Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding*

Maximilian Scherr and Shigeru Chiba

The University of Tokyo, Japan
scherr@csg.ci.i.u-tokyo.ac.jp
chiba@acm.org

Abstract. Common implementation approaches for embedding DSLs in general-purpose host languages force developers to choose between a *shallow* (single-staged) embedding which offers seamless usage, but limits DSL developers, or a *deep* (multi-staged) embedding which offers freedom to optimize at will, but is less seamless to use and incurs additional runtime overhead. We propose a metaprogrammatic approach for extracting domain-specific programs from user programs for custom processing. This allows for similar optimization options as deep embedding, while still allowing for seamless embedded usage. We have implemented a simplified instance of this approach in a prototype framework for Java-embedded EDSL expressions, which relies on load-time reflection for improved deployability and usability.

Keywords: DSL, metaprogramming, Java, programming languages

1 Introduction

In recent years, the study of domain-specific languages (DSLs) and the investigation of their usage and implementation methods have attracted increasing interest. These languages, which are limited in scope and tailored to a specific problem domain, are said to be easier to reason about and maintain, and open the door to domain-specific optimizations [1].

One form of DSL implementation of particular interest is the embedding of DSLs by means of the available language constructs of an enclosing general-purpose programming language. These embedded DSLs (EDSLs) bring several advantages to the table. For one, sizable parts of the existing tool and general language support (e.g. syntactic and semantic analysis) can be inherited from the host language [1]. More importantly, they enable the embedded usage of

* This is a preprint version of the paper that appeared in the proceedings of ECOOP 2014, LNCS 8586, pp. 385-410, DOI: 10.1007/978-3-662-44202-9_16. The original publication is available at www.springerlink.com (http://link.springer.com/chapter/10.1007%2F978-3-662-44202-9_16).

DSL programs side by side with host language code. As they have a look-and-feel similar to the host language code, they can be approached by programmers in a fashion similar to using traditional libraries.

Recent examples of such EDSLs in object-oriented programming are JMock [2], Guava's fluent APIs (e.g. `FluentIterable`, `Splitter`, etc.) [3], SQuOpt [4], and jOOQ [5].

When using only facilities expressible within the host language, an EDSL developer may commonly approach pure language embedding, i.e. without "preprocessor, macro-expander, or generator" [1], in one of two different fashions:

- Execution of the atomic surface elements of the EDSL is directly governed by the semantics of the host language. Evaluation of DSL programs occurs immediately in small steps, yielding and passing intermediate results. This implementation approach is called *shallow embedding*.
- Execution of the atomic surface elements of the EDSL first produces (or *stages*) an intermediate representation (e.g. AST) of the expressed program or snippet. Evaluation to final result values occurs separately. This approach is called *deep embedding*.

Hybrid forms are also possible, where only selected parts of an EDSL are deeply embedded. Intuitively, it may help to think of depth here as a measure of freedom of EDSL programs from the host language's semantics. This has implications on the degree of expressiveness and ability to optimize domain-specific computation. In section 2 we describe the trade-offs between these two approaches.

The strength of deep embedding lies in the fact that computation is staged, allowing for intermediate, customized processing. However, staging commonly occurs in an explicit fashion with the potential to detract users and cause overhead. In order to approximate this staging without incurring penalties we propose a method called *implicit staging*, which takes the form of a framework to be employed by EDSL developers. Section 3 outlines this approach in general.

The main idea behind implicit staging is to statically extract domain-specific code before it is executed by means of static analysis, in particular abstract interpretation. This yields a representation of the domain-specific code that can be processed by an EDSL's developer in a customized fashion. We make the following contributions:

- Implicit staging is a method to channel the processing of domain-specific computation within its static context by semi-automatically isolating it from general-purpose code.
- We present how increasing the amount of contextual information bears the potential for rich optimizations that take into account the intermixed nature of both shallow as well as deep EDSLs.
- In order to concretely illustrate and evaluate our approach, we implemented a proof-of-concept framework using load-time reflection [6] for the Java language. It enables implicit staging of compound EDSL expressions and is mainly focused on bridging the gap between shallow and deep embedding. We present this implementation in section 4 and its evaluation in section 5.

– The prototype shows that our approach is feasible even without full source code availability and that even basic data-flow analysis suffices to extract worthwhile portions of EDSL subprograms.

2 Implementation of Embedded DSLs

In our treatment of EDSLs, provided as libraries, we distinguish between three main roles (cf. figure 1): The *developer* (alternatively *implementor* or *provider*) of the EDSL defines the interface, implements the language behavior, and writes its documentation. The *user* of the EDSL is any developer who employs the EDSL (directly or indirectly) to support the implementation of programs. The *end user* then is anyone who actually causes the execution of these programs.

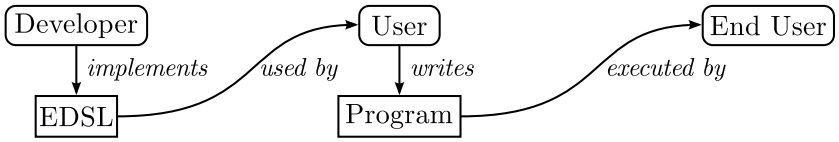

Fig. 1. EDSL implementation and usage roles

Depending on the choice of host language, the choice of basic building bricks of an EDSL varies. For instance, in modern ML variants and Haskell these are commonly data constructor and function applications. In Java they are mainly field accesses and method calls. In our treatment we simply call these atomic surface elements the *tokens* of an EDSL. Furthermore, in statically typed host languages the introduction of EDSL-specific types and the restriction of type signatures in effect allow developers to restrict certain combinations of EDSL tokens, i.e. the EDSL's specific syntax. It is mainly the runtime behavior of these tokens that defines the concrete nature of the embedding.

In the following, we introduce shallow and deep embedding [7] by example of a simple EDSL for matrix operations in Java. A materialized matrix is represented by a `Matrix` data type which wraps a two-dimensional `double` array. We assume the existence of factory methods for creating matrices from given elements. We consider only three tokens: static methods for matrix addition (`add`) and multiplication (`mul`), and a static method to stand for a uniform scaling matrix (`sca`), i.e. a diagonal matrix of a given dimension and a scaling factor on its diagonal.

