(add(sca(dim, d3), mul(sca(dim,
       ↪ d25), m17)), mul(add(sca(dim, d8), sca(dim, d7)), add(sca(dim,
       ↪ d13), sca(dim, d0))))), mul(sca(dim, d22), add(mul(sca(dim, d25),
       ↪ m12), sca(dim, d9))))
16 mul(add(mul(mul(add(sca(dim, d23), m5), sca(dim, d9)), add(mul(sca(dim,
       ↪ d7), sca(dim, d16)), sca(dim, d2))), m4), add(mul(sca(dim, d1),
       ↪ sca(dim, d4)), add(mul(mul(sca(dim, d2), m10), add(m15, sca(dim,
       ↪ d0))), sca(dim, d17))))
17 add(mul(add(add(mul(sca(dim, d24), sca(dim, d4)), sca(dim, d8)),
       ↪ add(mul(m24, sca(dim, d2)), sca(dim, d4))), sca(dim, d14)),
       ↪ add(sca(dim, d25), mul(add(add(sca(dim, d17), m10), add(sca(dim,
       ↪ d5), m5)), mul(mul(m18, sca(dim, d25)), mul(sca(dim, d5),
       ↪ sca(dim, d15))))))
18 add(mul(mul(mul(mul(m14, m20), mul(m10, sca(dim, d24))), m13), mul(m2,
       ↪ mul(mul(m18, sca(dim, d9)), add(sca(dim, d0), m4))))), m15)
19 mul(mul(mul(mul(add(m13, m0), add(m2, m6)), sca(dim, d19)), add(mul(m11,
       ↪ m0), add(m0, add(m3, m1)))), mul(m11, mul(sca(dim, d23), mul(m19,
       ↪ add(m12, sca(dim, d6))))))
20 mul(mul(mul(add(mul(m3, m0), m16), m4), add(sca(dim, d21), m6)),
       ↪ mul(mul(mul(mul(sca(dim, d25), m14), m18), sca(dim, d5)),
       ↪ mul(sca(dim, d0), add(sca(dim, d18), add(sca(dim, d19), sca(dim,
       ↪ d9))))))
21 add(add(add(add(mul(m11, sca(dim, d4)), mul(m25, sca(dim, d24))),
       ↪ sca(dim, d12)), sca(dim, d6)), mul(m1, sca(dim, d20)))
22 mul(mul(add(add(add(m23, m4), sca(dim, d0)), mul(sca(dim, d3),
       ↪ mul(sca(dim, d14), m15))), sca(dim, d1)), sca(dim, d23))
23 mul(add(mul(mul(add(sca(dim, d17), sca(dim, d24)), add(m15, m13)),
       ↪ mul(sca(dim, d20), m21)), m9), sca(dim, d1))
24 add(mul(mul(mul(mul(sca(dim, d25), m7), add(sca(dim, d0), m25)),
       ↪ add(add(m21, m21), mul(sca(dim, d6), sca(dim, d20)))),
       ↪ mul(mul(mul(m18, m7), m7), m0)), mul(sca(dim, d1), mul(sca(dim,
       ↪ d4), m20)))
25 mul(add(add(mul(add(m7, m7), m10), sca(dim, d7)), sca(dim, d19)),
       ↪ sca(dim, d2))
26 add(mul(mul(add(mul(m23, m12), add(sca(dim, d4), sca(dim, d1))), add(m3,
       ↪ mul(m5, sca(dim, d25)))), mul(add(add(sca(dim, d21), m13),
       ↪ mul(sca(dim, d5), m17)), m23)), sca(dim, d21))
27 mul(add(mul(add(add(sca(dim, d0), sca(dim, d16)), add(sca(dim, d3),
       ↪ m17)), mul(mul(sca(dim, d10), sca(dim, d5)), m1)), mul(sca(dim,
       ↪ d9), m23)), sca(dim, d5))
28 mul(mul(mul(add(mul(m23, m3), sca(dim, d14)), mul(add(sca(dim, d7), m4),
       ↪ m24)), mul(add(mul(sca(dim, d4), m20), sca(dim, d8)), sca(dim,
       ↪ d23))), sca(dim, d21))
29 mul(mul(add(add(add(sca(dim, d8), sca(dim, d1)), sca(dim, d1)), m1),
       ↪ add(sca(dim, d7), sca(dim, d9))), m11)
30 mul(mul(mul(mul(add(m9, m14), add(sca(dim, d0), m15)), add(sca(dim,
       ↪ d17), sca(dim, d3))), add(mul(sca(dim, d19), m1), sca(dim, d0))),
       ↪ sca(dim, d17))
```

A.1.2 Deep Embedding

A.1.2.1 Depth 1 (Random, Deep)

```
1 add(lit(m1), sca(dim, d1)).eval()
2 mul(sca(dim, d1), lit(m1)).eval()
3 mul(sca(dim, d1), lit(m0)).eval()
```

```
4  add(lit(m0), lit(m1)).eval()
5  mul(lit(m0), lit(m1)).eval()
6  add(lit(m1), lit(m0)).eval()
7  add(lit(m1), lit(m0)).eval()
8  mul(lit(m0), sca(dim, d0)).eval()
9  mul(sca(dim, d0), lit(m1)).eval()
10 add(sca(dim, d1), lit(m1)).eval()
11 add(lit(m1), sca(dim, d1)).eval()
12 mul(sca(dim, d0), lit(m0)).eval()
13 add(sca(dim, d0), sca(dim, d1)).eval()
14 add(sca(dim, d1), lit(m0)).eval()
15 add(sca(dim, d1), sca(dim, d1)).eval()
16 add(sca(dim, d0), sca(dim, d0)).eval()
17 mul(lit(m0), sca(dim, d1)).eval()
18 add(sca(dim, d1), sca(dim, d0)).eval()
19 add(lit(m0), sca(dim, d0)).eval()
20 add(lit(m0), lit(m1)).eval()
21 add(sca(dim, d1), lit(m1)).eval()
22 add(lit(m1), lit(m1)).eval()
23 mul(lit(m1), lit(m1)).eval()
24 add(sca(dim, d1), lit(m1)).eval()
25 add(sca(dim, d1), sca(dim, d0)).eval()
26 add(sca(dim, d0), sca(dim, d0)).eval()
27 add(sca(dim, d1), sca(dim, d0)).eval()
28 mul(lit(m0), sca(dim, d1)).eval()
29 add(sca(dim, d1), sca(dim, d1)).eval()
30 add(lit(m0), sca(dim, d0)).eval()
```

A.1.2.2 Depth 2 (Random, Deep)

```
1  add(add(sca(dim, d3), sca(dim, d4)), mul(lit(m2), sca(dim, d3))).eval()
2  add(add(sca(dim, d2), lit(m2)), mul(lit(m3), sca(dim, d4))).eval()
3  mul(add(lit(m4), lit(m1)), lit(m0)).eval()
4  add(mul(sca(dim, d2), sca(dim, d4)), mul(lit(m3), lit(m0))).eval()
5  add(mul(lit(m1), sca(dim, d4)), lit(m3)).eval()
6  add(add(sca(dim, d1), sca(dim, d1)), add(sca(dim, d1), sca(dim,
       ↪ d4))).eval()
7  mul(mul(sca(dim, d4), lit(m4)), add(lit(m3), lit(m1))).eval()
8  mul(mul(sca(dim, d2), lit(m4)), sca(dim, d1)).eval()
9  add(mul(lit(m2), sca(dim, d0)), add(sca(dim, d3), sca(dim, d1))).eval()
10 mul(mul(sca(dim, d2), lit(m0)), sca(dim, d2)).eval()
11 mul(mul(sca(dim, d2), lit(m3)), mul(lit(m0), lit(m0))).eval()
12 mul(mul(sca(dim, d3), lit(m0)), mul(sca(dim, d1), sca(dim, d4))).eval()
13 add(add(lit(m1), sca(dim, d0)), sca(dim, d0)).eval()
14 mul(add(sca(dim, d2), lit(m4)), sca(dim, d0)).eval()
15 mul(mul(sca(dim, d2), sca(dim, d1)), add(sca(dim, d4), sca(dim,
       ↪ d0))).eval()
16 add(add(sca(dim, d2), lit(m0)), mul(lit(m1), sca(dim, d0))).eval()
17 mul(add(lit(m1), lit(m1)), mul(lit(m2), sca(dim, d0))).eval()
18 mul(add(lit(m4), lit(m3)), lit(m3)).eval()
19 mul(mul(sca(dim, d1), sca(dim, d3)), lit(m4)).eval()
20 mul(mul(lit(m3), lit(m4)), mul(lit(m1), sca(dim, d4))).eval()
21 mul(mul(lit(m2), sca(dim, d0)), add(lit(m2), sca(dim, d0))).eval()
22 add(mul(sca(dim, d2), lit(m1)), sca(dim, d3)).eval()
23 mul(add(lit(m3), lit(m1)), mul(lit(m1), lit(m4))).eval()
24 mul(mul(sca(dim, d1), sca(dim, d0)), add(lit(m2), sca(dim, d0))).eval()
25 mul(mul(sca(dim, d3), sca(dim, d4)), add(lit(m4), sca(dim, d2))).eval()
```

```
26 mul(mul(sca(dim, d1), sca(dim, d0)), sca(dim, d4)).eval()
27 add(mul(lit(m4), lit(m2)), add(lit(m1), lit(m1))).eval()
28 mul(add(sca(dim, d0), lit(m2)), lit(m2)).eval()
29 add(mul(sca(dim, d2), sca(dim, d1)), lit(m2)).eval()
30 mul(add(lit(m0), lit(m3)), lit(m3)).eval()
```

A.1.2.3 Depth 3 (Random, Deep)

```
1  add(mul(mul(sca(dim, d2), lit(m7)), lit(m8)), sca(dim, d4)).eval()
2  mul(add(mul(lit(m2), lit(m8)), mul(sca(dim, d5), lit(m2))), sca(dim,
       ↪ d7)).eval()
3  add(add(mul(lit(m5), lit(m4)), lit(m4)), add(lit(m0), add(lit(m8),
       ↪ sca(dim, d1)))).eval()
4  add(mul(add(sca(dim, d1), lit(m5)), mul(lit(m9), lit(m5))),
       ↪ add(add(sca(dim, d7), sca(dim, d3)), lit(m5))).eval()
5  add(mul(mul(lit(m6), lit(m6)), lit(m8)), mul(mul(sca(dim, d5), sca(dim,
       ↪ d6)), mul(sca(dim, d3), sca(dim, d1)))).eval()
6  mul(mul(add(sca(dim, d2), sca(dim, d0)), lit(m7)), sca(dim, d3)).eval()
7  mul(mul(mul(lit(m4), lit(m0)), add(lit(m5), sca(dim, d2))),
       ↪ lit(m2)).eval()
8  add(add(mul(lit(m2), sca(dim, d9)), sca(dim, d1)), sca(dim, d8)).eval()
9  add(mul(mul(lit(m4), sca(dim, d0)), mul(lit(m7), sca(dim, d7))),
       ↪ add(sca(dim, d5), lit(m3))).eval()
10 mul(mul(mul(lit(m4), sca(dim, d0)), mul(sca(dim, d7), sca(dim, d2))),
       ↪ lit(m5)).eval()
11 add(mul(mul(sca(dim, d2), lit(m7)), add(sca(dim, d6), lit(m8))),
       ↪ mul(sca(dim, d9), mul(sca(dim, d1), lit(m0)))).eval()
12 add(mul(add(lit(m4), sca(dim, d2)), mul(lit(m2), lit(m7))),
       ↪ add(mul(lit(m1), lit(m3)), lit(m0))).eval()
13 mul(add(add(sca(dim, d7), lit(m9)), lit(m3)), lit(m7)).eval()
14 add(add(mul(lit(m9), sca(dim, d2)), mul(sca(dim, d1), lit(m1))),
       ↪ lit(m5)).eval()
15 mul(mul(add(sca(dim, d9), lit(m2)), add(sca(dim, d0), lit(m7))),
       ↪ sca(dim, d6)).eval()
16 mul(add(add(lit(m2), lit(m7)), sca(dim, d7)), sca(dim, d4)).eval()
17 mul(mul(add(sca(dim, d8), sca(dim, d4)), add(lit(m1), lit(m3))),
       ↪ lit(m5)).eval()
18 mul(add(add(lit(m0), lit(m2)), sca(dim, d3)), sca(dim, d2)).eval()
19 add(mul(add(sca(dim, d1), sca(dim, d2)), mul(lit(m4), sca(dim, d8))),
       ↪ lit(m2)).eval()
20 mul(add(mul(sca(dim, d8), sca(dim, d9)), lit(m6)), lit(m3)).eval()
21 add(add(add(sca(dim, d7), sca(dim, d1)), add(lit(m3), lit(m0))),
       ↪ add(add(lit(m0), lit(m5)), mul(lit(m5), lit(m1)))).eval()
22 mul(mul(add(sca(dim, d5), sca(dim, d7)), sca(dim, d7)), sca(dim,
       ↪ d0)).eval()
23 mul(add(add(sca(dim, d5), sca(dim, d4)), lit(m6)), sca(dim, d1)).eval()
24 add(mul(add(sca(dim, d7), sca(dim, d9)), add(lit(m0), lit(m0))),
       ↪ add(sca(dim, d4), sca(dim, d9))).eval()
25 add(add(mul(sca(dim, d7), sca(dim, d4)), add(lit(m3), sca(dim, d7))),
       ↪ mul(mul(sca(dim, d9), lit(m4)), lit(m2))).eval()
26 add(mul(mul(sca(dim, d7), sca(dim, d1)), sca(dim, d5)), mul(mul(lit(m6),
       ↪ lit(m0)), sca(dim, d7))).eval()
27 add(add(add(sca(dim, d9), sca(dim, d5)), sca(dim, d9)), sca(dim,
       ↪ d0)).eval()
28 add(add(add(sca(dim, d5), lit(m1)), add(sca(dim, d6), lit(m8))),
       ↪ lit(m4)).eval()
```

