A Survey of Metaprogramming Languages

YANNIS LILIS, ICS-FORTH
ANTHONY SAVIDIS, University of Crete and ICS-FORTH

Metaprogramming is the process of writing computer programs that treat programs as data, enabling them to analyze or transform existing programs or generate new ones. While the concept of metaprogramming has existed for several decades, activities focusing on metaprogramming have been increasing rapidly over the past few years, with most languages offering some metaprogramming support and the amount of metacode being developed growing exponentially. In this article, we introduce a taxonomy of metaprogramming languages and present a survey of metaprogramming languages and systems based on the taxonomy. Our classification is based on the metaprogramming model adopted by the language, the phase of the metaprogram evaluation, the metaprogram source location, and the relation between the metalanguage and the object language.

CCS Concepts: • **Software and its engineering → Multiparadigm languages**; **Language features**; *Patterns*; *Frameworks*;

Additional Key Words and Phrases: Metaprogramming, macro systems, reflection, meta-object protocols, aspect-oriented programming, generative programming, multistage languages

ACM Reference format:
Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (October 2019), 39 pages.
https://doi.org/10.1145/3354584

1 INTRODUCTION

Metaprogramming refers to the writing of computer programs, called *metaprograms*, that can treat programs as data, thus enabling them to generate new programs or modify existing ones. The language in which the metaprogram is written is called the metalanguage, while the language in which the generated or transformed program is written is called the object language. If the object language and the metalanguage are the same, it is a case of *homogeneous* metaprogramming, while if they are different it is a case of *heterogeneous* metaprogramming.

Manipulating source code to apply transformations requires handling of the code itself in a convenient abstract syntactic representation, typically that of an Abstract Syntax Tree (AST), a notion originating from the s-expressions of Lisp [Steele 1990]. Manually creating and composing ASTs can be cumbersome and hinder code readability [Weise and Crew 1993]. Instead, quasi-quotations [Bawden 1999] allow using concrete syntax while converting the enclosed code into AST values.

Authors' addresses: Y. Lilis, Institute of Computer Science, FORTH, N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece; email: lilis@ics.forth.gr; A. Savidis, Computer Science Department, University of Crete, Voutes Campus, GR-700 13 Heraklion, Crete, Greece and Institute of Computer Science, FORTH, N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece; email: as@ics.forth.gr.

0360-0300/2019/10-ART113 $15.00
https://doi.org/10.1145/3354584

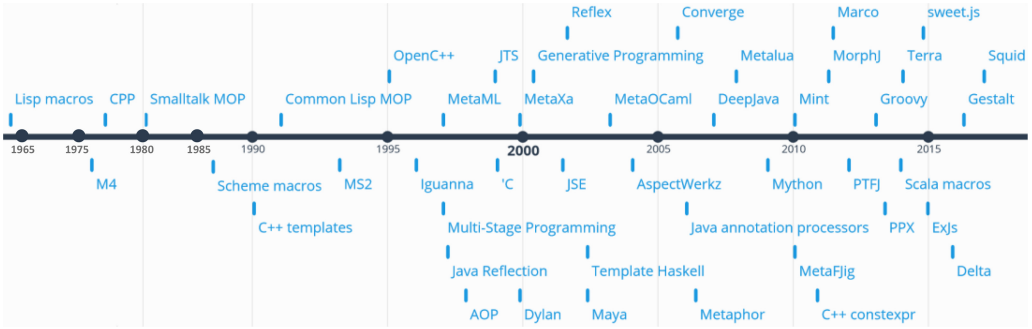

Fig. 1. Timeline of metaprogramming languages and systems; the list is not exhaustive.