2.1 Shallow Embedding

When operations of an embedded language are directly mapped to equivalent operations of its host language the embedding is said to be shallow. This means that EDSL tokens both represent a domain-specific operation and their implementation or meaning is to immediately perform it. In a shallow embedding of

our matrix EDSL this means that the `add` method is implemented to take in two matrices and return a new one with added elements. The case of matrix multiplication is analogous to addition, and the `sca` method creates an actual scaling matrix. Listing 1 shows a simple usage example.

Listing 1. Shallow EDSL usage example

```
1  Matrix a, b, c;
2  // Omission
3  Matrix d = add(add(a, b), mul(c, sca(5, 3.0)));
4  Matrix e = add(a, d);
```

The advantage from an EDSL user's point of view lies in the fact that the behavior of an EDSL expression is easily predictable right where it appears in the code. The expression `add(add(a, b), mul(c, sca(5, 3.0)))` on line 3 yields a new matrix, and does nothing more (or less). The fact that intermediate results are created may be detrimental for the runtime performance, but not for the understanding of that code line. This makes the usage of shallow embedded DSLs very seamless.

From an EDSL developer's point of view shallow embedding is easy to implement. However, this advantage is outweighed by the limitations due to having to directly adhere to the host language's semantics. In particular, this means that optimizations such as common subexpression elimination or fusion of operations cannot be implemented, execution cannot be chunked and scheduled in sizes worthwhile for parallel execution, and execution is bound to occur on the same machine and architecture that executes the host program.

2.2 Deep Embedding: Staging at Runtime

Instead of directly mapping tokens to equivalent host language operations, it is possible to make them generate an intermediate representation (IR), usually an AST. Here, the sole purpose of token execution is to contribute to building and composing the next stage of computation, i.e. staging. This IR can be processed (e.g. optimized, transformed, compiled, etc.) in a separate step and subsequently executed. This typically yields the end result of the expressed computation. However, it is also possible for the result to be a further stage. A case for this would be a program P_1 whose evaluation generates a program P_2 whose evaluation generates a program P_3, and so on.

For a deep embedding of our matrix EDSL we could use an abstract data type `MatrixExp` with concrete data types for the different AST node types. The token methods are implemented to create corresponding nodes. Instead of taking arguments of type `Matrix`, the `add` and `mul` methods now take arguments of type `MatrixExp`. We add a method `cnst` to create an AST node that will evaluate to a provided (materialized) matrix.

Listing 2 shows a simple usage example. Lines 3–4 show the rough equivalent of listing 1, lines 6–7 show a different usage of the EDSL in which the multi-staged nature of the embedding is more apparent.

Listing 2. Deep EDSL usage example

```
1  Matrix a, b, c;
2  // Omission
3  Matrix d = add(add(cnst(a), cnst(b)), mul(cnst(c), sca(5, 3.0))).evaluate();
4  Matrix e = add(cnst(a), cnst(d)).evaluate();
5  // Omission
6  MatrixExp dExp = add(add(cnst(a), cnst(b)), mul(cnst(c), sca(5, 3.0)));
7  Matrix f = add(cnst(a), dExp).evaluate();
```

When the `evaluate` method is called, the entire EDSL program has already been staged and can be fully inspected. This enables domain-specific optimizations, alternative interpretations, or compilation to a possibly different target language and execution. For instance, nested binary additions can be specialized to a flattened addition which does not produce fully materialized intermediate results.

There are potential downsides to deep embedding. Depending on the host language, it can be hard to hide from the programmer the fact that computation is staged. This may sometimes be desirable, but when not it arguably adds an additional layer of complexity for code understanding, especially if the ability to dynamically create and pass around computation (e.g. ASTs) is abused by users. Furthermore, EDSL developers have to build data structures for their specific IR and make the tokens generate the correct IR nodes.

From a runtime performance point of view, the overhead associated with IR construction, in particular the IR's memory footprint, optimization, and interpretation (or compilation) need to be carefully considered. After all, an EDSL developer has no picture of and no influence on how the EDSL's users place EDSL expressions and trigger their evaluation in their programs.

2.3 No Middle Ground?

It is no surprise that deep embedding allows for much more powerful and expressive EDSLs than shallow embedding. Essentially, it can be seen as "just" providing an elegant way to explicitly perform (domain-specific) code generation and execution at runtime. On the other hand, shallow embedding offers a more immediate and seamless usage than deep embedding. However, this immediate usage relies on the immediate execution by the host language, limiting EDSL performance and expressiveness.

Both have in common the fact that it is reasonable to assume that snippets of EDSL programs do in fact occur as static, compound expressions. However, EDSL developers are unable to exploit this fact with either of the embedding styles. Without the ability to do so, a true alternative between shallow and deep embedding seems unattainable.

3 Implicit Staging

Custom treatment of EDSL programs, the crucial step for optimization, occurs after a representation for them has been constructed. In the case of shallow embedding, this step is never really allowed to happen. Although a sort of representation is in fact implicitly constructed during compile time or interpretation

time, commonly there is neither awareness of what constitutes domain-specific code, nor is it possible to customize its processing. As described, deep embedding can be used to circumvent this. However, it means explicit IR construction, explicit processing, and explicit triggering of execution at runtime.[1]

We propose an approach called *implicit staging* that aims to reduce explicitness to a minimum where it is a hindrance, i.e. the IR construction and execution triggering, and retains it where it is desirable, i.e. customized IR processing, while not changing the way EDSL programs are expressed by EDSL users.

Unlike the described pure embedding approaches it is an impure approach. Namely, it requires an outside, static, meta-level view and transformations on user programs. With the exception of languages which allow arbitrary self-modification, implicit staging can typically occur only once before the execution of a program. Figure 2 shows the general 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 EDSL developers 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:** The residue is reflected within the original program, yielding a new, transformed program.