```
29 mul(add(add(sca(dim, d6), sca(dim, d1)), add(sca(dim, d7), lit(m0))),
       ↪ sca(dim, d9)).eval()
30 add(add(add(lit(m7), sca(dim, d3)), add(sca(dim, d6), sca(dim, d2))),
       ↪ mul(mul(sca(dim, d4), sca(dim, d2)), lit(m3))).eval()
```

A.1.2.4 Depth 4 (Random, Deep)

```
1 mul(add(add(mul(lit(m12), lit(m8)), lit(m4)), lit(m7)),
      ↪ add(add(lit(m15), add(lit(m2), sca(dim, d3))), lit(m5))).eval()
2 add(mul(mul(add(lit(m0), lit(m10)), add(lit(m9), sca(dim, d0))),
      ↪ add(sca(dim, d0), sca(dim, d10))), add(sca(dim, d14), sca(dim,
      ↪ d11))).eval()
3 add(add(mul(add(lit(m15), sca(dim, d9)), mul(sca(dim, d9), sca(dim,
      ↪ d12))), lit(m4)), mul(mul(add(lit(m15), sca(dim, d1)),
      ↪ mul(sca(dim, d8), lit(m1))), lit(m3))).eval()
4 add(mul(mul(add(sca(dim, d10), lit(m8)), add(sca(dim, d2), sca(dim,
      ↪ d1))), add(add(sca(dim, d16), lit(m5)), sca(dim, d13))),
      ↪ add(add(sca(dim, d0), add(lit(m5), sca(dim, d15))), add(sca(dim,
      ↪ d3), mul(lit(m0), sca(dim, d7))))).eval()
5 add(add(mul(add(lit(m13), sca(dim, d0)), mul(lit(m12), lit(m11))),
      ↪ add(lit(m10), sca(dim, d14))), add(sca(dim, d0), lit(m11))).eval()
6 add(add(add(mul(sca(dim, d13), sca(dim, d1)), mul(lit(m2), sca(dim,
      ↪ d6))), add(add(lit(m15), sca(dim, d12)), mul(sca(dim, d10),
      ↪ lit(m6)))), mul(mul(sca(dim, d11), sca(dim, d10)),
      ↪ lit(m1))).eval()
7 mul(add(add(add(sca(dim, d11), lit(m12)), mul(sca(dim, d15), sca(dim,
      ↪ d5))), lit(m14)), lit(m7)).eval()
8 add(mul(add(add(lit(m5), sca(dim, d11)), mul(lit(m0), sca(dim, d12))),
      ↪ lit(m10)), mul(mul(mul(sca(dim, d4), sca(dim, d9)), mul(sca(dim,
      ↪ d5), lit(m9))), lit(m7))).eval()
9 mul(mul(mul(mul(lit(m0), sca(dim, d10)), sca(dim, d5)), add(mul(lit(m0),
      ↪ lit(m5)), add(sca(dim, d3), lit(m10)))), add(sca(dim, d16),
      ↪ mul(add(lit(m6), sca(dim, d14)), lit(m5)))).eval()
10 mul(mul(mul(add(lit(m2), lit(m9)), mul(lit(m3), sca(dim, d0))), sca(dim,
      ↪ d11)), add(lit(m5), lit(m13))).eval()
11 add(mul(mul(add(lit(m5), lit(m14)), add(sca(dim, d10), sca(dim, d7))),
      ↪ lit(m6)), lit(m3)).eval()
12 mul(add(add(mul(sca(dim, d12), sca(dim, d5)), add(lit(m0), lit(m6))),
      ↪ mul(mul(sca(dim, d10), lit(m7)), add(sca(dim, d9), lit(m3)))),
      ↪ add(lit(m5), lit(m8))).eval()
13 add(add(add(add(sca(dim, d6), lit(m5)), mul(sca(dim, d5), sca(dim,
      ↪ d4))), lit(m7)), add(lit(m2), mul(add(sca(dim, d12), lit(m11)),
      ↪ add(lit(m10), sca(dim, d5))))).eval()
14 add(add(mul(mul(lit(m0), sca(dim, d8)), sca(dim, d1)), mul(mul(sca(dim,
      ↪ d2), lit(m13)), sca(dim, d16))), add(sca(dim, d11), sca(dim,
      ↪ d10))).eval()
15 add(mul(add(mul(sca(dim, d0), lit(m12)), sca(dim, d13)), lit(m4)),
      ↪ add(sca(dim, d1), mul(lit(m6), mul(lit(m9), sca(dim,
      ↪ d12))))).eval()
16 add(add(add(mul(lit(m3), lit(m10)), mul(sca(dim, d2), lit(m2))),
      ↪ sca(dim, d10)), add(mul(mul(sca(dim, d8), lit(m0)), mul(sca(dim,
      ↪ d1), sca(dim, d9))), mul(mul(lit(m3), lit(m0)), mul(sca(dim,
      ↪ d15), lit(m5))))).eval()
17 mul(add(add(mul(sca(dim, d14), sca(dim, d12)), mul(sca(dim, d6),
      ↪ lit(m12))), mul(mul(lit(m15), lit(m5)), lit(m9))),
      ↪ mul(mul(sca(dim, d8), add(sca(dim, d12), sca(dim, d13))),
      ↪ sca(dim, d7))).eval()
```

```
18 add(mul(add(mul(lit(m8), sca(dim, d3)), add(sca(dim, d10), lit(m14))),
       ↪ mul(add(lit(m12), sca(dim, d5)), mul(lit(m9), lit(m7)))),
       ↪ add(mul(add(sca(dim, d11), lit(m16)), sca(dim, d13)),
       ↪ add(sca(dim, d8), sca(dim, d5)))).eval()
19 add(mul(mul(add(sca(dim, d11), sca(dim, d6)), add(lit(m3), sca(dim,
       ↪ d5))), add(lit(m7), sca(dim, d3))), lit(m7)).eval()
20 add(add(mul(mul(sca(dim, d4), lit(m2)), sca(dim, d11)), add(lit(m16),
       ↪ mul(sca(dim, d9), sca(dim, d7)))), sca(dim, d14)).eval()
21 mul(add(mul(mul(sca(dim, d1), sca(dim, d7)), sca(dim, d1)), sca(dim,
       ↪ d10)), mul(add(lit(m2), sca(dim, d6)), mul(add(lit(m16),
       ↪ lit(m9)), add(lit(m15), lit(m12))))).eval()
22 mul(mul(mul(mul(lit(m4), sca(dim, d9)), lit(m16)), mul(add(sca(dim, d0),
       ↪ lit(m4)), mul(sca(dim, d15), sca(dim, d1)))),
       ↪ mul(mul(add(sca(dim, d11), lit(m6)), mul(sca(dim, d1), sca(dim,
       ↪ d13))), lit(m15))).eval()
23 add(add(mul(mul(lit(m8), sca(dim, d9)), sca(dim, d2)), sca(dim, d4)),
       ↪ add(add(sca(dim, d15), lit(m2)), mul(sca(dim, d10), sca(dim,
       ↪ d0)))).eval()
24 mul(mul(add(mul(sca(dim, d12), sca(dim, d6)), mul(lit(m14), lit(m8))),
       ↪ lit(m11)), lit(m2)).eval()
25 mul(add(add(add(lit(m9), sca(dim, d16)), add(lit(m10), sca(dim, d7))),
       ↪ lit(m15)), sca(dim, d7)).eval()
26 mul(mul(add(add(lit(m3), sca(dim, d5)), mul(lit(m5), sca(dim, d14))),
       ↪ sca(dim, d7)), add(mul(mul(sca(dim, d9), lit(m2)), sca(dim,
       ↪ d11)), add(lit(m16), add(sca(dim, d3), sca(dim, d9))))).eval()
27 add(mul(mul(add(lit(m4), lit(m3)), add(sca(dim, d12), sca(dim, d16))),
       ↪ mul(sca(dim, d8), sca(dim, d3))), add(add(add(lit(m6), lit(m7)),
       ↪ mul(lit(m4), sca(dim, d5))), add(sca(dim, d16), mul(lit(m4),
       ↪ sca(dim, d13))))).eval()
28 add(mul(add(add(sca(dim, d5), sca(dim, d13)), sca(dim, d10)),
       ↪ add(add(lit(m11), sca(dim, d0)), sca(dim, d12))), add(lit(m11),
       ↪ sca(dim, d3))).eval()
29 add(mul(mul(mul(lit(m12), lit(m4)), add(lit(m11), lit(m12))),
       ↪ mul(add(sca(dim, d15), lit(m6)), add(lit(m5), sca(dim, d4)))),
       ↪ sca(dim, d16)).eval()
30 mul(mul(add(mul(lit(m4), lit(m6)), sca(dim, d6)), add(add(lit(m9),
       ↪ lit(m13)), sca(dim, d16))), add(mul(add(lit(m2), lit(m6)),
       ↪ sca(dim, d13)), mul(mul(sca(dim, d8), sca(dim, d7)), add(sca(dim,
       ↪ d7), lit(m4))))).eval()
```

A.1.2.5 Depth 5 (Random, Deep)

```
1 mul(add(mul(add(add(lit(m18), lit(m0)), lit(m21)), lit(m9)), sca(dim,
      ↪ d0)), add(lit(m11), sca(dim, d17))).eval()
2 mul(add(add(add(add(lit(m4), sca(dim, d14)), lit(m8)), add(add(lit(m5),
      ↪ lit(m19)), sca(dim, d2))), add(lit(m22), mul(add(lit(m0),
      ↪ sca(dim, d16)), sca(dim, d13)))), mul(lit(m10), sca(dim,
      ↪ d5))).eval()
3 mul(mul(mul(mul(add(lit(m24), sca(dim, d22)), add(lit(m16), sca(dim,
      ↪ d3))), lit(m17)), sca(dim, d24)), lit(m14)).eval()
4 add(mul(mul(add(add(sca(dim, d12), sca(dim, d14)), mul(lit(m0), sca(dim,
      ↪ d1))), mul(mul(sca(dim, d13), sca(dim, d22)), sca(dim, d22))),
      ↪ sca(dim, d19)), sca(dim, d24)).eval()
5 add(mul(add(mul(mul(lit(m17), sca(dim, d5)), sca(dim, d25)), sca(dim,
      ↪ d1)), add(sca(dim, d10), lit(m23))), mul(sca(dim, d1),
      ↪ add(sca(dim, d6), lit(m9)))).eval()
```