Metaprogramming offers various benefits [Sheard 2001], including performance, reasoning about object programs, and source code reuse. Metaprograms can offer improved performance by generating efficient specialized programs based on specifications instead of using generic but inefficient programs. Partial evaluation can also minimize runtime overhead by identifying computations operating on static inputs and executing them during compilation. Reasoning about object programs is also important as it allows analyzing and discovering object-program characteristics that enable applying further optimizations as well as inspecting and validating the behavior of the object program (e.g., flow analyzers and type checkers). Finally, and most importantly, metaprogramming can be used for code reuse at both a microscopic and macroscopic scale. Languages have traditionally supported code reuse via functions, generics, polymorphism, classes, and interfaces. However, there are recurring code patterns that cannot be abstracted and reused with the above approaches. Since metaprogramming transforms or generates code, operating on code segments, it is possible to capture and abstract the recurring code using some structured representation and deliver it as a directly reusable unit.

Metaprogramming is not a new concept; in fact, it has a history spanning several decades, dating back to early Lisp macros. As shown in Figure 1, the field of metaprogramming has been receiving increasing attention over the years and especially in the last couple of decades where research focusing on metaprogramming has grown rapidly with new metaprogramming paradigms emerging: aspect-oriented programming [Kiczales et al. 1997], multistage programming [Taha and Sheard 1997], and generative programming [Czarnecki and Eisenecker 2000]. In the same period, various metaprogramming languages and systems have emerged, while even mainstream languages such as C++ and Java have introduced or improved their metaprogramming support. C++11 introduced *constexpr* and extended template support, C++14 and C++17 improved them further, and more additions are planned for C++20. Java 6 added compile-time generative metaprogramming through annotation processors [Darcy 2006]. Other JVM-based languages such as Groovy and Scala offered metaprogramming support through AST transformations [Subramaniam 2013] and macros [Burmako 2013], respectively.

In this article, we introduce a taxonomy of metaprogramming languages and present a survey of metaprogramming languages and systems based on the taxonomy. In particular, we classify languages by the metaprogramming model they adopt (Section 2), the phase of the metaprogram evaluation (Section 3), the metaprogram source location (Section 4), and the relation between the metalanguage and the object language (Section 5).

Our classification builds on previous metaprogramming taxonomies [Sheard 2001; Pasalic 2004; Damaševičius and Štuikys 2008], extending some aspects and introducing new categories. Some existing categories, such as the kind of metaprogram (generator vs. analyzer) or the openness of the

metalanguage (open vs. closed) of Sheard's and Pasalic's taxonomies are relevant but orthogonal to our discussion and are not further presented in our classification.

The most important addition not present in existing taxonomies is the model adopted by the language. The taxonomy of Damaševičius and Štuikys presents some structural concepts (e.g., separation of concerns) and process concepts (e.g., generation, reflection) that are closely related to some of our metaprogramming models. However, their taxonomy uses them as generic metaprogramming concepts, while in our classification they constitute established practices for supporting metaprogramming. Our classification is also comprehensive, covering various models: macro systems, reflection, metaobject protocols, aspect-oriented programming, generative programming, and multistage programming. Another addition is the introduction of a classification category based on the metaprogram source location, as a metaprogram may be embedded within the source code of its subject program or it may be defined and applied externally with respect to the program it operates on.

We also extend aspects of the Sheard and Pasalic taxonomies. We extend the static (i.e., compile time) versus runtime classification of the metaprogram evaluation phase to also include metaprograms executed during a preprocessing phase. In our categorization, we distinguish three evaluation phases for metaprograms: (1) before compilation, as a preprocessing step; (2) during compilation; and (3) during execution. Finally, our taxonomy extends the classification between homogeneous and heterogeneous systems, present in both taxonomies, to consider all possible relations between object language and metalanguage: (1) the metalanguage being indistinguishable from the object language (identical or subset), (2) the metalanguage extending the object language, and (3) the metalanguage being a different language. The proposed taxonomy is depicted in Figure 2, while a classification of various metaprogramming languages and systems based on the taxonomy is summarized in Table 8 at the end of the article.