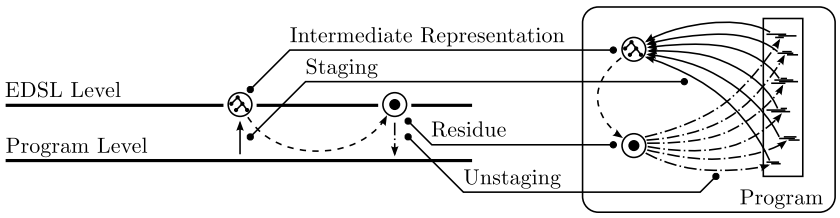

Fig. 2. Implicit staging overview

In principal, the staging step should be able to be performed on the basis of a simple description or enumeration of the EDSL's tokens by its developer. The processing step is to be entirely defined and customized by EDSL developers to form a desired residue suitable for unstaging. Like staging, the unstaging step might also be guided by configuration, which is to be kept simple. Hence, implicit staging lends itself to be provided in the form of a framework to EDSL developers for whom it performs the staging and unstaging tasks.

[1] Though in practice some of these steps can be combined, the separation of steps here is helpful for a high-level discussion.

3.1 Staging by Static Token Reinterpretation

The extraction of domain-specific code can be approached by statically simulating runtime staging behavior to varying degrees. Since EDSL tokens are elements of the host language, they commonly have a defined runtime behavior. 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 runtime 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. For instance, this might include type information, uses and definitions, or value ranges. During processing of the IR, developers can 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.

3.2 The Approach's Potential

Implicit staging provides the basis for exposing a non-atomic, non-local, or even global view on domain-specific computation. 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))`, which yields a materialized matrix, the common subexpression can be eliminated during processing. This is the middle ground we were looking for. However, implicit staging does not necessarily stop there. In fact, it is 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 the described pure embedding approaches on their own. After all, even with deep embedding, what can be inspected during (runtime) IR processing is only what has been dynamically constructed. For instance, in the deeply embedded expression `add(mul(cnst(a), cnst(b), mul(cnst(a), cnst(b)))` the fact that there is a common subexpression may eventually be discovered (during processing steps at runtime), but this is redundantly and possibly repeatedly done. With implicit staging it can be optimized in the residue to help reduce runtime staging overhead.

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 dedicated DSLs, embedded DSL code snippets live within a general purpose program with its own data flow and control flow. With an appropriate interface for EDSL developers, global optimizations could be applied to EDSL programs which are intermixed and dispersed in user code.

Sometimes, the purpose of dynamically staging an EDSL program at runtime 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 our deeply embedded matrix DSL from section 2.2 and consider the user code in listing 3. It might be wise to ever so slightly alter the surrounding user program 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.

Listing 3. Deep EDSL context example (eager)

```
1  MatrixExp aExp, bExp, cExp, dExp;
2  // Omission
3  Matrix e = add(aExp, bExp).evaluate();
4  System.out.println(mul(cnst(e), add(cExp, dExp)).evaluate());
5  System.out.println(e);
```

Listing 4 shows such a lazier version. However, if we (statically) knew that `evaluate` does not perform optimizations in this situation, it might be worthwhile to stay with the eager version of listing 3 or make different changes.

Listing 4. Deep EDSL context example (lazy)

```
1  MatrixExp aExp, bExp, cExp, dExp;
2  // Omission
3  MatrixExp eExp = add(aExp, bExp);
4  System.out.println(mul(eExp, add(cExp, dExp)).evaluate());
5  System.out.println(eExp.evaluate());
```

It is our 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 `evaluate` method).

3.3 Design Aspects

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

- The choice of host language determines the type of 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 suited. Generally speaking, any language that makes static code analysis hard is 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 dedicated (pre-)processing of source code is an option, it usually comes with restrictions regarding the deployment 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 forces an early 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 developers. This is the main hurdle anticipated for fully realizing the vision outlined in section 3.2.

4 Implicit Staging at Load Time

In order to concretely illustrate and evaluate implicit staging, we developed a simple and limited proof-of-concept framework for DSLs embedded in Java. As indicated in section 3, implicit staging does not necessarily have to occur at compile time. Java serves well to show this, as it is a language environment where compilation, class loading, and runtime are closely related. Compilation results in bytecode [8] which retains sufficient language-level information (e.g. method names) and its loading occurs on demand at runtime.

Java neither has compile-time metaprogramming facilities, 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). Choosing load time as the time for performing implicit staging has the following advantages:

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

Runtime System Specialization: User programs and contained EDSL expressions in bytecode remain as is until they are loaded on a specific runtime system. The processing of their IR can be specialized dynamically to that runtime 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 runtime environment (including the heap) as the staging, IR processing, and unstaging steps. This establishes *cross-stage persistence* [9,10,11] which grants EDSL developers certain freedoms and ease of use. For instance, in our implementation it is used to provide a simple interface for returning the results of IR processing as live objects.

EDSL Deployment and Evolution: Any upgrade or patch of an EDSL's implementation as well as of our 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.

Working with Java comes with the issues of late binding (i.e. virtual method calls) 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.

4.1 Prototype Overview

In the following we will describe the components, interfaces, and workflow of our prototype implementation. We designed it with a focus on enabling optimization and semantic customization of (mostly local) compound EDSL expressions in user programs, in order to offer a bridge between shallow and deep embedding for EDSL developers. Compound EDSL expressions are individual Java expressions that are composed of EDSL tokens either in a nested or chained fashion.

Aside from providing skeleton token implementations, an EDSL's developer 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 with the custom processing step mentioned in section 3.

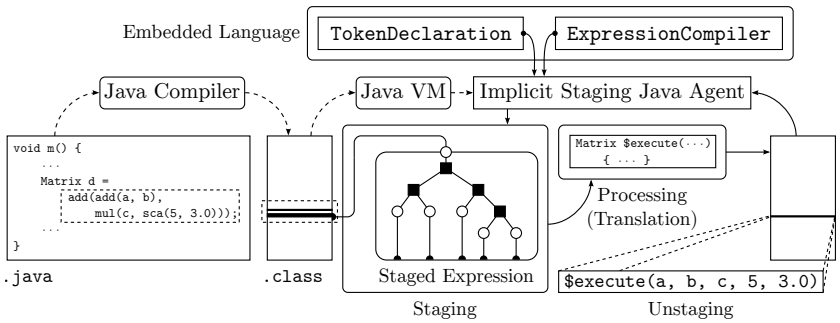

Fig. 3. Java prototype overview

Figure 3 shows a simplified, combined workflow for the usage as well as the inner workings of our prototype. EDSL users, e.g. application developers, may write EDSL expressions as they traditionally could with pure embedding approaches. After all, the tokens (i.e. methods or method calls) may exist independently of our framework. User programs are compiled as usual and deployed with application startup configured to use the implicit staging framework.