```
6  add(add(mul(add(add(lit(m21), sca(dim, d14)), lit(m17)),
       ↪ mul(mul(sca(dim, d20), lit(m4)), sca(dim, d21))), mul(lit(m4),
       ↪ sca(dim, d6))), mul(sca(dim, d19), lit(m16))).eval()
7  add(add(add(add(add(lit(m14), sca(dim, d1)), add(sca(dim, d14),
       ↪ lit(m23))), add(add(lit(m12), lit(m13)), mul(sca(dim, d6),
       ↪ lit(m25)))), sca(dim, d1)), add(add(sca(dim, d12),
       ↪ mul(add(sca(dim, d19), lit(m2)), mul(lit(m22), lit(m7)))),
       ↪ lit(m14))).eval()
8  add(mul(mul(mul(mul(lit(m1), lit(m22)), lit(m3)), lit(m11)), sca(dim,
       ↪ d22)), mul(sca(dim, d3), mul(add(sca(dim, d23), lit(m2)),
       ↪ sca(dim, d2)))).eval()
9  mul(add(mul(mul(mul(sca(dim, d5), sca(dim, d23)), mul(sca(dim, d15),
       ↪ lit(m5))), lit(m15)), add(sca(dim, d17), sca(dim, d14))),
       ↪ sca(dim, d20)).eval()
10 add(mul(add(mul(mul(sca(dim, d23), sca(dim, d25)), mul(lit(m11),
       ↪ sca(dim, d1))), sca(dim, d22)), mul(lit(m8), add(mul(lit(m5),
       ↪ lit(m7)), sca(dim, d15)))), sca(dim, d7)).eval()
11 mul(mul(mul(mul(mul(lit(m0), sca(dim, d19)), add(lit(m25), sca(dim,
       ↪ d7))), mul(lit(m17), sca(dim, d23))), lit(m12)), sca(dim,
       ↪ d2)).eval()
12 add(mul(mul(mul(mul(lit(m16), lit(m6)), add(sca(dim, d14), sca(dim,
       ↪ d23))), sca(dim, d23)), add(add(mul(sca(dim, d16), lit(m21)),
       ↪ lit(m18)), sca(dim, d12))), mul(add(sca(dim, d7), lit(m25)),
       ↪ sca(dim, d13))).eval()
13 add(add(mul(mul(mul(lit(m21), lit(m1)), add(sca(dim, d25), sca(dim,
       ↪ d19))), add(lit(m17), sca(dim, d11))), sca(dim, d8)), sca(dim,
       ↪ d25)).eval()
14 add(mul(add(add(mul(lit(m23), sca(dim, d22)), mul(lit(m25), lit(m3))),
       ↪ mul(lit(m9), sca(dim, d12))), sca(dim, d20)), lit(m11)).eval()
15 add(add(add(mul(mul(lit(m14), lit(m22)), sca(dim, d21)), mul(sca(dim,
       ↪ d13), add(sca(dim, d19), lit(m1)))), mul(add(sca(dim, d3),
       ↪ mul(sca(dim, d25), lit(m17))), mul(add(sca(dim, d8), sca(dim,
       ↪ d7)), add(sca(dim, d13), sca(dim, d0))))), mul(sca(dim, d22),
       ↪ add(mul(sca(dim, d25), lit(m12)), sca(dim, d9)))).eval()
16 mul(add(mul(mul(add(sca(dim, d23), lit(m5)), sca(dim, d9)),
       ↪ add(mul(sca(dim, d7), sca(dim, d16)), sca(dim, d2))), lit(m4)),
       ↪ add(mul(sca(dim, d1), sca(dim, d4)), add(mul(mul(sca(dim, d2),
       ↪ lit(m10)), add(lit(m15), sca(dim, d0))), sca(dim, d17)))).eval()
17 add(mul(add(add(mul(sca(dim, d24), sca(dim, d4)), sca(dim, d8)),
       ↪ add(mul(lit(m24), sca(dim, d2)), sca(dim, d4))), sca(dim, d14)),
       ↪ add(sca(dim, d25), mul(add(add(sca(dim, d17), lit(m10)),
       ↪ add(sca(dim, d5), lit(m5))), mul(mul(lit(m18), sca(dim, d25)),
       ↪ mul(sca(dim, d5), sca(dim, d15)))))).eval()
18 add(mul(mul(mul(mul(lit(m14), lit(m20)), mul(lit(m10), sca(dim, d24))),
       ↪ lit(m13)), mul(lit(m2), mul(mul(lit(m18), sca(dim, d9)),
       ↪ add(sca(dim, d0), lit(m4))))), lit(m15)).eval()
19 mul(mul(mul(mul(add(lit(m13), lit(m0)), add(lit(m2), lit(m6))), sca(dim,
       ↪ d19)), add(mul(lit(m11), lit(m0)), add(lit(m0), add(lit(m3),
       ↪ lit(m1))))), mul(lit(m11), mul(sca(dim, d23), mul(lit(m19),
       ↪ add(lit(m12), sca(dim, d6))))).eval()
20 mul(mul(mul(add(mul(lit(m3), lit(m0)), lit(m16)), lit(m4)), add(sca(dim,
       ↪ d21), lit(m6))), mul(mul(mul(mul(sca(dim, d25), lit(m14)),
       ↪ lit(m18)), sca(dim, d5)), mul(sca(dim, d0), add(sca(dim, d18),
       ↪ add(sca(dim, d19), sca(dim, d9))))).eval()
21 add(add(add(add(mul(lit(m11), sca(dim, d4)), mul(lit(m25), sca(dim,
       ↪ d24))), sca(dim, d12)), sca(dim, d6)), mul(lit(m1), sca(dim,
       ↪ d20))).eval()
```

```
22 mul(mul(add(add(add(lit(m23), lit(m4)), sca(dim, d0)), mul(sca(dim, d3),
       ↪ mul(sca(dim, d14), lit(m15)))), sca(dim, d1)), sca(dim,
       ↪ d23)).eval()
23 mul(add(mul(mul(add(sca(dim, d17), sca(dim, d24)), add(lit(m15),
       ↪ lit(m13))), mul(sca(dim, d20), lit(m21))), lit(m9)), sca(dim,
       ↪ d1)).eval()
24 add(mul(mul(mul(mul(sca(dim, d25), lit(m7)), add(sca(dim, d0),
       ↪ lit(m25))), add(add(lit(m21), lit(m21)), mul(sca(dim, d6),
       ↪ sca(dim, d20)))), mul(mul(mul(lit(m18), lit(m7)), lit(m7)),
       ↪ lit(m0))), mul(sca(dim, d1), mul(sca(dim, d4), lit(m20)))).eval()
25 mul(add(add(mul(add(lit(m7), lit(m7)), lit(m10)), sca(dim, d7)),
       ↪ sca(dim, d19)), sca(dim, d2)).eval()
26 add(mul(mul(add(mul(lit(m23), lit(m12)), add(sca(dim, d4), sca(dim,
       ↪ d1))), add(lit(m3), mul(lit(m5), sca(dim, d25)))),
       ↪ mul(add(add(sca(dim, d21), lit(m13)), mul(sca(dim, d5),
       ↪ lit(m17))), lit(m23))), sca(dim, d21)).eval()
27 mul(add(mul(add(add(sca(dim, d0), sca(dim, d16)), add(sca(dim, d3),
       ↪ lit(m17))), mul(mul(sca(dim, d10), sca(dim, d5)), lit(m1))),
       ↪ mul(sca(dim, d9), lit(m23))), sca(dim, d5)).eval()
28 mul(mul(mul(add(mul(lit(m23), lit(m3)), sca(dim, d14)), mul(add(sca(dim,
       ↪ d7), lit(m4)), lit(m24))), mul(add(mul(sca(dim, d4), lit(m20)),
       ↪ sca(dim, d8)), sca(dim, d23))), sca(dim, d21)).eval()
29 mul(mul(add(add(add(sca(dim, d8), sca(dim, d1)), sca(dim, d1)),
       ↪ lit(m1)), add(sca(dim, d7), sca(dim, d9))), lit(m11)).eval()
30 mul(mul(mul(mul(add(lit(m9), lit(m14)), add(sca(dim, d0), lit(m15))),
       ↪ add(sca(dim, d17), sca(dim, d3))), add(mul(sca(dim, d19),
       ↪ lit(m1)), sca(dim, d0))), sca(dim, d17)).eval()
```

A.2 Biased-Randomly Generated Expression

A.2.1 Shallow Embedding and Implicit Staging

A.2.1.1 Depth 1 (Biased, Shallow)

```
1  add(m1, m1)
2  mul(sca(dim, d1), m0)
3  mul(sca(dim, d0), m0)
4  add(m0, m1)
5  add(m1, m1)
6  add(m0, m1)
7  add(m0, m1)
8  add(m0, m0)
9  mul(sca(dim, d1), m0)
10 add(m0, m1)
11 add(m0, m1)
12 mul(sca(dim, d1), m1)
13 add(m0, m1)
14 add(m1, m1)
15 mul(sca(dim, d0), m1)
16 mul(sca(dim, d1), m0)
17 mul(sca(dim, d0), m0)
18 mul(sca(dim, d0), m1)
19 mul(sca(dim, d0), m1)
20 mul(sca(dim, d0), m0)
21 add(m0, m0)
22 mul(sca(dim, d1), m1)
```

```
23 mul(sca(dim, d0), m1)
24 add(m0, m1)
25 add(m0, m1)
26 mul(sca(dim, d1), m0)
27 add(m0, m0)
28 mul(sca(dim, d0), m0)
29 add(m1, m0)
30 mul(sca(dim, d0), m1)
```

A.2.1.2 Depth 2 (Biased, Shallow)

```
1  add(add(m0, m0), add(m0, m3))
2  add(add(m4, m3), add(m3, m0))
3  add(add(m2, m4), add(m4, m4))
4  add(add(m2, m2), m3)
5  mul(sca(dim, d4), add(m2, m1))
6  mul(sca(dim, d1), add(m2, m1))
7  mul(sca(dim, d2), mul(sca(dim, d4), m3))
8  add(add(m2, m1), add(m4, m3))
9  mul(sca(dim, d0), mul(sca(dim, d2), m3))
10 add(add(m1, m0), m2)
11 mul(sca(dim, d1), mul(sca(dim, d1), m3))
12 add(add(m3, m1), add(m4, m0))
13 add(add(m4, m4), m0)
14 mul(sca(dim, d4), add(m2, m1))
15 mul(sca(dim, d2), mul(sca(dim, d0), m2))
16 mul(sca(dim, d4), mul(sca(dim, d1), m0))
17 add(add(m2, m3), m3)
18 mul(sca(dim, d0), add(m3, m3))
19 mul(sca(dim, d2), add(m0, m0))
20 mul(sca(dim, d2), add(m2, m0))
21 mul(sca(dim, d3), add(m0, m2))
22 mul(sca(dim, d1), add(m3, m2))
23 add(add(m0, m4), add(m1, m4))
24 mul(sca(dim, d2), mul(sca(dim, d3), m4))
25 mul(sca(dim, d4), add(m1, m2))
26 mul(sca(dim, d1), mul(sca(dim, d4), m4))
27 add(add(m3, m2), m4)
28 add(add(m3, m1), m4)
29 mul(sca(dim, d1), add(m1, m3))
30 mul(sca(dim, d0), add(m4, m1))
```

A.2.1.3 Depth 3 (Biased, Shallow)

```
1  mul(sca(dim, d9), add(add(m9, m2), m8))
2  add(add(add(m4, m8), add(m2, m9)), m4)
3  mul(sca(dim, d4), add(add(m2, m2), add(m8, m7)))
4  add(add(add(m4, m2), add(m0, m0)), add(m7, m3))
5  add(add(add(m7, m6), add(m6, m1)), add(m2, add(m2, m4)))
6  add(add(add(m6, m5), m4), add(m0, m5))
7  add(add(add(m4, m1), m7), m4)
8  mul(sca(dim, d8), add(add(m3, m3), m9))
9  mul(sca(dim, d1), mul(sca(dim, d5), add(m2, m0)))
10 mul(sca(dim, d3), mul(sca(dim, d6), add(m0, m3)))
11 mul(sca(dim, d3), mul(sca(dim, d5), mul(sca(dim, d5), m6)))
12 mul(sca(dim, d2), add(add(m1, m0), m2))
```

```
13 mul(sca(dim, d0), add(add(m9, m1), add(m5, m9)))
14 add(add(add(m1, m0), add(m3, m3)), add(m1, m8))
15 add(add(add(m4, m0), add(m7, m7)), add(m5, m3))
16 mul(sca(dim, d5), mul(sca(dim, d3), add(m8, m7)))
17 mul(sca(dim, d5), add(add(m6, m1), add(m7, m1)))
18 mul(sca(dim, d3), mul(sca(dim, d7), mul(sca(dim, d8), m8)))
19 add(add(add(m1, m2), m9), m8)
20 mul(sca(dim, d2), add(add(m5, m1), add(m2, m8)))
21 add(add(add(m2, m3), m4), add(m3, add(m7, m8)))
22 mul(sca(dim, d8), add(add(m5, m5), m6))
23 mul(sca(dim, d8), add(add(m0, m7), m6))
24 mul(sca(dim, d3), add(add(m0, m9), add(m5, m6)))
25 add(add(add(m6, m0), m1), m2)
26 mul(sca(dim, d0), add(add(m3, m3), add(m2, m6)))
27 add(add(add(m2, m3), m6), add(m2, add(m6, m8)))
28 mul(sca(dim, d9), add(add(m3, m0), m7))
29 mul(sca(dim, d9), mul(sca(dim, d3), add(m0, m6)))
30 add(add(add(m5, m0), add(m5, m8)), m8)
```

A.2.1.4 Depth 4 (Biased, Shallow)