2 METAPROGRAMMING MODELS

The model adopted by a metaprogramming language is of significant importance as it defines the prevalent method toward practicing metaprogramming in the language. Thus, we consider it to be a fundamental part of our taxonomy, with categories constituting proven methods for metaprogramming across various existing language and system implementations. In particular, we distinguish between the following categories: (1) macro systems, (2) reflection, (3) metaobject protocols, (4) aspect-oriented programming, (5) generative programming, and (6) multistage programming. Each of the models may target different engineering tasks (e.g., code generation, code specialization, introspection, change of behavior, separation of concerns) or involve different development practices, but there are also close relations and overlaps between some of them. We elaborate on each model category and present representative languages and systems related to every metaprogramming model. Since many languages have characteristics fitting more than one model category, in our classification we try to list each language under the most fitting model, while discussing extra cross-model connections where necessary.

2.1 Macro Systems

2.1.1 Model Details. Macro systems operate on a source file by specifying input sequences that should be mapped to output sequences according to a user-defined procedure. This mapping process, called *macro expansion*, is applied iteratively until no more macro invocations exist in the source file, at which point the resulting macro-free program is sent for translation (Figure 3). Macro systems fall into two categories, *lexical macros*, which are language agnostic and operate on a lexical level (i.e., on a sequence of tokens), and *syntactic macros,* which are aware of the language syntax and semantics, typically constituting a built-in language mechanism. Macro systems may

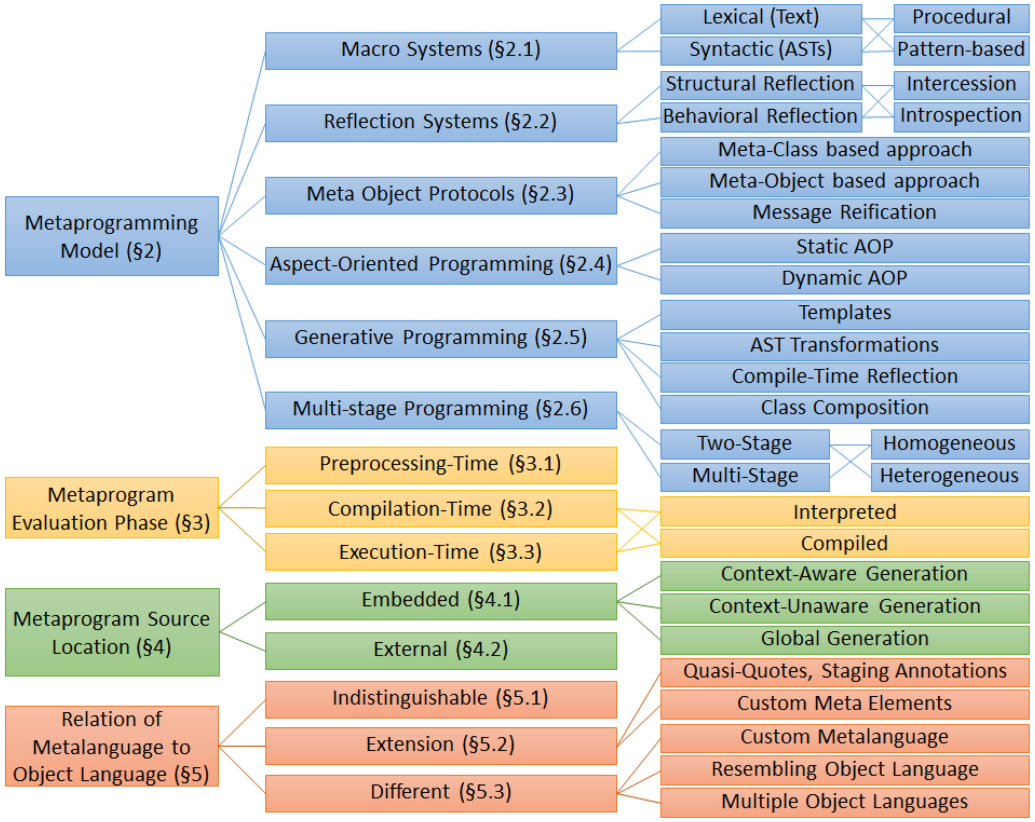

Fig. 2. The proposed taxonomy: the four dimensions with their categories (linked to sections and subsections of the article) and subcategories where applicable.