At the core of our prototype lies a custom *Java programming language agent* (of `java.lang.instrument`) which intercepts class 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 staging, 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 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.

4.2 Staging: Expression Extraction

The staging process is configured and guided by the *token declaration* of an EDSL. It specifies which methods[2] 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.

Our implementation 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<T>`, provided by the Javassist library [6] used in our implementation. Additionally, our prototype offers a helper class 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 and EDSL-specific parts of a program, we can perform staging using a simple abstract interpretation (forward flow) data-flow analysis approach [12]. A trivial parsing 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.

[2] This could be extended to fields but in our current implementation we limit ourselves to methods.

The idea of implicit staging extends beyond mere syntactic extraction. Instead, we attempt to statically interpret tokens as if they were deeply embedded and thus retrieve a static, anticipated shape of compound EDSL expressions. For the sake of simple API design and implementation complexity, our prototype is still very limited in that it only extracts expressions on a mostly local scale.

Intermediate Representation. The staged IR in terms of section 3 is simply a list of the contained EDSL expressions' ASTs. In the following we will discuss their representation. Every instance of `Expression` 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 this 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 numbering analysis are currently performed as part of the same data-flow analysis. The latter is currently very simple and only tracks storing and loading of local variables and some stack operations such as duplication, the former follows a similar pattern as is found in bytecode verification in the JVM. In fact, our data-flow analysis is an extension of such type-analysis component already present in Javassist[3]. Hence, for the sake of brevity, we will omit value numbering and type analysis for the rest of our 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 currently only one type 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 as a sequence of expressions. A similar expression type is `ConversionExpression` (*cnv*) that wraps a convertee expression. It integrates with the parameters of invocation expressions to bookkeep for potential conversions (casting, boxing, unboxing).

The following expressions constitute the terminal leaves of a resulting expression AST and are considered *parameter expressions* (`ParameterExpression`) as they stand for the parameters to domain-specific computation:

- `LocalAccessExpression` (*lac*) holds the stored local variable that is accessed and its potential indices in the local variable array.
- `StringConstantExpression` (*str*) and `NullExpression` (*nul*) stand for (and hold) constant values. This can be easily extended to other constants.
- `StandaloneExpression` (*sta*) wraps an expression that is to be treated as standalone.
- `UnknownExpression` ($\top$) stands for a value resulting from unknown, usually EDSL-external, computation.

[3] `javassist.bytecode.analysis.Analyzer`

Figure 4 shows the AST resulting from staging the expression `add(add(a, b), mul(c, sca(5, 3.0)))` (cf. listing 1). The reason for the two rightmost leaf expressions being $\top$ is that we currently do not handle numeric constants. We would get the same result if the two values came from non-EDSL method calls.

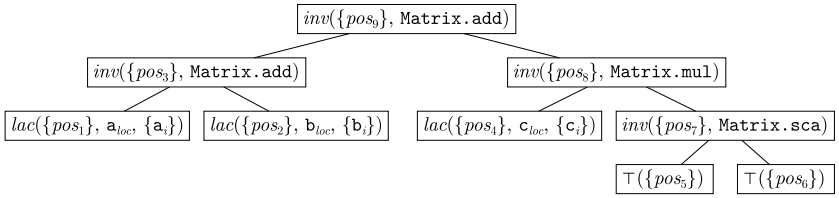

Fig. 4. AST for `add(add(a, b), mul(c, sca(5, 3.0)))`

The `StandaloneExpression` type requires additional explanation. Consider the following expression: `add(a, d = mul(b, c))`. The `mul(b, c)` part is required to be considered standalone, as it could be shared with EDSL-external code. For the current discussion it can be considered equivalent to $\top$.

Abstract Interpretation (Transferring States). The subject states of our data-flow analysis are JVM stack frames containing the contents of the operand stack (i.e. expressions) as well as local variables. Our abstract interpretation models the effect of bytecode instructions using a *transfer function* (as is common in data-flow analysis) which transfers the state before interpreting an instruction to that after it. We informally describe this function as follows:

(i) If we encounter an invocation instruction for a method m, we first check whether m is a token of the EDSL using `isToken`. If so, we pop the number of parameters for this method from the stack, create a new invocation expression (*inv*) with these parameters and the instruction's position, and add this expression to a (global) list of extracted expressions. If the method returns a value, we also push the expression onto the stack. If m is not a token but one of the auto-boxing and unboxing methods, we pop an expression from the stack, wrap it into a conversion expression (*cnv*) with the instruction's position, and push it onto the stack. Checked cast instructions are handled in a similar fashion.

(ii) If we encounter a store instruction, we pop an expression from the stack and create a stored local variable (*loc*) with the instruction's position and place it at the desired index into the local variable array of the stack frame. We also mark the positions of the popped expression as standalone.

(iii) In case of a load instruction, we retrieve the associated stored local variable (*loc*), then create a new local access expression (*lac*) containing this (with its index) as well as the instruction's position, and push it onto the stack.

(iv) Handling the various constant instructions is trivial.

(v) Any other 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.

Abstract Interpretation (Merging States). When our abstract interpretation encounters a branching instruction, it needs to explore all the branches. When these branches join back together (e.g. after an `if` statement), the states of these branches are merged. For our stack frames we do this by pointwise merging of the contained expressions and local variables using a *merge function* (as is common in data-flow analysis). It can be briefly summarized as follows:

(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 constant expressions yields the same constant if they share the same value and are of same type, otherwise $\top$ with merged positions.
(iv) 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.
(v) Merging different types of expressions and merging any expression with $\top$ always yields $\top$ with merged positions.
(vi) Merging expressions of same type 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 by simply overwriting. Merging stack frames of different size should not happen and when detected produces an error.

Post-processing. 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 post-processing step the global list of expressions is then purged of true subexpressions, and expressions whose positions have been marked standalone are turned into standalone expressions.