```
1  mul(add(add(mul(m12, m8), m4), m7), add(add(m15, add(m2, sca(dim, d3))),
       ↪ m5))
2  mul(sca(dim, d16), mul(sca(dim, d0), add(add(m8, m11), m5)))
3  add(add(add(add(m6, m1), m3), m9), m3)
4  add(add(add(add(m2, m11), m15), add(m12, m1)), add(add(m14, m7), m5))
5  mul(sca(dim, d2), mul(sca(dim, d0), add(add(m0, m5), m12)))
6  mul(sca(dim, d11), mul(sca(dim, d11), add(add(m10, m4), add(m13, m9))))
7  add(add(add(add(m14, m5), m7), m3), m0)
8  add(add(add(add(m5, m5), m12), m2), add(add(m10, m11), add(m10, add(m11,
       ↪ m6))))
9  add(add(add(add(m0, m8), add(m5, m1)), m3), add(m14, m0))
10 add(add(add(add(m2, m14), m6), m12), m7)
11 add(add(add(add(m4, m5), add(m0, m10)), m0), m0)
12 add(add(add(add(m9, m8), m9), add(m15, m9)), add(add(add(m3, m9),
       ↪ add(m2, m15)), m1))
13 mul(sca(dim, d1), add(add(add(m8, m4), add(m10, m5)), add(m15, m9)))
14 add(add(add(add(m2, m16), add(m8, m0)), add(add(m8, m2), m3)), add(m0,
       ↪ add(m1, m13)))
15 add(add(add(add(m9, m12), m11), add(m10, m14)), add(m0, m11))
16 add(add(add(add(m13, m1), add(m2, m6)), add(add(m15, m12), add(m10,
       ↪ m6))), add(add(m11, m10), m1))
17 mul(sca(dim, d1), add(add(add(m5, m11), m15), add(m4, add(m7, m7))))
18 add(add(add(add(m10, m3), m12), add(m16, add(m11, m4))), m9)
19 mul(sca(dim, d6), add(add(add(m9, m11), add(m4, m12)), add(add(m11, m5),
       ↪ m5)))
20 add(add(add(add(m11, m11), add(m7, m6)), add(add(m14, m2), add(m2,
       ↪ m7))), add(add(m3, add(m11, m4)), m6))
21 add(add(add(add(m2, m1), m8), m11), add(add(m6, m3), add(m1, m12)))
22 mul(sca(dim, d12), add(add(add(m4, m9), add(m10, m7)), add(add(m1, m5),
       ↪ m5)))
23 add(add(add(add(m7, m6), m5), add(m5, m4)), m7)
24 mul(sca(dim, d5), add(add(add(m12, m11), add(m10, m5)), m2))
25 add(add(add(add(m8, m1), add(m10, m12)), m1), add(add(m11, m10), m13))
26 add(add(add(add(m12, m13), m4), add(m1, add(m6, m2))), m11)
27 mul(sca(dim, d5), add(add(add(m3, m10), add(m2, m2)), m10))
28 mul(sca(dim, d9), mul(sca(dim, d8), add(add(m12, m1), add(m8, m11))))
```

```
29 add(add(add(add(m15, m5), add(m4, m2)), add(m12, add(m6, m12))),
       ↪ add(add(add(m7, m3), add(m12, m8)), add(m12, m13)))
30 add(add(add(add(m15, m8), add(m7, m10)), add(add(m8, m12), add(m13,
       ↪ m9))), m7)
31 mul(sca(dim, d12), mul(sca(dim, d11), mul(sca(dim, d7), add(m14, m10))))
```

A.2.1.5 Depth 5 (Biased, Shallow)

```
1  add(add(add(add(add(m4, m8), m13), m14), add(m0, add(add(m15, m20),
       ↪ add(m3, m23)))), m3)
2  add(add(add(add(add(m3, m10), add(m2, m22)), m12), m15), m11)
3  add(add(add(add(add(m14, m14), m23), m4), add(m21, m4)), m2)
4  mul(sca(dim, d25), mul(sca(dim, d22), mul(sca(dim, d12), mul(sca(dim,
       ↪ d24), mul(sca(dim, d3), m11)))))
5  mul(sca(dim, d0), add(add(add(add(m13, m9), add(m2, m12)), add(m12,
       ↪ m2)), m1))
6  add(add(add(add(add(m21, m5), m13), add(add(m18, m6), m22)), add(m23,
       ↪ add(m24, m12))), add(m12, add(m6, add(add(m19, m6), add(m1,
       ↪ m17)))))
7  add(add(add(add(add(m25, m6), m16), m22), add(m22, m9)), add(m0, m10))
8  mul(sca(dim, d18), add(add(add(add(m5, m23), m12), add(m18, add(m6,
       ↪ m4))), m13))
9  add(add(add(add(add(m15, m10), m1), m0), add(add(add(m12, m16), add(m5,
       ↪ m6)), m11)), m22)
10 mul(sca(dim, d16), add(add(add(add(m6, m17), add(m17, m23)), add(m12,
       ↪ m17)), add(add(m20, m16), add(add(m7, m6), add(m5, m18)))))
11 mul(sca(dim, d11), mul(sca(dim, d7), add(add(add(m8, m16), m4), m18)))
12 mul(sca(dim, d10), mul(sca(dim, d19), mul(sca(dim, d18), add(add(m3,
       ↪ m5), m11))))
13 mul(sca(dim, d5), add(add(add(add(m15, m16), add(m12, m13)), add(add(m3,
       ↪ m3), add(m16, m18))), m13))
14 mul(sca(dim, d16), mul(sca(dim, d19), mul(sca(dim, d3), mul(sca(dim,
       ↪ d6), add(m12, m5)))))
15 mul(sca(dim, d23), add(add(add(add(m11, m3), add(m4, m14)), m8),
       ↪ add(add(m5, m19), m2)))
16 mul(sca(dim, d10), mul(sca(dim, d21), add(add(add(m11, m22), add(m2,
       ↪ m23)), m11)))
17 mul(sca(dim, d22), add(add(add(add(m7, m19), add(m17, m24)), m14), m15))
18 mul(sca(dim, d7), mul(sca(dim, d23), add(add(add(m6, m24), m24),
       ↪ add(m13, add(m17, m25)))))
19 mul(sca(dim, d19), add(add(add(add(m8, m0), add(m25, m1)), add(m10,
       ↪ m23)), add(m1, add(m6, m9))))
20 add(add(add(add(add(m21, m14), m17), add(add(m20, m4), m21)), add(m4,
       ↪ m6)), add(m19, m16))
21 add(add(add(add(add(m14, m1), add(m14, m23)), add(add(m12, m13), add(m6,
       ↪ m25))), m1), add(add(m12, add(add(m19, m2), add(m22, m7))), m14))
22 add(add(add(add(add(m1, m22), m3), m11), m22), add(m3, add(add(m23, m2),
       ↪ m2)))
23 mul(sca(dim, d5), mul(sca(dim, d22), mul(sca(dim, d17), add(add(m8,
       ↪ m25), m11))))
24 add(add(add(add(add(m17, m12), add(m20, m6)), add(m25, add(m11, m1))),
       ↪ m22), add(m8, add(add(add(m3, m14), add(m13, m8)), add(add(m3,
       ↪ m20), add(m5, m19)))))
25 add(add(add(add(add(m12, m2), m20), add(add(m25, m5), m18)), m20),
       ↪ add(add(m23, add(add(m10, m14), m10)), m2))
26 mul(sca(dim, d25), mul(sca(dim, d0), add(add(add(m3, m3), add(m24,
       ↪ m21)), m1)))
```

```
27 mul(sca(dim, d5), mul(sca(dim, d13), mul(sca(dim, d1), mul(sca(dim,
       ↪ d19), add(m3, m6)))))
28 add(add(add(add(add(m4, m16), add(m14, m15)), add(m7, add(m20, m11))),
       ↪ m1), add(m14, add(m25, m6)))
29 mul(sca(dim, d15), add(add(add(add(m10, m23), add(m3, m8)), add(add(m14,
       ↪ m20), add(m7, m18))), add(m0, add(m22, add(m12, m2)))))
30 add(add(add(add(add(m25, m5), add(m5, m9)), add(add(m7, m16), m2)), m4),
       ↪ add(add(m1, m4), add(add(add(m2, m10), add(m15, m0)), m17)))
```

A.2.2 Deep Embedding

A.2.2.1 Depth 1 (Biased, Deep)

```
1  add(lit(m1), lit(m1)).eval()
2  mul(sca(dim, d1), lit(m0)).eval()
3  mul(sca(dim, d0), lit(m0)).eval()
4  add(lit(m0), lit(m1)).eval()
5  add(lit(m1), lit(m1)).eval()
6  add(lit(m0), lit(m1)).eval()
7  add(lit(m0), lit(m1)).eval()
8  add(lit(m0), lit(m0)).eval()
9  mul(sca(dim, d1), lit(m0)).eval()
10 add(lit(m0), lit(m1)).eval()
11 add(lit(m0), lit(m1)).eval()
12 mul(sca(dim, d1), lit(m1)).eval()
13 add(lit(m0), lit(m1)).eval()
14 add(lit(m1), lit(m1)).eval()
15 mul(sca(dim, d0), lit(m1)).eval()
16 mul(sca(dim, d1), lit(m0)).eval()
17 mul(sca(dim, d0), lit(m0)).eval()
18 mul(sca(dim, d0), lit(m1)).eval()
19 mul(sca(dim, d0), lit(m1)).eval()
20 mul(sca(dim, d0), lit(m0)).eval()
21 add(lit(m0), lit(m0)).eval()
22 mul(sca(dim, d1), lit(m1)).eval()
23 mul(sca(dim, d0), lit(m1)).eval()
24 add(lit(m0), lit(m1)).eval()
25 add(lit(m0), lit(m1)).eval()
26 mul(sca(dim, d1), lit(m0)).eval()
27 add(lit(m0), lit(m0)).eval()
28 mul(sca(dim, d0), lit(m0)).eval()
29 add(lit(m1), lit(m0)).eval()
30 mul(sca(dim, d0), lit(m1)).eval()
```

A.2.2.2 Depth 2 (Biased, Deep)

```
1  add(add(lit(m0), lit(m0)), add(lit(m0), lit(m3))).eval()
2  add(add(lit(m4), lit(m3)), add(lit(m3), lit(m0))).eval()
3  add(add(lit(m2), lit(m4)), add(lit(m4), lit(m4))).eval()
4  add(add(lit(m2), lit(m2)), lit(m3)).eval()
5  mul(sca(dim, d4), add(lit(m2), lit(m1))).eval()
6  mul(sca(dim, d1), add(lit(m2), lit(m1))).eval()
7  mul(sca(dim, d2), mul(sca(dim, d4), lit(m3))).eval()
8  add(add(lit(m2), lit(m1)), add(lit(m4), lit(m3))).eval()
9  mul(sca(dim, d0), mul(sca(dim, d2), lit(m3))).eval()
10 add(add(lit(m1), lit(m0)), lit(m2)).eval()
```

```
11 mul(sca(dim, d1), mul(sca(dim, d1), lit(m3))).eval()
12 add(add(lit(m3), lit(m1)), add(lit(m4), lit(m0))).eval()
13 add(add(lit(m4), lit(m4)), lit(m0)).eval()
14 mul(sca(dim, d4), add(lit(m2), lit(m1))).eval()
15 mul(sca(dim, d2), mul(sca(dim, d0), lit(m2))).eval()
16 mul(sca(dim, d4), mul(sca(dim, d1), lit(m0))).eval()
17 add(add(lit(m2), lit(m3)), lit(m3)).eval()
18 mul(sca(dim, d0), add(lit(m3), lit(m3))).eval()
19 mul(sca(dim, d2), add(lit(m0), lit(m0))).eval()
20 mul(sca(dim, d2), add(lit(m2), lit(m0))).eval()
21 mul(sca(dim, d3), add(lit(m0), lit(m2))).eval()
22 mul(sca(dim, d1), add(lit(m3), lit(m2))).eval()
23 add(add(lit(m0), lit(m4)), add(lit(m1), lit(m4))).eval()
24 mul(sca(dim, d2), mul(sca(dim, d3), lit(m4))).eval()
25 mul(sca(dim, d4), add(lit(m1), lit(m2))).eval()
26 mul(sca(dim, d1), mul(sca(dim, d4), lit(m4))).eval()
27 add(add(lit(m3), lit(m2)), lit(m4)).eval()
28 add(add(lit(m3), lit(m1)), lit(m4)).eval()
29 mul(sca(dim, d1), add(lit(m1), lit(m3))).eval()
30 mul(sca(dim, d0), add(lit(m4), lit(m1))).eval()
```

A.2.2.3 Depth 3 (Biased, Deep)