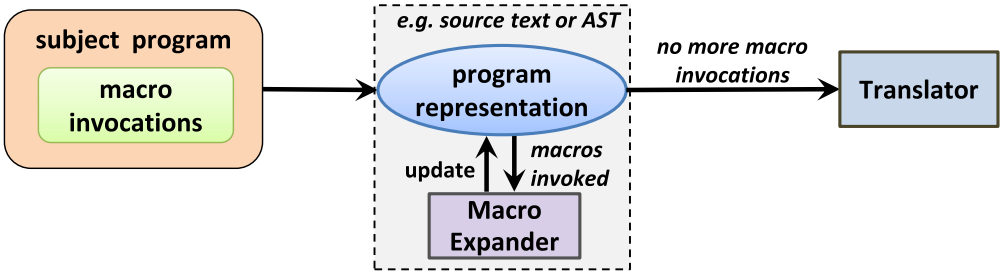

Fig. 3. Macro systems: macro invocations update the program representation in an iterative macro expansion process, after which the macro-free final program is sent for translation.

be *procedural* (i.e., supporting algorithmic computations for generating their output), or they may be *pattern based*, relying only on pattern matching and substitution. Finally, an important property of macro systems is *hygienic* expansion, that is, guaranteeing that no unintended name conflicts are introduced by the macro expansion, a problem referred to as *variable capture*.

2.1.2 Representative Languages and Systems. Lexical macros typically operate externally with respect to the language translator and are thus also called external preprocessors, with the

C preprocessor (CPP) [Kernighan and Ritchie 1988] and *M4* [Turner 1994] being representative examples. CPP supports object-like and function-like macros as well as token concatenation and token stringification, while M4 supports a freeform syntax and a high degree of macro expansion and supports macros that can generate other macros. Even though such systems are widely used and support some basic metaprogramming, they tend to be insufficient for full-scale metaprogramming. The main reason is that treating code as text disables syntactic-level modifications or inspecting the internals of code supplied as an argument. Also, such macros may cause unintended side effects or name clashes and may introduce difficult-to-solve bugs.

The first language to offer a syntax-based macro system that utilizes the full language itself for the transformation logic is Lisp. The main difference between Lisp and other languages is that in Lisp, the textual program representation is simply a human-readable description of the internal data structures. Thus, any operation Lisp can perform to data structures, Lisp macros can perform on code. A macro is defined using the *defmacro* keyword followed by the name, arguments, and code replacement that may include the backquote, unquote, and splicing operators to allow representing code structures with arguments injected in them both in evaluated and unevaluated forms. However, Lisp macros are vulnerable to variable capture and require *gensym* calls for conflict resolution. FEXPRs are functions used in early Lisp dialects whose arguments are passed in unevaluated form, operating similarly to macros. They were less straightforward and reliable than macros, and hindered static analysis as the compiler could not distinguish between them and ordinary functions, leading to their deprecation.

Scheme [Dybvig 2009] was the first language with a hygienic macro system, pioneered by Kohlbecker et al. [1986], who also proposed a declarative pattern-based language for macros that led to the *syntax-rules* macro. Bawden and Rees [1988] introduced syntactic closures as an alternative way of solving scoping problems at macro expansion time, Clinger and Rees [1991] offered support for lexically scoped macros, and Dybvig et al. [1993] introduced the more powerful *syntax-case* macro that allows side conditions via arbitrary functions and breaking hygiene for intentional variable capture. After the introduction of hygiene in Scheme, most other macro systems (apart from Lisp derivatives), generative programming systems, and multistage languages (see, respectively, Sections 2.5 and 2.6) offered hygienic code expansion, but it is also quite common to offer facilities for breaking hygiene. *Racket* [Flatt 2002], a Scheme descendant, enforced a clear separation between phases and their computations, introduced robust pattern specifications that allowed validating macro constraints and reporting errors at the proper level of abstraction [Culpepper and Felleisen 2010], and offered a macro API that exposes compile-time information to enable macro cooperation [Flatt et al. 2012].