Having introduced this, we can now illustrate the effects of the abstract interpretation. Consider the expression `mul(a, x > 0 ? add(b, c) : mul(b, c))`. Java's ternary operator is not reconstructed by our analysis. Instead, the analysis deals with this situation by merging the stack frames at the end of the two branches. For the case that `x` is greater than zero we have this expression at the top of the abstract operand stack:

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

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

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

Our 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:

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

This means that our analysis would yield 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.

4.3 Processing: Expression Translation

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

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 runtime 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 to the given expression site via an instance method on it, called `setCode`.

Translation to Live Objects. Since compiling from our intermediate representation AST format to source code can be a daunting task, we also offer a high-level alternative: Translation to live objects. To this end, we provide the abstract class `ExpressionToCallableCompiler` which implements the low-level `ExpressionCompiler` interface.

EDSL developers implement the `compileToCallable` method which returns an instance of `Callable<T>`. Eventually, our framework implementation will replace the original EDSL expression (site) with a call to the `call` method of the returned `Callable<T>` instance. Our `Callable<T>` 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 arguments passed to the staged EDSL expression.

Environment elements can be accessed through instances of the `Variable<T>` class, which trivially implements the `Callable<T>` 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. Glue code generated by our 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.

`ExpressionToCallableCompiler` implements the low-level `compile` method in three steps. First, `compileToCallable` is called. Then, an accessor class is created and the return value from the first step is written to a static field of this accessor class (using runtime reflection). Finally, glue code is generated which creates an `Environment` instance filled with the expression's arguments and calls the `Callable<T>` 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 our `Callable<T>` interface with semantics close to the shallow embedded methods of similar names introduced in section 2.1. 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.

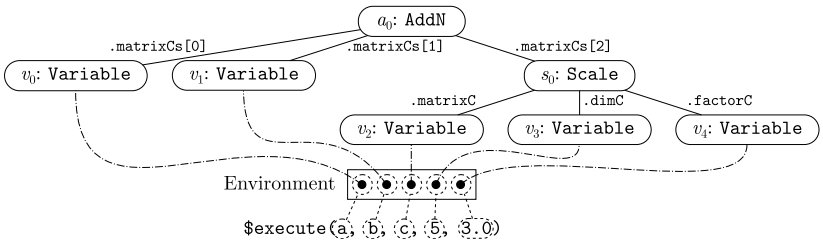

Fig. 5. Optimized `Callable<T>` tree for `add(add(a, b), mul(c, sca(5, 3.0)))`

Take again the expression `add(add(a, b), mul(c, sca(5, 3.0)))` (cf. figure 4). A high-level expression compiler can be defined by the EDSL developer to optimize and translate this expression to the tree presented in figure 5. 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 5 shows this glue code, assuming the generated accessor class is called `$CallableAccessor`.

Listing 5. Glue code (shortened) in `$execute` method

```
1 static Matrix $execute(Matrix u, Matrix v, Matrix w, int x, double y) {
2     Object[] values =
3         new Object[] { u, v, w, Integer.valueOf(x), Double.valueOf(y) };
4     Environment environment = $CallableAccessor.createEnvironment(values);
5     return (Matrix) $CallableAccessor.callable.call(environment);
6 }
```

Glue code generation happens behind the scenes and can safely be ignored by EDSL developers. All of this allows the definition of an expression's semantics via an essentially static computational object entry point. What this actually looks like internally is in the hands of the EDSL developer.

4.4 Unstaging: Relinking Expression Sites

Having translated all expressions and provided Java method bodies (e.g. as in listing 5) for the expression sites, our framework implementation then needs to establish the appropriate links in the user program.

For every expression site, a (uniquely named) static method (like `$execute` in figure 4 and listing 5) with the expression site's (flattened) type signature is added to the surrounding class, and its body is set to the provided source code. Javassist comes with an inbuilt, custom compiler that makes this possible. Subsequently, every instruction associated with non-parameter subexpressions of an EDSL expression site are removed from the bytecode. Finally, a call to the associated method is inserted at the expression site's position.

For the sake of brevity, we 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 (as imposed by the JVM).

5 Evaluation

The evaluation of our prototype is split into two parts. We first discuss the limitations of our current IR and data-flow analysis and give hints at potential extensions. In the second part we present simple EDSLs and experiments on the runtime performance impact of implicit staging.

5.1 IR and Staging Limitations

We kept our prototype simple both for illustration purposes as well as for the simplification of implementation and API design. This means that the vision outlined in section 3.2 has by no means been fully achieved. The most significant limitations currently stem from the very simple data-flow analysis used for the extraction of EDSL programs. Namely, only domain-specific code originally occurring as compound expressions is extracted. Furthermore, these expressions are treated in a very isolated fashion.

While not discussed in section 4, we have actually experimented with allowing the inspection of variable accesses (*lac*) to offer some level of non-local view. Take for instance the code snippet in listing 6. During expression translation our prototype allows the inspection of the accesses to `t` in order to optimize all multiplications referring to it, e.g. to perform appropriate scalar scaling instead of matrix multiplication.

Listing 6. Non-local, interleaved EDSL code example

```
1  t = sca(5, 3.0);
2  u = mul(a, t);
3  if (/* Omission */) { v = mul(u, t); } else { v = add(u, t); }
```

However, we found it challenging to devise an easy-to-use API on the current level of processing single expressions 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 external uses. We believe it is necessary to expose more details (for instance as graphs of shared expression usage) to EDSL developers to handle these non-local aspects.

Note that with deep embedding the aforementioned case is not an issue. Furthermore, at the end of the given code snippet, `v` would be a dynamically staged program that depends on the actual flow of control. However, in a static setting we cannot predict what path will be taken. One way around this would be to pre-optimize expressions for every possible shape they may take. However, in the general case this is likely to cause intractable code explosion. Another approach currently under consideration is to implicitly switch to runtime staging for these dynamic, interleaved code situations using site-specific type conversions.

Listing 7 provides another example trivially solved by staging at runtime. However, in cases like this, a more powerful data-flow analysis could actually statically determine that this code can be unrolled to stand for `t = add(add(add(a, b), b), b)`. Doing so, we enter the realm of partial evaluation to improve the prediction of the concrete shape of EDSL code, while still allowing its processing to be guided by the EDSL developer.