```
1  mul(sca(dim, d9), add(add(lit(m9), lit(m2)), lit(m8))).eval()
2  add(add(add(lit(m4), lit(m8)), add(lit(m2), lit(m9))), lit(m4)).eval()
3  mul(sca(dim, d4), add(add(lit(m2), lit(m2)), add(lit(m8),
       ↪ lit(m7)))).eval()
4  add(add(add(lit(m4), lit(m2)), add(lit(m0), lit(m0))), add(lit(m7),
       ↪ lit(m3))).eval()
5  add(add(add(lit(m7), lit(m6)), add(lit(m6), lit(m1))), add(lit(m2),
       ↪ add(lit(m2), lit(m4)))).eval()
6  add(add(add(lit(m6), lit(m5)), lit(m4)), add(lit(m0), lit(m5))).eval()
7  add(add(add(lit(m4), lit(m1)), lit(m7)), lit(m4)).eval()
8  mul(sca(dim, d8), add(add(lit(m3), lit(m3)), lit(m9))).eval()
9  mul(sca(dim, d1), mul(sca(dim, d5), add(lit(m2), lit(m0)))).eval()
10 mul(sca(dim, d3), mul(sca(dim, d6), add(lit(m0), lit(m3)))).eval()
11 mul(sca(dim, d3), mul(sca(dim, d5), mul(sca(dim, d5), lit(m6)))).eval()
12 mul(sca(dim, d2), add(add(lit(m1), lit(m0)), lit(m2))).eval()
13 mul(sca(dim, d0), add(add(lit(m9), lit(m1)), add(lit(m5),
       ↪ lit(m9)))).eval()
14 add(add(add(lit(m1), lit(m0)), add(lit(m3), lit(m3))), add(lit(m1),
       ↪ lit(m8))).eval()
15 add(add(add(lit(m4), lit(m0)), add(lit(m7), lit(m7))), add(lit(m5),
       ↪ lit(m3))).eval()
16 mul(sca(dim, d5), mul(sca(dim, d3), add(lit(m8), lit(m7)))).eval()
17 mul(sca(dim, d5), add(add(lit(m6), lit(m1)), add(lit(m7),
       ↪ lit(m1)))).eval()
18 mul(sca(dim, d3), mul(sca(dim, d7), mul(sca(dim, d8), lit(m8)))).eval()
19 add(add(add(lit(m1), lit(m2)), lit(m9)), lit(m8)).eval()
20 mul(sca(dim, d2), add(add(lit(m5), lit(m1)), add(lit(m2),
       ↪ lit(m8)))).eval()
21 add(add(add(lit(m2), lit(m3)), lit(m4)), add(lit(m3), add(lit(m7),
       ↪ lit(m8)))).eval()
22 mul(sca(dim, d8), add(add(lit(m5), lit(m5)), lit(m6))).eval()
23 mul(sca(dim, d8), add(add(lit(m0), lit(m7)), lit(m6))).eval()
24 mul(sca(dim, d3), add(add(lit(m0), lit(m9)), add(lit(m5),
       ↪ lit(m6)))).eval()
```

```
25 add(add(add(lit(m6), lit(m0)), lit(m1)), lit(m2)).eval()
26 mul(sca(dim, d0), add(add(lit(m3), lit(m3)), add(lit(m2),
       ↪ lit(m6)))).eval()
27 add(add(add(lit(m2), lit(m3)), lit(m6)), add(lit(m2), add(lit(m6),
       ↪ lit(m8)))).eval()
28 mul(sca(dim, d9), add(add(lit(m3), lit(m0)), lit(m7))).eval()
29 mul(sca(dim, d9), mul(sca(dim, d3), add(lit(m0), lit(m6)))).eval()
30 add(add(add(lit(m5), lit(m0)), add(lit(m5), lit(m8))), lit(m8)).eval()
```

A.2.2.4 Depth 4 (Biased, Deep)

```
1  mul(sca(dim, d16), mul(sca(dim, d0), add(add(lit(m8), lit(m11)),
       ↪ lit(m5)))).eval()
2  add(add(add(add(lit(m6), lit(m1)), lit(m3)), lit(m9)), lit(m3)).eval()
3  add(add(add(add(lit(m2), lit(m11)), lit(m15)), add(lit(m12), lit(m1))),
       ↪ add(add(lit(m14), lit(m7)), lit(m5))).eval()
4  mul(sca(dim, d2), mul(sca(dim, d0), add(add(lit(m0), lit(m5)),
       ↪ lit(m12)))).eval()
5  mul(sca(dim, d11), mul(sca(dim, d11), add(add(lit(m10), lit(m4)),
       ↪ add(lit(m13), lit(m9))))).eval()
6  add(add(add(add(lit(m14), lit(m5)), lit(m7)), lit(m3)), lit(m0)).eval()
7  add(add(add(add(lit(m5), lit(m5)), lit(m12)), lit(m2)),
       ↪ add(add(lit(m10), lit(m11)), add(lit(m10), add(lit(m11),
       ↪ lit(m6))))).eval()
8  add(add(add(add(lit(m0), lit(m8)), add(lit(m5), lit(m1))), lit(m3)),
       ↪ add(lit(m14), lit(m0))).eval()
9  add(add(add(add(lit(m2), lit(m14)), lit(m6)), lit(m12)), lit(m7)).eval()
10 add(add(add(add(lit(m4), lit(m5)), add(lit(m0), lit(m10))), lit(m0)),
       ↪ lit(m0)).eval()
11 add(add(add(add(lit(m9), lit(m8)), lit(m9)), add(lit(m15), lit(m9))),
       ↪ add(add(add(lit(m3), lit(m9)), add(lit(m2), lit(m15))),
       ↪ lit(m1))).eval()
12 mul(sca(dim, d1), add(add(add(lit(m8), lit(m4)), add(lit(m10),
       ↪ lit(m5))), add(lit(m15), lit(m9)))).eval()
13 add(add(add(add(lit(m2), lit(m16)), add(lit(m8), lit(m0))),
       ↪ add(add(lit(m8), lit(m2)), lit(m3))), add(lit(m0), add(lit(m1),
       ↪ lit(m13)))).eval()
14 add(add(add(add(lit(m9), lit(m12)), lit(m11)), add(lit(m10), lit(m14))),
       ↪ add(lit(m0), lit(m11))).eval()
15 add(add(add(add(lit(m13), lit(m1)), add(lit(m2), lit(m6))),
       ↪ add(add(lit(m15), lit(m12)), add(lit(m10), lit(m6)))),
       ↪ add(add(lit(m11), lit(m10)), lit(m1))).eval()
16 mul(sca(dim, d1), add(add(add(lit(m5), lit(m11)), lit(m15)),
       ↪ add(lit(m4), add(lit(m7), lit(m7))))).eval()
17 add(add(add(add(lit(m10), lit(m3)), lit(m12)), add(lit(m16),
       ↪ add(lit(m11), lit(m4)))), lit(m9)).eval()
18 mul(sca(dim, d6), add(add(add(lit(m9), lit(m11)), add(lit(m4),
       ↪ lit(m12))), add(add(lit(m11), lit(m5)), lit(m5)))).eval()
19 add(add(add(add(lit(m11), lit(m11)), add(lit(m7), lit(m6))),
       ↪ add(add(lit(m14), lit(m2)), add(lit(m2), lit(m7)))),
       ↪ add(add(lit(m3), add(lit(m11), lit(m4))), lit(m6))).eval()
20 add(add(add(add(lit(m2), lit(m1)), lit(m8)), lit(m11)), add(add(lit(m6),
       ↪ lit(m3)), add(lit(m1), lit(m12)))).eval()
21 mul(sca(dim, d12), add(add(add(lit(m4), lit(m9)), add(lit(m10),
       ↪ lit(m7))), add(add(lit(m1), lit(m5)), lit(m5)))).eval()
22 add(add(add(add(lit(m7), lit(m6)), lit(m5)), add(lit(m5), lit(m4))),
       ↪ lit(m7)).eval()
```

```
23 mul(sca(dim, d5), add(add(add(lit(m12), lit(m11)), add(lit(m10),
       ↪ lit(m5))), lit(m2))).eval()
24 add(add(add(add(lit(m8), lit(m1)), add(lit(m10), lit(m12))), lit(m1)),
       ↪ add(add(lit(m11), lit(m10)), lit(m13))).eval()
25 add(add(add(add(lit(m12), lit(m13)), lit(m4)), add(lit(m1), add(lit(m6),
       ↪ lit(m2)))), lit(m11)).eval()
26 mul(sca(dim, d5), add(add(add(lit(m3), lit(m10)), add(lit(m2),
       ↪ lit(m2))), lit(m10))).eval()
27 mul(sca(dim, d9), mul(sca(dim, d8), add(add(lit(m12), lit(m1)),
       ↪ add(lit(m8), lit(m11))))).eval()
28 add(add(add(add(lit(m15), lit(m5)), add(lit(m4), lit(m2))),
       ↪ add(lit(m12), add(lit(m6), lit(m12)))), add(add(add(lit(m7),
       ↪ lit(m3)), add(lit(m12), lit(m8))), add(lit(m12),
       ↪ lit(m13)))).eval()
29 add(add(add(add(lit(m15), lit(m8)), add(lit(m7), lit(m10))),
       ↪ add(add(lit(m8), lit(m12)), add(lit(m13), lit(m9)))),
       ↪ lit(m7)).eval()
30 mul(sca(dim, d12), mul(sca(dim, d11), mul(sca(dim, d7), add(lit(m14),
       ↪ lit(m10))))).eval()
```

A.2.2.5 Depth 5 (Biased, Deep)

```
1 add(add(add(add(add(lit(m4), lit(m8)), lit(m13)), lit(m14)),
      ↪ add(lit(m0), add(add(lit(m15), lit(m20)), add(lit(m3),
      ↪ lit(m23))))), lit(m3)).eval()
2 add(add(add(add(add(lit(m3), lit(m10)), add(lit(m2), lit(m22))),
      ↪ lit(m12)), lit(m15)), lit(m11)).eval()
3 add(add(add(add(add(lit(m14), lit(m14)), lit(m23)), lit(m4)),
      ↪ add(lit(m21), lit(m4))), lit(m2)).eval()
4 mul(sca(dim, d25), mul(sca(dim, d22), mul(sca(dim, d12), mul(sca(dim,
      ↪ d24), mul(sca(dim, d3), lit(m11)))))).eval()
5 mul(sca(dim, d0), add(add(add(add(lit(m13), lit(m9)), add(lit(m2),
      ↪ lit(m12))), add(lit(m12), lit(m2))), lit(m1))).eval()
6 add(add(add(add(add(lit(m21), lit(m5)), lit(m13)), add(add(lit(m18),
      ↪ lit(m6)), lit(m22))), add(lit(m23), add(lit(m24), lit(m12)))),
      ↪ add(lit(m12), add(lit(m6), add(add(lit(m19), lit(m6)),
      ↪ add(lit(m1), lit(m17))))).eval()
7 add(add(add(add(add(lit(m25), lit(m6)), lit(m16)), lit(m22)),
      ↪ add(lit(m22), lit(m9))), add(lit(m0), lit(m10))).eval()
8 mul(sca(dim, d18), add(add(add(add(lit(m5), lit(m23)), lit(m12)),
      ↪ add(lit(m18), add(lit(m6), lit(m4)))), lit(m13))).eval()
9 add(add(add(add(add(lit(m15), lit(m10)), lit(m1)), lit(m0)),
      ↪ add(add(add(lit(m12), lit(m16)), add(lit(m5), lit(m6))),
      ↪ lit(m11))), lit(m22)).eval()
10 mul(sca(dim, d16), add(add(add(add(lit(m6), lit(m17)), add(lit(m17),
      ↪ lit(m23))), add(lit(m12), lit(m17))), add(add(lit(m20),
      ↪ lit(m16)), add(add(lit(m7), lit(m6)), add(lit(m5),
      ↪ lit(m18))))).eval()
11 mul(sca(dim, d11), mul(sca(dim, d7), add(add(add(lit(m8), lit(m16)),
      ↪ lit(m4)), lit(m18)))).eval()
12 mul(sca(dim, d10), mul(sca(dim, d19), mul(sca(dim, d18),
      ↪ add(add(lit(m3), lit(m5)), lit(m11))))).eval()
13 mul(sca(dim, d5), add(add(add(add(lit(m15), lit(m16)), add(lit(m12),
      ↪ lit(m13))), add(add(lit(m3), lit(m3)), add(lit(m16), lit(m18)))),
      ↪ lit(m13))).eval()
14 mul(sca(dim, d16), mul(sca(dim, d19), mul(sca(dim, d3), mul(sca(dim,
      ↪ d6), add(lit(m12), lit(m5))))).eval()
```