MS^2 [Weise and Crew 1993] was the first programmable syntactic macro system for syntactically rich languages, such as C. In MS^2, the macro language is a minimal extension of C offering a template substitution mechanism based on Lisp's quasi-quotes and uses a type system to ensure that all generated code fragments are syntactically correct. Like Lisp, it offers no support for hygiene but requires programmer intervention to avoid variable capture errors. Further syntactic macro systems for languages with a rich syntax include <*bigwig*> [Brabrand and Schwartzbach 2002] and Camlp4 [Rauglaudre 2003]. <bigwig> guarantees type safety and termination and introduces the concept of metamorphisms enabling user-defined grammar for invocation syntax and the ability to simultaneously operate on multiple parse trees. However, it is not programmable, supporting only simple declarative concepts such as grammars and substitution. Camlp4 is an OCaml preprocessor and pretty-printer offering a quasi-quote syntax and allows manipulating and generating code based on object language ASTs. However, it lacks a high-level pattern language and requires knowledge of its internal data structures and APIs.

Dylan [Bachrach and Playford 1999] aimed at achieving the power and simplicity of Lisp-style macros in a syntactically rich language. Its macro system is based on a skeleton syntax tree (SST) approach and uses a set of rewrite rules: the program is initially parsed using a phrase grammar that understands tokens and balanced delimiters. Then the SST is traversed parsing built-in forms and expanding macros based on the rewrite rules. Dylan macros respect hygiene by automatically renaming macro variables on expansion to guarantee unique names. However, syntactically, Dylan is not more flexible than Lisp, being essentially a Lisp dialect with infix syntax, while its macro system is a different language from Dylan. *JSE* [Bachrach and Playford 2001] is a Java macro system following a similar approach to Dylan macros. It differs from Dylan as it exploits Java's compilation model to offer a full procedural macro system instead of one based only on rewrite rules. JSE can also package and reuse syntax expansions as any other Java code, while the elements of its pattern-matching engine are open to extensions.

Maya [Baker and Hsieh 2002b] and *Nemerle* [Skalski et al. 2004] offer macro systems for Java and C#-like languages. Maya extends Java with a macro system based on lexically scoped syntax transformers, called Mayans, that are expanded during parsing to manipulate the syntax through AST operations, overriding the translation of built-in productions and adding new productions. Nemerle is a C# language featuring a macro system also capable of supporting syntax extensions. Both systems support arbitrary computations, respect hygiene, and allow macros to be compiled separately from their invocations, clearly separating normal code and metacode.

More recent attempts to combine hygienic macros systems with traditional algebraic notation include *ZL* [Atkinson and Flatt 2011], *Honu* [Rafkind and Flatt 2012], *sweet.js* [Disney et al. 2014], and *ExJs* [Wakita et al. 2014]. ZL offers Scheme-style macros for a C-like language. To deal with C's syntax, ZL does not parse a program in a single pass but uses an iterative-deepening approach and interleaves parsing with macro expansions. Honu introduced a precedence-based parsing step called *enforestation* that turns a relatively flat sequence of terms into a Lisp-like AST. Enforestation is integrated in the macro expansion process and supports hygiene, macro-generating macros, and local macro bindings. Sweet.js, a macro system for JavaScript, adapts Honu's enforestation for JavaScript and introduces infix macros. Both Honu and sweet.js support term rewriting macros and procedural macros. ExJs is another syntactic macro system for JavaScript offering hygienic term rewriting macros. It does not implement a hygienic macro expander from scratch but defers macro expansion to a Scheme interpreter. The original JavaScript code with macros is translated to Scheme, then a Scheme macro expander handles expansion, and finally, the result is translated back to JavaScript free of macros.

Marco [Lee et al. 2012] was the first expressive and safe macro system that is language independent. It is based on the observation that the macro system need not know all the syntactic and semantic rules of the target language but need only enforce specific rules that can be checked by special oracles utilizing unmodified target-language translators. These oracles are deployed by translating specially crafted programs and then analyzing any error messages. Marco provides static types, conditionals, loops, and functions, making it Turing Complete, while supporting target language fragments as first-class values through a quasi-quote syntax. For safety, it uses macro-language types to check target-language syntax and uses dataflow analysis to check target-language naming discipline. However, supporting extra languages requires providing the appropriate language-specific oracles, and guaranteeing safety requires the target language to produce descriptive error messages to identify error causes and locations.