Listing 7. Constant EDSL expression generation example

```
1  t = a;
2  for (i = 0; i < 3; i++) { t = add(t, b); }
```

To some extent this notion could be extended to staging occurring over several method calls. However, in Java and many other OOP languages it is not always possible to statically determine the exact target of a method call. Such highly dynamic cases are best left to deep embedding, not precluding the aid of implicit staging (cf. 3.2) within method bodies.

5.2 Experiment A: Matrix EDSL

For evaluation purposes we implemented three versions of the matrix EDSL appearing throughout this paper, using shallow embedding, implicit staging per our prototype (with compilation to `Callable<T>`) imitating the look-and-feel of the shallowly embedded version, and deep embedding. The latter two perform optimizations as indicated in section 4.3, i.e. fusing binary additions and turning multiplications with scaling matrices into scaling operations with further fusion when applicable. We made the utmost effort to keep these implementations as comparable as possible to each other.

We set up an experiment to assess not only how effective our optimizations actually are, but also to get a rough idea of how often they might actually be applicable. To this end, we considered randomly generated matrix operation expressions up to a depth of 5 for which we counted `Matrix` variables and `sca` expressions as leaves. For each depth we have 30 such expressions, once occurring in a warm-up loop and once in a loop for which execution time is measured. We generated random 8×8 matrices and scalar values of type `double` to serve as parameters for these expressions and assign them to local variables as we were

not interested in the literal generation time. This generated benchmark was also adapted for the deeply embedded language version. Note that all randomness was only part of the benchmark code generation.

Initially we ran the benchmark code ten times for each version with 100000 loop iterations for warm-up and measurement, each on a 3 GHz Intel Core i7 machine with 8 GB of RAM with JRE 7[4]. Due to measurement fluctuations, we opted to increase the number of loop iterations to 10000000 and reran the benchmark code, this time only three times per EDSL implementation version due to the increased running time per benchmark. Apart from the much lower fluctuations between the results, on average these new measurements match very closely with the results of the earlier, shorter experiment. We will mainly discuss the results of the 10000000 iterations experiment here.

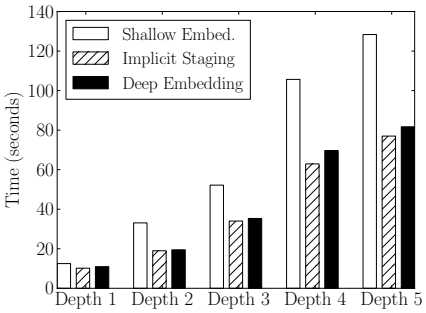

Fig. 6. Random matrix expr. results

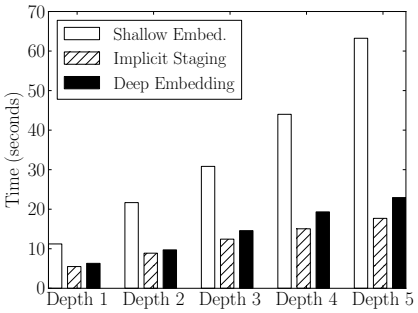

Fig. 7. Biased matrix expr. results

Figure 6 shows the results of our experiment with random expressions. Due to space concerns, we summarize the results by averaging over all 30 expression execution times per expression depth. For expressions at depth 1, implicit staging 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 probability of optimization opportunities for expressions of depth 1 and the added overhead of boxing and `Callable<T>` 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 depth 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

[4] Java(TM) SE Runtime Environment (build 1.7.0_21-b12)
Java HotSpot(TM) 64-Bit Server VM (build 23.21-b01, mixed mode)

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%.

We also generated the same type of benchmark with a bias towards optimizable expressions. Figure 7 shows the results of our experiment with this benchmark code. It is no surprise that shallow embedding did not fare well in this 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 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%.

We also wanted to explore worst-case performance for our implementations. To this end, we chose the expression `mul(mul(add(sca(5, 2.0), sca(5, 2.0)) , add(sca(5, 2.0), sca(5, 2.0))), add(sca(5, 2.0), sca(5, 2.0)))`. It lends itself as a worst-case specimen, since no optimizations (though possible) were implemented for adding scaling matrices.

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 overhead caused by expression tree (or `Callable<T>` tree) evaluation is significant. In order to further test this case, we implemented an expression compiler that generates Java code identical to the original expression instead of a `Callable<T>`. This implementation was only 1.1% slower (for our worst-case expression) than shallow embedding. Hence it seems advisable to move away from the simpler `Callable<T>` compilation for final versions of an implicitly staged EDSL implementation. It may be worthwhile to investigate how we can automate this code generation from a compiled `Callable<T>` instance.

Implicit Staging Overhead. Unfortunately, our implicit staging implementation incurs substantial overhead at class-loading time. In our experiments, expressions at a certain depth were collected in their own class whose static initializers we used for measuring the time class loading took. As a basis for comparison, we took our earliest ten runs experiment and therein the class loaded last, i.e. the one for depth 5, as we can assume the runtime environment to be warmed up at this point. For this case, we measured that our implementation slowed down the class-loading process by 529.4 ms. That was 138.3 times slower than without using implicit staging.

It is important to note that this overhead is incurred only once per class and only if this class actually contains code potentially referring to EDSL expressions. In a large code base, this overhead might indeed become problematic but to a

certain degree it is an inevitable side effect of our approach. We will still attempt to further optimize at least the fixed parts of our framework implementation (data-flow analysis, expression site relinking, etc.).

Implementation Complexity. Although only of limited reliability, we use lines of code as a metric to estimate the implementation complexity for each of the approaches. The shallow embedding implementation was accomplished in about 100 lines of code, the implicit staging implementation took up about 360 lines of code, and the deep embedding implementation took up about 300 lines of code. It appears that implicit staging does not incur much more implementation complexity than deep embedding. Of course, this can mainly be attributed to the fact that we tried (and managed) to stay as similar as possible with our implementations.

5.3 Experiment B: Chained Filtering and Mapping EDSL

Our previous example language used static method call nesting for its syntax. Of course, it is also possible to implicitly stage EDSLs which use method chaining.