```
15 mul(sca(dim, d23), add(add(add(add(lit(m11), lit(m3)), add(lit(m4),
       ↪ lit(m14))), lit(m8)), add(add(lit(m5), lit(m19)),
       ↪ lit(m2))))).eval()
16 mul(sca(dim, d10), mul(sca(dim, d21), add(add(add(lit(m11), lit(m22)),
       ↪ add(lit(m2), lit(m23))), lit(m11)))).eval()
17 mul(sca(dim, d22), add(add(add(add(lit(m7), lit(m19)), add(lit(m17),
       ↪ lit(m24))), lit(m14)), lit(m15))).eval()
18 mul(sca(dim, d7), mul(sca(dim, d23), add(add(add(lit(m6), lit(m24)),
       ↪ lit(m24)), add(lit(m13), add(lit(m17), lit(m25))))))).eval()
19 mul(sca(dim, d19), add(add(add(add(lit(m8), lit(m0)), add(lit(m25),
       ↪ lit(m1))), add(lit(m10), lit(m23))), add(lit(m1), add(lit(m6),
       ↪ lit(m9))))).eval()
20 add(add(add(add(add(lit(m21), lit(m14)), lit(m17)), add(add(lit(m20),
       ↪ lit(m4)), lit(m21))), add(lit(m4), lit(m6))), add(lit(m19),
       ↪ lit(m16))).eval()
21 add(add(add(add(add(lit(m14), lit(m1)), add(lit(m14), lit(m23))),
       ↪ add(add(lit(m12), lit(m13)), add(lit(m6), lit(m25)))), lit(m1)),
       ↪ add(add(lit(m12), add(add(lit(m19), lit(m2)), add(lit(m22),
       ↪ lit(m7)))), lit(m14))).eval()
22 add(add(add(add(add(lit(m1), lit(m22)), lit(m3)), lit(m11)), lit(m22)),
       ↪ add(lit(m3), add(add(lit(m23), lit(m2)), lit(m2)))).eval()
23 mul(sca(dim, d5), mul(sca(dim, d22), mul(sca(dim, d17), add(add(lit(m8),
       ↪ lit(m25)), lit(m11))))).eval()
24 add(add(add(add(add(lit(m17), lit(m12)), add(lit(m20), lit(m6))),
       ↪ add(lit(m25), add(lit(m11), lit(m1)))), lit(m22)), add(lit(m8),
       ↪ add(add(add(lit(m3), lit(m14)), add(lit(m13), lit(m8))),
       ↪ add(add(lit(m3), lit(m20)), add(lit(m5), lit(m19))))))).eval()
25 add(add(add(add(add(lit(m12), lit(m2)), lit(m20)), add(add(lit(m25),
       ↪ lit(m5)), lit(m18))), lit(m20)), add(add(lit(m23),
       ↪ add(add(lit(m10), lit(m14)), lit(m10))), lit(m2))).eval()
26 mul(sca(dim, d25), mul(sca(dim, d0), add(add(add(lit(m3), lit(m3)),
       ↪ add(lit(m24), lit(m21))), lit(m1)))).eval()
27 mul(sca(dim, d5), mul(sca(dim, d13), mul(sca(dim, d1), mul(sca(dim,
       ↪ d19), add(lit(m3), lit(m6))))))).eval()
28 add(add(add(add(add(lit(m4), lit(m16)), add(lit(m14), lit(m15))),
       ↪ add(lit(m7), add(lit(m20), lit(m11)))), lit(m1)), add(lit(m14),
       ↪ add(lit(m25), lit(m6)))).eval()
29 mul(sca(dim, d15), add(add(add(add(lit(m10), lit(m23)), add(lit(m3),
       ↪ lit(m8))), add(add(lit(m14), lit(m20)), add(lit(m7), lit(m18)))),
       ↪ add(lit(m0), add(lit(m22), add(lit(m12), lit(m2))))))).eval()
30 add(add(add(add(add(lit(m25), lit(m5)), add(lit(m5), lit(m9))),
       ↪ add(add(lit(m7), lit(m16)), lit(m2))), lit(m4)), add(add(lit(m1),
       ↪ lit(m4)), add(add(add(lit(m2), lit(m10)), add(lit(m15),
       ↪ lit(m0))), lit(m17)))).eval()
```

APPENDIX B

Tame-Staging Experiments

B.1 Centroid-Calculation Experiment

Listing B.1 shows the generation of weighted-vector arrays and the benchmark loops. `TickTock.tick()`, `TickTock.tock()`, and `TickTock.tockPrint()` are utility methods for measuring time differences in milliseconds.

LISTING B.1: Centroid-calculation benchmark

```
Random r = new Random(42);
WeightedVec[][] weightedVecs = new WeightedVec[20][];
for (int i = 0; i < 20; i++) {
  weightedVecs[i] = new WeightedVec[Math.abs(r.nextInt(50)) + 50];
  for (int j = 0; j < weightedVecs[i].length; j++) {
    Vec v = Vec.create(r.doubles(100000).toArray());
    int t = r.nextInt(20);
    double w = t > 15 ? 0.0 : t > 10 ? 1.0 : (double) t / 20.0;
    weightedVecs[i][j] = new WeightedVec(v, w);
  }
}

Vec res = null;
// Warm-up
for (int i = 0; i < 1000; i++) {
  res = centroid(weightedVecs[i % 20]);
}
System.gc();
try { Thread.sleep(2000); } catch (··· e) { e.printStackTrace(); }
// Measurement
TickTock.tick();
for (int i = 0; i < 1000; i++) {
  res = centroid(weightedVecs[i % 20]);
}
// Print time difference in ms
TickTock.tockPrint();
···
```

B.2 Radar Experiment

Listing B.2 shows the generation of tracked objects and the benchmark loops.

LISTING B.2: Radar benchmark

```
ArrayList<NamedRegion> regions = new ArrayList<>();
regions.add(r1);
…
regions.add(r14);
Random r = new Random(24);
int width = 1000;
int height = 1000;
List<TrackedObject> objs = new ArrayList<>();
for (int i = 0; i < 100000; i++) {
  objs.add(new TrackedObject(
    r.nextInt(width) - width / 2, r.nextInt(height) - height / 2)
  );
}

Radar radar = new Radar(regions, objs);
radar.track(1000, width, height);
System.gc();
try { Thread.sleep(2000); } catch (… e) { e.printStackTrace(); }

TickTock.tick();
List<TrackingData> res = radar.track(1000, width, height);
TickTock.tockPrint();
…
```

B.3 Connections Experiment

Listings B.3 and B.4 show the generation of list entries and the benchmark loops.

LISTING B.3: Connections benchmark

```
Database db = Database.createRandom(2000000);
ImmutableList<String> l = null;
for (int i = 0; i < 100; i++) {
  l = db.query("Tokyo", "Frankfurt", 2500.0, Vessel.Type.PLANE);
}
System.gc();
try { Thread.sleep(2000); } catch (… e) { e.printStackTrace(); }

TickTock.tick();
for (int i = 0; i < 100; i++) {
  l = db.query("Tokyo", "Frankfurt", 2500.0, Vessel.Type.PLANE);
}
TickTock.tockPrint();
…
```

LISTING B.4: Random entries factory

```
public static Database createRandom(int size) {
  Random r = new Random(42);

  ImmutableList.Builder<Vessel> vesselsBuilder = ImmutableList.builder();
  for (int i = 0; i < 100; i++) {
    Vessel v = new Vessel(
      r.nextInt(200),
      Vessel.Type.values()[r.nextInt(Vessel.Type.values().length)]
    );
    vesselsBuilder.add(v);
  }
  ImmutableList<Vessel> vessels = vesselsBuilder.build();

  String[] locations = { "Tokyo", "Frankfurt", "London", "New York" };

  ImmutableList.Builder<Connection> connectionsBuilder =
    ImmutableList.builder();
  for (int i = 0; i < size; i++) {
    Vessel v = vessels.get(r.nextInt(vessels.size()));
    Connection c = new Connection(
      locations[r.nextInt(locations.length)],
      locations[r.nextInt(locations.length)],
      r.nextInt(v.getCapacity() + 20),
      v,
      r.nextInt(2000)
    );
    connectionsBuilder.add(c);
  }

  return new Database(connectionsBuilder.build());
}
```

B.3.1 Compiler Implementation

```
class ImmListLCompiler implements Expression.Visitor {
  private abstract static class Op { ObjectClosure<?> closure; }
  private static class TransformOp extends Op {
    private TransformOp(
              ObjectClosure<Function<Object, ?>> functionClosure) {
      this.closure = functionClosure;
    }
  }
  private static class FilterOp extends Op {
    private FilterOp(
              ObjectClosure<Predicate<Object>> predicateClosure) {
      this.closure = predicateClosure;
    }
  }

  private final Environment.Binder binder;
  private ObjectClosure<?> closure;

  private ObjectClosure<?> input;
  private CtClass returnType;
```

```
21    private List<Op> ops = null;
22
23    ImmListLCompiler(Environment.Binder binder) { this.binder = binder; }
24
25    public ObjectClosure getClosure() {
26      if (ops == null) { return input; }
27
28      ClassPool cp = ClassPool.getDefault();
29      try {
30        CtClass cloClazz = cp.makeClass(
31          ImmListLCompiler.class.getName()+ "$FusedOps"
32        );
33        cloClazz.setModifiers(Modifier.PUBLIC | Modifier.FINAL);
34        cloClazz.setInterfaces(
35          new CtClass[]{ cp.get(ObjectClosure.class.getName()) }
36        );
37
38        CtField field = CtField.make(
39          "public static " + ObjectClosure.class.getName() + " c0;",
40          cloClazz
41        );
42        cloClazz.addField(field);
43        for (int i = 0; i < ops.size(); i++) {
44          field = CtField.make(
45            "public static "
46            + ObjectClosure.class.getName()
47            + " c" + (i + 1) + ";",
48            cloClazz
49          );
50          cloClazz.addField(field);
51        }
52
53        StringBuilder cloSource = new StringBuilder();
54        cloSource.append(
55          "public Object evaluate("
56          + Environment.class.getName()
57          + " env) {\n"
58        );
59        for (int i = 0; i < ops.size(); i++) {
60          if (ops.get(i) instanceof TransformOp) {
61            cloSource.append(
62              "  " + Function.class.getName()
63              + " f" + (i + 1) + " = ("
64              + Function.class.getName() + ") c"
65              + (i + 1) + ".evaluate(env);\n"
66            );
67          } else if (ops.get(i) instanceof FilterOp) {
68            cloSource.append(
69              "  " + Predicate.class.getName()
70              + " p" + (i + 1) + " = ("
71              + Predicate.class.getName() + ") c"
72              + (i + 1) + ".evaluate(env);\n");
73          }
74        }
75        cloSource.append(
76          "  " + returnType.getName()
77          + "$Builder builder = " + returnType.getName()
```

```
78          + ".builder();\n"
79        );
80        cloSource.append(
81          "  " + Iterator.class.getName()
82          + " iterator = ((" + Iterable.class.getName()
83          + ") c0.evaluate(env)).iterator();\n"
84        );
85        cloSource.append(
86          "  while (iterator.hasNext()) {\n"
87          + "    Object o = iterator.next();\n"
88        );
89        for (int i = 0; i < ops.size(); i++) {
90          if (ops.get(i) instanceof TransformOp) {
91            cloSource.append(
92              "    o = f" + (i + 1)
93              + ".apply(o);\n"
94            );
95          } else if (ops.get(i) instanceof FilterOp) {
96            cloSource.append(
97              "    if (!p" + (i + 1)
98              + ".apply(o)) { continue; }\n"
99            );
100         }
101       }
102       cloSource.append(
103         "    builder.add(o);\n"
104         + "  }\n"
105         + "  return builder.build();\n"
106         + "}");
107       CtMethod cloMethod = CtNewMethod.make(
108         cloSource.toString(),
109         cloClazz
110       );
111       cloClazz.addMethod(cloMethod);
112
113       CtClassLoader loader = new CtClassLoader();
114       Class<?> cloC = loader.load(cloClazz);
115       cloClazz.detach();
116
117       Field f = cloC.getDeclaredField("c0");
118       f.set(null, input);
119       for (int i = 0; i < ops.size(); i++) {
120         f = cloC.getDeclaredField("c" + (i + 1));
121         f.set(null, ops.get(i).closure);
122       }
123
124       return (ObjectClosure) cloC.newInstance();
125     } catch (
126         NotFoundException |
127         CannotCompileException |
128         InstantiationException |
129         IllegalAccessException |
130         NoSuchFieldException |
131         IOException e) {
132       throw new RuntimeException(e);
133     }
134   }
```

```
135
136   public void visit(Expression.FieldRead staged) { }
137   public void visit(Expression.FieldAssignment staged) { }
138   public void visit(Expression.MethodInvocation staged) {
139     CtMethod method = staged.getMember();
140     if (returnType == null) {
141       try {
142         returnType = method.getReturnType();
143       } catch (NotFoundException e) {
144         throw new RuntimeException(e);
145       }
146     }
147
148     staged.getArgument(0).accept(this);
149     ObjectClosure<?> arg0 = closure;
150     if (input == null) {
151       input = arg0;
152     }
153
154     if (staged.getArgumentCount() > 1) {
155       if (ops == null) {
156         ops = new LinkedList<>();
157       }
158
159       staged.getArgument(1).accept(this);
160       ObjectClosure<?> arg1 = closure;
161
162       switch (method.getName()) {
163         case "map": {
164           ops.add(
165             new TransformOp((ObjectClosure<Function<Object, ?>>) arg1)
166           );
167           break;
168         }
169         case "filter": {
170           ops.add(
171             new FilterOp((ObjectClosure<Predicate<Object>>) arg1)
172           );
173           break;
174         }
175       }
176     }
177   }
178
179   public void visit(Expression.ObjectValue value) {
180     closure = value.bind(binder);
181   }
182   public void visit(Expression.BooleanValue value)   { }
183   public void visit(Expression.IntegerValue value)   { }
184   public void visit(Expression.LongValue value)      { }
185   public void visit(Expression.FloatValue value)     { }
186   public void visit(Expression.DoubleValue value)    { }
187   public void visit(Expression.ByteValue value)      { }
188   public void visit(Expression.CharacterValue value) { }
189   public void visit(Expression.ShortValue value)     { }
190 }
```

B.4 Vector EDSL Overhead Experiment

Listings B.5 and B.5 show the generation of vectors and the benchmark loops.

LISTING B.5: Overhead (`Vec`) benchmark (`run`)