Apart from the typical Lisp-style macros, Scala [Burmako 2013] has also explored a variety of macro categories: dynamic, string interpolation, implicit, type, and annotations. Every category is designed to address distinct applications scenarios such as "language virtualization, type providers, materialization of type class instances, type-level programming, external domain-specific

Table 1. Classification of Macro Systems Based on Taxonomy Subcategories and Other Aspects

Classification Dimensions \ Marco Systems	<bigwig>	Camlp4	Common Lisp	CPP	Dylan	ExJs	Gestalt	Honu	Inverse Macros	JSE	M4	Marco	Maya	MS²	Nemerle	Racket	Scala	Scheme	sweet.js	ZL
Lexical				✓							✓									
Syntactic	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Procedural		✓	✓				✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Pattern-based	✓			✓	✓	✓		✓		✓	✓		✓			✓		✓	✓	
Hygienic	✓				✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓

languages (DSLs) and language extensibility" [Burmako 2013]. A Scala extension [Yamaguchi and Chiba 2015] tries to further improve macro expressiveness through a new typed syntactic macro system called *inverse macro*. Inverse macros capture and operate on the continuation sequence of a macro invocation (i.e., the AST of the code following the macro invocation), supporting global AST rewriting and thus enabling the implementation of interprocedural operators or operators with complex side effects, like lazy and delimited continuation operators. *Gestalt* [Liu and Burmako 2017], a new macro system for Scala, further improved metaprogramming support in terms of simplicity, robustness, and hygiene, while it introduced the notion of portable macros by defining standard abstract syntax.

Table 1 classifies macro systems based on taxonomy subcategories and other considerations.

2.2 Reflection Systems

2.2.1 Model Details. Reflection is supported when a computational system can reason about and act upon itself, modifying its own structure and behavior at runtime [Maes 1987]. In this sense, a reflective system can perform computations on the system itself in the same way as for the target application, enabling one to adjust the system behavior based on the needs of its execution.

The two main conceptual aspects of reflection are *introspection* and *intercession*. Introspection is the ability of the program to examine its self-representation, while intercession is its ability to modify its self-representation. Another distinction is between *structural* and *behavioral* reflection. Structural reflection is the ability of a program to access the representation of its structure (e.g., classes, methods, and fields), while behavioral reflection is its ability to access the dynamic representation of its aspects (e.g., variable assignment, object creation, method invocation, etc.). The two distinctions are orthogonal; introspection and intercession relate to the kind of access (i.e., read-only or read/write) given to the self-representation, while the structural versus behavioral distinction relates to the type of the representation itself.

For object-oriented languages, reflection is closely coupled with the notion of having independent objects for base and metalevel computations: base objects perform computations for the target domain and metaobjects handle any metalevel computations, while the two can interact through well-defined interfaces. The latter relates to metaobject protocols, which we further discuss in the next section. Here, we discuss a relaxed notion of reflection that focuses more on introspection and structural features and less on intercession and behavioral features.

2.2.2 Representative Languages and Systems. Full reflection is difficult to implement without involving efficiency issues, requiring sophisticated techniques such as compilation via specialization based on staging annotations [Asai 2014], dispatch chains based on polymorphic inline caches for optimizations at the virtual machine level [Marr et al. 2015], speculative metaobject caches on top of dynamic compilers [Chari et al. 2016], and a combination of staging constructs and stage