We implemented an abstract data type `FunSequence<T>` which allows for (immutable) list operations as are common in functional programming, i.e. filtering (`filter` with `Predicate<T>`) and mapping (`map` with `Function<T, R>`). Two concrete classes implement this data type, the list-backed `FunList<T>`, and the array-backed `FunArray<T>`. Again, we started out with a naive shallow embedding.

Consider the code snippet presented in listing 8 and assume that the functions and predicates used as arguments are defined outside. Every mapping or filtering creates a new list and makes this compound expression rather memory demanding and slow.

Listing 8. `FunSequence` usage example

```
1 FunSequence<Integer> res = inputSeq.map(sqrt).map(square).map(increment)
2                                    .filter(even).filter(greaterThanZero)
3                                    .map(invert).map(invert).map(increment)
4                                    .filter(divisibleByFour);
```

Running this code (warmed up) in a loop with 10 iterations, `inputSeq` initialized to 6000000 random elements with values between -100 and 100, took on average across 10 runs about 14613.4 ms ($\sigma = 1178.2$ ms) with `FunArray<T>` and about 31221 ms ($\sigma = 2192.2$ ms) with `FunList<T>`[5].

With the help of implicit staging, we implemented optimizations for this EDSL, notably the fusion of filtering and mapping operations into a single loop. Its implementation was encapsulated in an non-type-safe method and should thus not be exposed publicly. Though artificial this example may seem, it showcases that implicit staging can be used to expose optimized but unsafe functionality in a type-safe fashion.

[5] In the latter case it was necessary to increase the maximum heap space size.

Running the aforementioned benchmark with implicit staging took 8786.3 ms on average (σ = 180.2 ms) with `FunArray<T>` and about 9790.8 ms (σ = 934 ms) with `FunList<T>`. The former was faster by 66.3% compared with pure shallow embedding, the latter by 218.9%.

5.4 Experiment C: Safe Arithmetic EDSL

Our last example is a language for performing integer arithmetic with overflow detection. It is based on one of the methods described in "The CERT Oracle Secure Coding Standard for Java" [13], which involves conversion to values of `BigInteger` type.

Listing 9. Safe addition

```
1 public static final int add(int left, int right) {
2     return intRangeCheck(
3         BigInteger.valueOf(left).add(BigInteger.valueOf(right))).intValue();
4 }
```

Listing 9 shows the implementation for addition, where `intRangeCheck` will throw a runtime exception in case of detected overflow. Other operations are implemented in a similar fashion. Warmed up, executing the loop shown in listing 10 takes across 10 runs on average 2490.3 ms (σ = 21.8 ms).

Listing 10. Safe arithmetic EDSL benchmark

```
1 for (int i = 0; i < 10000000; i++) {
2     int j = i % 100;
3     res = mul(mul(add(a, j), add(a, j)), add(a, b));
4 }
```

We also implemented an implicitly staged version, which does away with the redundant conversion of intermediate values. All parameters are converted to `BigInteger` and only at the end, before converting back to `int`, overflow checking is performed. This simple optimization reduces the running time of the benchmark (with implicit staging) on average to 2372.9 ms (σ = 35.3 ms). This may not seem much, but consider that our prototypical EDSL implementation causes additional boxing, unboxing, and `Callable<T>` calling overhead. With an additional common subexpression elimination optimization, the benchmark running time is reduced to 2066.7 ms (σ = 17.5 ms). Of course, this effect is more drastic the more common subexpressions occur.

Note that implicit staging here effectively changes the semantics. Namely, it is fine for intermediate results to exceed the `int` extrema as long as the end result is within them. The expression `sub(add(Integer.MAX_VALUE, 5), 5)` will throw an exception in the shallowly embedded implementation, whereas our implicitly staged implementation would return `Integer.MAX_VALUE`. This is intentional, since we want to consider compound expressions of the EDSL as closed entities. This may seem as an unfair advantage against shallow embedding but the fact that implicit staging allows us to do so is in the first place is exactly what we want to highlight here.

6 Related Work

DSLs and little ad-hoc languages have been advocated for use in domain-specific tasks at least since Bentley's article on "Little Languages" [14]. Syntactic extension allows general-purpose languages to embed such languages. There exist general-purpose languages such as Converge [15] or Lisp with powerful compile-time metaprogramming features that allow this in an integrated fashion.

Converge is a dynamically typed language with a strong focus on allowing custom and rich syntax extensions in combination with splicing annotations for DSL development. Domain-specific code (with custom syntax) is explicitly marked as DSL blocks or shorter DSL snippets. In most Lisp dialects syntactic extension are somewhat limited in the framework of S-expressions, yet powerful macro systems effectively allow for a great deal of linguistic customizations and domain specialization. Template Haskell [16] allows compile-time metaprogramming in a type-safe fashion with explicit notation for compile-time expansion.

It is important to note that these compile-time facilities rely on the availability of source code user programs and consider syntactic entities. Our load-time implicit staging approach for Java is based on data-flow analysis instead, which manages to recover EDSL code snippets while hiding non-EDSL code.

As shown in section 5.2, there is a substantial overhead associated with our current prototype and undeniably with our load-time approach in general. This issue is not shared by traditional compile-time metaprogramming approaches. However, for these approaches it is much harder or impossible to avoid deployability issues and to enable cross-stage persistence in a way that allows generated code to access data available during the staging phase. At load time, the latter becomes a trivial issue. Furthermore, while our current prototype is still limited in scope and might in fact somewhat resemble a load-time hygienic macro system, we believe its abstract EDSL token interpretation approach is more amenable to further extensions, as indicated in section 3.2 and 5.1.

Even without syntactic extension capabilities or macro functionality, embedded DSLs have been shown to be feasible using deep or shallow embedding, or combinations thereof [7]. Hudak [1] and Elliot et al.[17] have shown that Haskell is well suited for this. Yet, even languages with stronger restrictions on syntax and more verbosity, such as Java, have been used to implement EDSLs [2,3,5].

To overcome runtime performance issues, Hudak [1] has proposed partial evaluation. Czarnecki et al. [18] have presented an effective approach using staged interpreters which requires a host language with *multi-stage programming* (MSP) support [9,11]. Bagge et al. [19] have used a source-to-source transformation solution for the C++ language, enabling optimizations via rewrite rules. Guyer et al. [20] have introduced a compiler architecture for the C language which enables domain-specific optimizations not on the syntactic level but on the data-flow level. In fact, Guyer et al. [20] claim not to target the optimization of DSLs, but that of the domain-specific aspects of software libraries. These optimizations are communicated to the compiler by analysis and action annotations (written in their own dedicated language). It is similar to implicit staging in its (external)

specification of domain-specific procedures as well as its detachment from the mere source code syntax level.