```
public static long run(int size) {
  Random r = new Random(4);
  Vec a = Vec.create(r.doubles(size).toArray());
  Vec b = Vec.create(r.doubles(size).toArray());
  Vec c = Vec.create(r.doubles(size).toArray());

  Vec res = null;

  TickTock.tick();
  for (int i = 0; i < 100000; i++) {
    res = example(i % 20, a, b, c);
  }
  long t = TickTock.tock();
  ...
  return t;
}
```

LISTING B.6: Overhead (`Vec`) benchmark

```
// Warm-up
run(10000);

// Measurement
for (int i = 0; i < 5; i++) {
  int size = (int) Math.pow(10, i);
  System.out.println(size);

  long[] times = new long[10];
  for (int j = 0; j < 10; j++) {
    times[j] = run(size);
  }

  out.println(Arrays.toString(times));
}
```

APPENDIX C

Tame-Staging Reference

Chapter 4 introduces the overview of the-tame staging framework for language embedding as an almost first-class feature. By itself that should serve as a good introduction on its effects on EDSL users and the framework's usage by EDSL authors. In order to supplement that overview, this appendix provides a minimal reference (documentation) for the public interfaces of the framework.

C.1 `Language` Classes

The `Language` interface has to be assigned to language-representing classes and is used by the majority of the framework's annotations. It is a simple marker interface defined as follows:

```
public interface Language<T extends Language<T>> { }
```

This interface does (and could) not impose any implementations constraints (e.g. signatures) on the classes defined by EDSL authors. Language classes have to implement certain (i.e. expected-signature) static methods, which (currently) cannot be governed by Java (instance) interfaces. The same concept can be found in Java's APIs, most famously in the `main` method as an application's entry point as well as for instance the `premain` method of Java agents.

The methods to be defined depend on the set of (return) types of `@Stage`-annotated EDSL tokens. If a language has no EDSL tokens, no static method has to be implemented, but commonly such an EDSL would of course not be very useful. The opposite, maximum case is found in an EDSL that has tokens of all nine (or eight) primitive Java types (**`void`**, **`boolean`**, **`int`**, **`long`**, **`float`**, **`double`**, **`byte`**, **`char`**, **`short`**) and any number of reference types (i.e. subtypes of `Object`). In this case the methods are:

- `VoidClosure makeVoidClosure(⋯)`
- `BooleanClosure makeBooleanClosure(⋯)`

- `IntegerClosure makeIntegerClosure(…)`
- `LongClosure makeLongClosure(…)`
- `FloatClosure makeFloatClosure(…)`
- `DoubleClosure makeDoubleClosure(…)`
- `ByteClosure makeByteClosure(…)`
- `CharacterClosure makeCharacterClosure(…)`
- `ShortClosure makeShortClosure(…)`
- `ObjectClosure makeObjectClosure(…)`

Their three respective parameters are (in order):

- `Expression.Staged staged`: The root node of the expression DAG for which processing has been triggered (via the indirection of materialization).
- `Environment.Binder binder`: The binder instance to be used for creating value-access closures for use in the returned computation.
- **`boolean`** `permCached`: Indicates whether the returned closure will be permanently associated with a static EDSL-program materialization situation instead of being only temporarily cached.

More documentation on processing and the closures to be returned are described in a later section.

C.2 Annotations

C.2.1 `@Stage`

The `@Stage` annotation is to be placed on methods or fields to be designated as tokens of an EDSL. Its parameters are:

- `Class<? extends Language<?>> language`: The associated EDSL class.
- **`boolean`** `isStrict`: Indicates whether using the token causes immediate materialization (with the token's representation as the root of the expression DAG to be processed). The default value is **`false`**.
- `StaticInfo.Element[] staticInfoElements`: The kinds of static-context information to be retained. The default value is `{}` (empty array).

C.2.1.1 Definition

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD, ElementType.FIELD })
public @interface Stage {
  Class<? extends Language<?>> language();
  boolean                      isStrict()           default false;
  StaticInfo.Element[]         staticInfoElements() default    {};
}
```

C.2.2 `@Accept` and `@Accept.This`

The `@Accept` annotation is to be placed on the parameters of EDSL-token methods or fields to indicate cross-language acceptance of staged terms. It is only effective in combination with `@Stage` and ignored otherwise. Its parameter is:

- `Class<? extends Language<?>>[] languages`: The set (i.e. duplicates are ignored) of EDSL classes whose terms are accepted.

The inner `@Accept.This` annotation serves the same purpose for the implicit **`this`** parameter of instance-method calls and instance-field accesses. On static methods it has no effect.

C.2.2.1 Definitions

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.PARAMETER })
public @interface Accept {
  Class<? extends Language<?>>[] languages();

  @Documented
  @Retention(RetentionPolicy.CLASS)
  @Target({ ElementType.FIELD, ElementType.METHOD })
  @interface This {
    Class<? extends Language<?>>[] languages();
  }
}
```

C.2.3 `@Suppress`

The `@Suppress` annotation is to be placed on methods, constructors, and types (e.g. classes) within whose scope the effects of certain `@Stage` annotations are to be ignored. Its parameter is:

- `Class<? extends Language<?>>[] languages`: The set (i.e. duplicates are ignored) of EDSL classes whose tokens are to be ineffective. The original token-construct definitions (e.g. method bodies) and behaviors apply.

C.2.3.1 Definition

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.TYPE })
public @interface Suppress {
  Class<? extends Language<?>>[] languages();
}
```

C.2.4 `@Configure`

The `@Configure` annotation is to be placed on EDSL classes for which a (partial) restriction of public annotation accessibility (or visibility) is to take effect. Accessibility from within the EDSL class's package is not affected. Its parameters are:

- `hasRestrictedAcceptAccessibility`: Restricts the language's accessibility from `@Accept` annotations. The default value is **`false`**.
- `hasRestrictedStageAccessibility`: Restricts the language's accessibility from `@Stage` annotations. The default value is **`false`**.
- `hasRestrictedSuppressAccessibility`: Restricts the language's accessibility from `@Suppress` annotations. The default value is **`false`**.

The name (i.e. "configure") of this annotation was chosen in anticipation of further language-global options in the future.