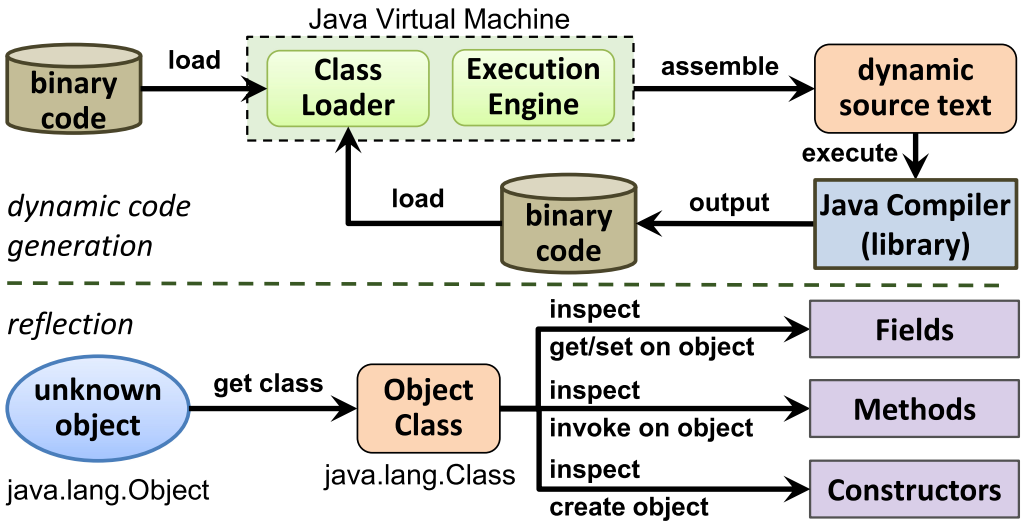

Fig. 4. Dynamic code generation (top) and reflection (bottom) in Java.

polymorphism to support abstracting over compilation versus interpretation [Amin and Rompf 2017]. Thus, in practice, many languages adopt this relaxed notion of reflection and couple it with runtime code generation facilities, as exemplified by Java.

Java reflection (Figure 4, bottom) supports structural introspection by examining the type and properties of an object at runtime and a limited form of behavioral reflection by dynamically creating object instances and invoking their methods. Both actions are supported through standard library APIs offered by the java.lang.reflect package. In particular, we can retrieve the class of an object, query the class structure for fields and methods, and use them to set or get an object field or invoke one of its methods. From the class object, we can also retrieve the class constructors to create new class objects. Furthermore, Java offers the compiler and class loader as runtime libraries (javax.tools.JavaCompiler and java.lang.ClassLoader), allowing for dynamically compiling and loading additional code during program execution (Figure 4, top). Using these facilities, we can turn a dynamic source text with new class definitions into executable code and then use reflection to query and use their functionality. Similar reflection facilities are also offered by other languages including C#, PHP, Python, and Ruby. For Java in particular, it is also possible to use some bytecode engineering library like *Javassist* [Chiba 1998] or *BCEL* [Apache Commons 2006], not only to create a new class but also to change the implementation of an existing class at runtime, thus providing support for structural reflection.

Runtime code generation based on source text can be impractical, inefficient, and unsafe, so alternatives have been explored based on ASTs and quasi-quote operators, offering a structured approach that is subject to typing for expressing and combining code at runtime. For example, *'C* [Poletto et al. 1999], *DynJava* [Oiwa et al. 2001], and *Jumbo* [Kamin et al. 2003] extend their respective base languages (C for 'C and Java for DynJava and Jumbo) with quasi-quote operators for expressing code in the host language, rather than through AST operators, and support metaprogramming through dynamic code generation. Jumbo performs type checking when code is generated at runtime, requiring only the result to be correct; this yields good expressiveness but no strong safety guarantees. On the contrary, 'C and DynJava impose restrictions to what can be expressed but offer static typing facilities that perform such checks during compilation. 'C is more expressive but also provides fewer safety guarantees, as code fragments lack context information

Table 2. Classification of Discussed Reflective Languages and Systems Based on Taxonomy Subcategories