The aforementioned ideas either rely on non-mainstream host languages or compiler extensions. Rompf et al. [10] have introduced a method called Lightweight Modular Staging (LMS) which is a purely library based approach. LMS brings MSP support to the Scala [21] language as a library, where lifted `Rep[T]` data-types stand for staged code. Using Scala's traits, it is easy to extend this library and implement EDSLs with it.

When optimizing EDSL programs, it is similar to the deep embedding approach but more elegantly hides its nature by employing Scala's type inference, trait composition, and implicit conversion features. LMS has been used to implement several EDSLs with the Delite [22,23] back end, such as OptiML [24] and OptiCVX [25], with great results. However, while the usage of EDSLs implemented using LMS is mostly seamless, the unstaging (i.e. the code generation, compilation, and loading) of staged code is triggered explicitly.

JIT macros as described by Rompf et al. in Project Lancet [26], a very ambitious and promising JVM implementation, resemble parts of our load-time staging approach. Combined with LMS, JIT macros are described as allowing domain-specific optimizations at JIT-compile time. However, their expansion or handling necessitates the localized, explicit triggering of JIT compilation in user programs, a feature of Lancet.

7 Conclusion

To address the issue of DSL (expression) embedding, we proposed implicit staging, an impure approach to language embedding. We have further concretely implemented and introduced an instance of implicit staging for the Java language using load-time reflection. Our prototype implementation has shown to be an effective tool for implementing EDSL expression semantics in a customized fashion, while letting EDSLs expose a shallow interface. By moving the process of staging to load time, we gain an advantage in reducing overhead compared to deep embedding.

Our prototype offers an improvement over pure shallow embedding and we believe it can serve as an alternative to deep embedding in many cases. Namely, when true, dynamic runtime staging of EDSL code is not mandatory. However, it remains to be seen how our approach scales beyond the small examples we evaluated. We intend to investigate this in the future.

Despite its current limitations we believe our framework can be used as stepping stone to more elaborate implicit staging (at load time) systems. It is our future work to explore designs, implementations, and use cases for exploiting static and dynamic contextual information both within user programs as well as the runtime system.

References

1. Hudak, P.: 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) 134–142
2. Freeman, S., Pryce, N.: 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, New York, NY, USA, ACM (2006) 855–865
3. `https://code.google.com/p/guava-libraries/` (retrieved: 2013-12-2)
4. Giarrusso, P.G., Ostermann, K., Eichberg, M., Mitschke, R., Rendel, T., Kästner, C.: Reify your collection queries for modularity and speed! In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development. AOSD '13, New York, NY, USA, ACM (2013) 1–12
5. `http://www.jooq.org/` (retrieved: 2013-12-2)
6. Chiba, S.: Load-time structural reflection in java. In Bertino, E., ed.: ECOOP 2000 Object-Oriented Programming. Volume 1850 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2000) 313–336
7. Svenningsson, J., Axelsson, E.: Combining deep and shallow embedding for EDSL. In Loidl, H.W., Pea, R., eds.: Trends in Functional Programming. Volume 7829 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 21–36
8. Gosling, J.: Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). In: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations. IR '95, New York, NY, USA, ACM (1995) 111–118
9. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '10, New York, NY, USA, ACM (2010) 400–411
10. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. Commun. ACM **55**(6) (June 2012) 121–130
11. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1-2) (October 2000) 211–242
12. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '73, New York, NY, USA, ACM (1973) 194–206
13. `https://www.securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow` (retrieved: 2013-12-2)
14. Bentley, J.L.: Programming pearls: Little languages. **29**(8) (August 1986) 711–721
15. Tratt, L.: Domain specific language implementation via compile-time meta-programming. TOPLAS **30**(6) (2008) 1–40
16. Sheard, T., Jones, S.P.: Template meta-programming for haskell. SIGPLAN Not. **37**(12) (December 2002) 60–75
17. Elliott, C., Finne, S., De Moor, O.: Compiling embedded languages. J. Funct. Program. **13**(3) (May 2003) 455–481
18. Czarnecki, K., O'Donnell, J., Striegnitz, J., Taha, W.: DSL implementation in MetaOCaml, Template Haskell, and C++. In Lengauer, C., Batory, D., Consel, C., Odersky, M., eds.: Domain-Specific Program Generation. Volume 3016 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 51–72

19. Bagge, O.S., Kalleberg, K.T., Haveraaen, M., Visser, E.: Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In Binkley, D., Tonella, P., eds.: Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003), Amsterdam, The Netherlands, IEEE Computer Society Press (September 2003) 65–75
20. Guyer, S., Lin, C.: Broadway: A compiler for exploiting the domain-specific semantics of software libraries. Proceedings of the IEEE **93**(2) (Feb 2005) 342–357
21. `http://www.scala-lang.org/` (retrieved: 2013-12-2)
22. Brown, K.J., Sujeeth, A.K., Lee, H.J., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques. PACT '11, Washington, DC, USA, IEEE Computer Society (2011) 89–100
23. Chafi, H., Sujeeth, A.K., Brown, K.J., Lee, H., Atreya, A.R., Olukotun, K.: A domain-specific approach to heterogeneous parallelism. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11, New York, NY, USA, ACM (2011) 35–46
24. Sujeeth, A.K., Lee, H., Brown, K.J., Chafi, H., Wu, M., Atreya, A.R., Olukotun, K., Rompf, T., Odersky, M.: Optiml: an implicitly parallel domainspecific language for machine learning. In: in Proceedings of the 28th International Conference on Machine Learning, ser. ICML. (2011)
25. `http://stanford-ppl.github.io/Delite/opticvx/index.html` (retrieved: 2013-12-2)
26. Rompf, T., Sujeethy, A.K., Browny, K.J., Lee, H., Chazy, H., Olukotuny, K., Odersky, M.: Project lancet: Surgical precision JIT compilers. Technical report (2013)