C.2.4.1 Definition

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE })
public @interface Configure {
  boolean hasRestrictedAcceptAccessibility()   default false;
  boolean hasRestrictedStageAccessibility()    default false;
  boolean hasRestrictedSuppressAccessibility() default false;
}
```

C.3 Expressions

Expression-DAG nodes are of type `Expression`, an abstract class with no public abstract methods. Token-representatives are subtypes of the abstract class `Expression.Staged`, placed into one of yet another three subcategories (as abstract classes):

- `Expression.FieldRead`
- `Expression.FieldAssignment`
- `Expression.MethodInvocation`

Concrete token-representations (unknown at Java-source compile time) implement the following data-access methods:

- `CtMember getMember()`: Returns the Javassist representation of the annotated construct. The subcategories make use of covariant return types to return `CtMethod` or `CtField`.
- `Optional<StaticInfo> getStaticInfo()`: Returns static-context information, if collected.
- **`int`** `getArgumentCount()`: Returns the number of (sub-) argument expressions of the staged term.[1]
- `Expression getArgument(`**`int`**` index)`: Returns the (sub-) argument expression at the given index.

Input-value nodes, i.e. EDSL-external values, are represented by the abstract class `Expression.Value`, which is the common supertype of the following concrete classes corresponding to the various kinds of Java types (excluding **`void`**):

- `Expression.BooleanValue`
- `Expression.IntegerValue`
- `Expression.LongValue`
- `Expression.FloatValue`
- `Expression.DoubleValue`
- `Expression.ByteValue`
- `Expression.CharacterValue`
- `Expression.ShortValue`
- `Expression.ObjectValue`

Their data-access methods are:

- C `bind(Environment.Binder binder)`: Returns an indirect-access closure, which is of the closure type C corresponding to the kind of Java type inhibited by the represented value (e.g. `IntegerClosure` for `IntegerValue`).
- V `inspect(Environment.Binder binder)`: Returns the represented value, which is of the type V corresponding to the kind of Java type inhibited by the represented value (e.g. **`int`** for `IntegerValue`). This will disable caching of the expression DAG's processing result unless the inspected value representation is a constant.
- **`boolean`** `isConstant()`: Indicates whether a represented value derived from a constant or not.

[1] Note that variable argument lists (so-called *varargs*) on token methods are currently unsupported.

In addition to the described data-access mechanisms, specific `accept` methods for visitors are provided so that EDSL authors can easily traverse the expression DAG for processing. The visitor interfaces are as follows:

- `Expression.Staged.Visitor` specifies three methods:
  - **void** `visit(FieldRead staged)`
  - **void** `visit(FieldAssignment staged)`
  - **void** `visit(MethodInvocation staged)`
- `Expression.Value.Visitor` specifies nine methods:
  - **void** `visit(BooleanValue value)`
  - **void** `visit(IntegerValue value)`
  - **void** `visit(LongValue value)`
  - **void** `visit(FloatValue value)`
  - **void** `visit(DoubleValue value)`
  - **void** `visit(ByteValue value)`
  - **void** `visit(CharacterValue value)`
  - **void** `visit(ShortValue value)`
  - **void** `visit(ObjectValue value)`
- `Expression.Visitor` combines (i.e. extends) both of the above.

C.4 Static Information

There are two kinds of static information that can be chosen for collection on `@Stage`-annotated tokens, exposed by the `StaticInfo.Element` enum type:

- `ORIGIN`
- `INFERRED_TYPES`

The former will cause an instance of `StaticInfo.Origin` to be available on `Expression.Staged` nodes for inspection, the latter corresponds to instances of `StaticInfo.InferredTypes`. They are not exclusive.

The `StaticInfo` class provides two methods providing this data (if available):

- `Optional<StaticInfo.Origin> getOrigin()`
- `Optional<StaticInfo.InferredTypes> getInferredTypes()`

Note that there is no inheritance relationship with `StaticInfo`.

C.4.0.2 `StaticInfo.Origin`

The `StaticInfo.Origin` class provides three data-access methods:

- `CtBehavior getBehavior()`: Returns the Javassist behavior (i.e. method, constructor, or initializer) representation in which the associated term was constructed.
- `int getPosition()`: Returns the bytecode position (within the behavior) of the term construction.
- `OptionalInt getLineNumber()`: Returns the source-code position of the term construction. This information might not be available, hence the `OptionalInt` return type.

C.4.0.3 `StaticInfo.InferredTypes`

The `StaticInfo.InferredTypes` class provides two data-access methods:

- `Type getArgumentType(int index)`: Returns the inferred type of the argument at the given index. It is calculated by finding the most general type from all (static) sources of the associated argument.
- `Type getType(int index)`: Returns the inferred (return) type. It is calculated by finding the most specific (return) type from all (static) uses (or consumptions) of the associated method or field.

For calculating the latter, the use-site types are determined by the respective expected types there, except for checked casts for which the output type is used. This means that the most specific (return) type reflects the type that is necessary to (pessimistically) fulfill all the constraints of subsequent casting (as well as non-cast uses). In the worst case only **`null`** values might achieve that. The `Type` type is borrowed from Javassist's `javassist.bytecode.analysis` package.

C.5 Closures

The return values of the static expression-DAG processing methods on an EDSL class (as well as the `bind` methods on value nodes) are closure objects. Although there is a common `Closure` supertype interface (with a concrete default method `evaluateToObject`), only the following interfaces are actually used:

- `VoidClosure`
- `BooleanClosure`
- `IntegerClosure`
- `LongClosure`
- `FloatClosure`
- `DoubleClosure`

- `ByteClosure`
- `CharacterClosure`
- `ShortClosure`
- `ObjectClosure`

Their abstract `evaluate` methods adhere to the following pattern:

- V `evaluate(Environment environment)`: Returns the computation's (materialization) result, which is of the type V corresponding to the closure type (e.g. **`int`** for `IntegerClosure`).

Bibliography

[1] http://www.graphviz.org/ (2015-05-06).

[2] http://www.povray.org/ (2015-05-06).

[3] https://www.python.org/ (2015-05-06).

[4] http://www.scipy.org/ (2015-05-06).

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

[6] http://ocaml.org/ (2015-05-07).

[7] http://stanford-ppl.github.io/Delite/opticvx/index.html (2015-03-19).

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

[9] https://www.ruby-lang.org/ (2015-05-15).

[10] https://downloads.haskell.org/~ghc/7.0.1/docs/html/users_guide/rewrite-rules.html (2015-05-14).

[11] http://racket-lang.org/ (2015-05-16).

[12] http://openjdk.java.net/projects/graal/ (2015-06-16).

[13] http://www.nvidia.com/object/cuda_home_new.html (2015-06-20).

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

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

[16] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[17] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[18] John Aycock. "A Brief History of Just-in-time". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113.

[19] Otto Skrove Bagge, Karl Trygve Kalleberg, Magne Haveraaen, and Eelco Visser. "Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs". In: *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. Ed. by Dave Binkley and Paolo Tonella. Amsterdam, The Netherlands: IEEE Computer Society Press, Sept. 2003, pp. 65–75.

[20] Jon Bentley. "Programming Pearls: Little Languages". In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721.

[21] Joshua Bloch. *Effective Java (Java Series)*. 2nd ed. Boston, MA, USA: Pearson Education, Inc., 2008.

[22] Gilad Bracha and William Cook. "Mixin-based Inheritance". In: *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA/ECOOP '90. Ottawa, Canada: ACM, 1990, pp. 303–311.

[23] Edwin C. Brady. "Idris: General Purpose Programming with Dependent Types". In: *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification*. PLPV '13. Rome, Italy: ACM, 2013, pp. 1–2.

[24] Edwin C. Brady and Kevin Hammond. "Scrapping Your Inefficient Engine: Using Partial Evaluation to Improve Domain-specific Language Implementation". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 297–308.

[25] Eugene Burmako. "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming". In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: ACM, 2013, 3:1–3:10.

[26] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. "Finally Tagless, Partially Evaluated". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Vol. 4807. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 222–238.

[27] William E. Carlson, Paul Hudak, and Mark P. Jones. *An Experiment Using Haskell to Prototype "Geometric Region Servers" for Navy Command And Control*. Tech. rep. Research Report YALEU/DCS/RR-1031. Yale University, 1993.

[28] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language". In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '74. Ann Arbor, Michigan: ACM, 1974, pp. 249–264.

[29] James Cheney and Ralf Hinze. *First-Class Phantom Types*. Tech. rep. Cornell University, 2003.

[30] Shigeru Chiba. "Load-Time Structural Reflection in Java". English. In: *ECOOP 2000 — Object-Oriented Programming*. Ed. by Elisa Bertino. Vol. 1850. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 313–336.

[31] Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. "Static Analysis of Multi-staged Programs via Unstaging Translation". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 81–92.

[32] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. "DSL Implementation in MetaOCaml". In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Vol. 3016. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 51–72.

[33] L. Peter Deutsch and Allan M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302.

[34] Lukas Diekmann and Laurence Tratt. "Eco: A language composition editor". In: *Software Language Engineering (SLE)*. Springer, Sept. 2014, pp. 82–101.

[35] Conal Elliott. "Functional Images". In: *The Fun of Programming*. Cornerstones of computing. Palgrave Macmillan, Mar. 2003.

[36] Conal Elliott, Sigbjørn Finne, and Oege De Moor. "Compiling Embedded Languages". In: *J. Funct. Program.* 13.3 (May 2003), pp. 455–481.

[37] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97. Amsterdam, The Netherlands: ACM, 1997, pp. 263–273.

[38] Martin Fowler. *Domain Specific Languages*. 1st ed. Addison-Wesley Professional, 2010.

[39] Steve Freeman and Nat 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. Portland, Oregon, USA: ACM, 2006, pp. 855–865.

[40] Yoshihiko Futamura. "Partial Evaluation of Computation Process – An approach to a Compiler-Compiler". In: *Transactions of the Institute of Electronics and Communication Engineers of Japan* Vol.54-C.8 (Aug. 1971), pp. 721–728.

[41] Yoshihiko Futamura. "Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler". In: *Higher-Order and Symbolic Computation* 12.4 (1999), pp. 381–391.

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[43] Steven E. Ganz, Amr Sabry, and Walid Taha. "Macros As Multi-stage Computations: Type-safe, Generative, Binding Macros in MacroML". In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ICFP '01. Florence, Italy: ACM, 2001, pp. 74–85.

[44] Debasish Ghosh. *DSLs in Action*. 1st ed. Greenwich, CT, USA: Manning Publications Co., 2010.

[45] Paolo G. Giarrusso, Klaus Ostermann, Michael Eichberg, Ralf Mitschke, Tillmann Rendel, and Christian Kästner. "Reify Your Collection Queries for Modularity and Speed!" In: *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*. AOSD '13. Fukuoka, Japan: ACM, 2013, pp. 1–12.

[46] Jeremy Gibbons and Nicolas Wu. "Folding Domain-specific Languages: Deep and Shallow Embeddings". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 339–347.

[47] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[48] James Gosling. "Java Intermediate Bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (IR'95)". In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. IR '95. San Francisco, California, USA: ACM, 1995, pp. 111–118.

[49] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996.

[50] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st ed. Addison-Wesley Professional, 2014.

[51] Maria Gouseti, Chiel Peters, and Tijs van der Storm. "Extensible Language Implementation with Object Algebras (Short Paper)". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. Västerås, Sweden: ACM, 2014, pp. 25–28.

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

[53] Samuel Z. Guyer and Calvin Lin. "Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 342–357.

[54] Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. "The Glasgow Haskell Compiler: A Retrospective". In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. London, UK: Springer-Verlag, 1993, pp. 62–71.

[55] Ben Hardekopf and Calvin Lin. "Flow-sensitive Pointer Analysis for Millions of Lines of Code". In: *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 289–298.

[56] Erik Hilsdale and Jim Hugunin. "Advice Weaving in AspectJ". In: *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. AOSD '04. Lancaster, UK: ACM, 2004, pp. 26–35.

[57] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. "Polymorphic Embedding of DSLs". In: *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 137–148.

[58] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP '91. London, UK, UK: Springer-Verlag, 1991, pp. 21–38.

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

[60] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pp. 1–55.

[61] John Hughes. "The Design of a Pretty-printing Library". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK, UK: Springer-Verlag, 1995, pp. 53–96.

[62] Graham Hutton. *Programming in Haskell*. 1st ed. Cambridge University Press, Jan. 2007.

[63] Kazuhiro Ichikawa and Shigeru Chiba. "Composable User-defined Operators That Can Express User-defined Literals". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland: ACM, 2014, pp. 13–24.

[64] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. "A Study of Devirtualization Techniques for a Java Just-In-Time Compiler". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Minneapolis, Minnesota, USA: ACM, 2000, pp. 294–310.

[65] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[66] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[67] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. "Staged Parser Combinators for Efficient Data Processing". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 637–653.

[68] Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. "Yin-Yang: Concealing the Deep Embedding of DSLs". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2014. Västerås, Sweden: ACM, 2014, pp. 73–82.

[69] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Prentice Hall Professional Technical Reference, 1988.

[70] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353.

[71] Gary 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. Boston, Massachusetts: ACM, 1973, pp. 194–206.

[72] Eugene E. Kohlbecker. "Syntactic Extensions in the Programming Language LISP". UMI Order No. GAX86-27998. PhD thesis. Bloomington, IN, USA, 1986.

[73] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. "JavaScript as an Embedded DSL". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. 2012, pp. 409–434.

[74] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Tech. rep. UU-CS-2001-35. Department of Information and Computing Sciences, Utrecht University, 2001. URL: `http://www.cs.uu.nl/research/techreps/repo/CS-2001/2001-35.pdf`.

[75] John McCarthy, R. Brayton, Daniel J. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell. *LISP I Programmer's Manual*. Tech. rep. Cambridge, Massachusetts: Massachusetts Institute of Technology – Computation Center and Research Laboratory, Mar. 1960. URL: `http://history.siam.org/sup/Fox_1960_LISP.pdf`.

[76] Adriaan Moors, Frank Piessens, and Martin Odersky. "Generics of a Higher Kind". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 423–438.

[77] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. "Evaluating the Design of the R Language: Objects and Functions for Data Analysis". In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 104–131.

[78] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[79] Ulf Norell. "Dependently Typed Programming in Agda". In: *Proceedings of the 6th International Conference on Advanced Functional Programming*. AFP'08. Heijen, The Netherlands: Springer-Verlag, 2009, pp. 230–266.

[80] Martin Odersky and Tiark Rompf. "Unifying Functional and Object-oriented Programming with Scala". In: *Commun. ACM* 57.4 (Apr. 2014), pp. 76–86.

[81] Martin Odersky and Matthias Zenger. "Scalable Component Abstractions". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 41–57.

[82] Bruno C. d. S. Oliveira and William R. Cook. "Extensibility for the Masses: Practical Extensibility with Object Algebras". In: *Proceedings of the 26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 2–27.

[83] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. "Safely Composable Type-Specific Languages". English. In: *ECOOP 2014 - Object-Oriented Programming*. Ed. by Richard Jones. Vol. 8586. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 105–130.

[84] Terence J. Parr and Russel W. Quong. "ANTLR: A predicated-LL(K) Parser Generator". In: *Softw. Pract. Exper.* 25.7 (July 1995), pp. 789–810.

[85] Frank Pfenning and Conal Elliot. "Higher-Order Abstract Syntax". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 199–208.

[86] John C. Reynolds. "Definitional Interpreters for Higher-order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM '72. Boston, Massachusetts, USA: ACM, 1972, pp. 717–740.

[87] Tiark Rompf and Martin 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. Eindhoven, The Netherlands: ACM, 2010, pp. 127–136.

[88] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Building-Blocks for Performance Oriented DSLs". In: *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011*. 2011, pp. 93–117.

[89] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. "Scala-Virtualized: Linguistic Reuse for Deep Embeddings". In: *Higher-Order and Symbolic Computation* 25.1 (2012), pp. 165–207.

[90] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. "Surgical Precision JIT Compilers". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 41–52.

[91] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 12–27.

[92] Maximilian Scherr and Shigeru Chiba. "Almost First-Class Language Embedding: Taming Staged Embedded DSLs". In: *Proceedings of the 2015 International Conference on Generative Programming: Concepts and Experiences*. GPCE 2015. To appear. New York, NY, USA: ACM, 2015.

[93] Maximilian Scherr and Shigeru Chiba. "Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Vol. 8586. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 385–410.

[94] Jonathan L. Schilling. "The Simplest Heuristics May Be the Best in Java JIT Compilers". In: *SIGPLAN Not.* 38.2 (Feb. 2003), pp. 36–46.

[95] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. "Optimising Embedded DSLs Using Template Haskell". In: *Generative Programming and Component Engineering*. Ed. by Gabor Karsai and Eelco Visser. Vol. 3286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 186–205.

[96] Tim Sheard and Simon Peyton Jones. "Template Meta-programming for Haskell". In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 60–75.

[97] Guy L. Steele Jr. and Richard P. Gabriel. "The Evolution of Lisp". In: *SIGPLAN Not.* 28.3 (Mar. 1993), pp. 231–270.

[98] Robert E. Strom and Shaula Yemini. "Typestate: A Programming Language Concept for Enhancing Software Reliability". In: *IEEE Transactions on Software Engineering* SE-12.1 (Jan. 1986), pp. 157–171.

[99] Bjarne Stroustrup. "A C++ Tutorial". In: *Proceedings of the 1985 ACM Annual Conference on The Range of Computing : Mid-80's Perspective: Mid-80's Perspective*. ACM '85. Denver, Colorado, USA: ACM, 1985, pp. 56–64.

[100] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 134:1–134:25.

[101] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning". In: *Proceedings of the 28th International Conference on Machine Learning*. ICML 2011. 2011.

[102] Josef Svenningsson and Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". In: *Trends in Functional Programming*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Vol. 7829. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 21–36.

[103] Walid Taha. "A Gentle Introduction to Multi-stage Programming". In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Vol. 3016. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 30–50.

[104] Walid Taha and Tim Sheard. "MetaML and Multi-stage Programming with Explicit Annotations". In: *Theor. Comput. Sci.* 248.1-2 (Oct. 2000), pp. 211–242.

[105] Walid Mohamed Taha. "Multistage Programming: Its Theory and Applications". AAI9949870. PhD thesis. 1999.

[106] Simon Thompson. *Haskell: The Craft of Functional Programming*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2011.

[107] Sam Tobin-Hochstadt and Matthias Felleisen. "The Design and Implementation of Typed Scheme". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 395–406.

[108] Mads Torgersen. "Language Integrated Query: Unified Querying Across Data Sources and Programming Languages". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 736–737.

[109] Laurence Tratt. "Domain Specific Language Implementation via Compile-Time Meta-Programming". In: *TOPLAS* 30.6 (2008), pp. 1–40.

[110] Jeffrey D. Ullman. *Elements of ML Programming (ML97 Ed.)* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[111] Todd Veldhuizen. "C++ Gems". In: ed. by Stanley B. Lippman. New York, NY, USA: SIGS Publications, Inc., 1996. Chap. Expression Templates, pp. 475–487.

[112] Todd L. Veldhuizen and Dennis Gannon. "Active Libraries: Rethinking the roles of compilers and libraries". In: *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[113] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 60–76.

[114] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. "Prolog - The Language and Its Implementation Compared with Lisp". In: *SIGPLAN Not.* 12.8 (Aug. 1977), pp. 109–115.

[115] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid 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. Toronto, Ontario, Canada: ACM, 2010, pp. 400–411.

[116] Hongwei Xi, Chiyan Chen, and Gang Chen. "Guarded Recursive Datatype Constructors". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '03. New Orleans, Louisiana, USA: ACM, 2003, pp. 224–235.

[117] Hao Xu. "EriLex: An Embedded Domain Specific Language Generator". English. In: *Objects, Models, Components, Patterns*. Ed. by Jan Vitek. Vol. 6141. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 192–212.