Classification Dimensions \ Reflective languages and Systems	BCEL	C#	Guaraná	Iguana/J	Java	Javaassist	JavaScript	Lisp	MetaXa	OpenJava	PHP	Python	Reflex	Ruby	Self	Scala	Smalltalk
Structural introspection	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Behavioral introspection			✓				↓	✓	✓			↓	✓	↓	✓		✓
Structural intercession	✓	↓				✓	✓	✓		✓		✓	✓	✓	✓		✓
Behavioral intercession		↓	✓	✓	↓		✓	✓	✓	↓	↓	✓	✓	✓	↓	↓	✓

Symbol ↓ means there is limited support for the feature.

and thus inconsistencies may arise in generated code. To offer stronger type safety guarantees, DynJava relies on annotating dynamic code fragments with the type and context information they involve. Such annotations, called *code specifications*, are essentially typed quasi-quotes that contain metadata about the free variables they contain. This information can then be used to check inconsistencies at compilation time. Similar type safety guarantees in a dynamic code generation context are offered by the *Mnemonics* [Rudolph and Thiemann 2010] library that spots most bytecode verification errors at compile time of the generator. However, Mnemonics specifies generated code using bytecode instead of quasi-quoted source code.

Mirrors [Bracha and Ungar 2004] constitute a design principle for offering reflection facilities based on three key principles: encapsulation, stratification, and ontological correspondence. Compared to traditional reflection, mirrors do not pollute the object interface, allow different mirror systems or the ability to remove the mirror system, and support remote code operation. Languages that support reflection through mirrors include Self, Scala, JavaScript, and Dart.

Reflection is not limited to runtime systems. There are also compile-time systems that rely on some form of structural introspection to perform code generation. In such systems, the code generator is typically a metaprogram that queries about the fields and methods of a target class in order to generate object language code accordingly. Such systems primarily focus on code generation, so they are discussed in the generative programming section (Section 2.5).

Table 2 classifies the discussed reflection systems based on taxonomy subcategories. It also includes systems featuring metaobject protocols that are discussed in the following section.

2.3 Metaobject Protocols

2.3.1 Model Details. In their original sense, Metaobject Protocols (MOPs) [Kiczales et al. 1991] are interfaces to the language enabling one to incrementally transform the original language behavior and implementation, with the MOPs of Smalltalk and the Common Lisp Object System (CLOS) being representative examples. In a more relaxed sense, a MOP allows accessing and manipulating the structure and behavior of objects within the object system. Classes are considered to be objects of metaclasses, called metaobjects, that are responsible for the overall behavior of the object system and provide the API to modify this behavior (e.g., by creating new methods or modifying existing method code, changing the class structure by inheriting from different classes, etc.) (Figure 5). In this sense, modifications performed on a metaobject through the MOP at a metalevel are directly reflected on all objects deriving from it on the base level. This functionality is the typical example of a *metaclass-based approach*, but there are other designs for relating classes and metaobjects. In particular, in a *metaobject-based approach*, classes and metaobjects are distinct, with objects sharing their class for structural purposes but each of them having a separate metaobject for behavioral purposes. In a message passing context, a third approach is based on *message reification*, where messages are objects of a message class able to react to the *send* message, and

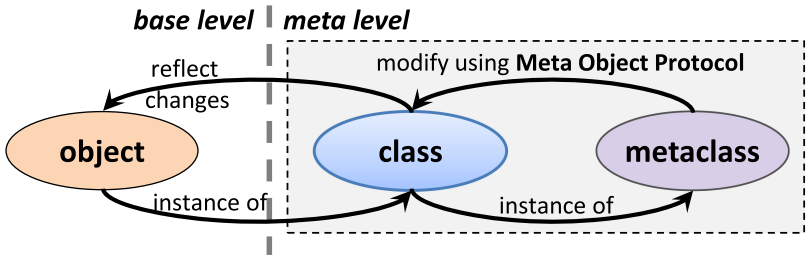

Fig. 5. Using metaclasses and Metaobject Protocols to modify the behavior of the object system.

message subclasses can override the default message interpretation, for example, as in the composable message semantics of Oberon [Hof 2000]. A more thorough discussion of these approaches, as well as a more in-depth overview of reflective systems and MOPs, is available in Tanter [2004].

2.3.2 Representative Languages and Systems. Many popular languages such as Python, Ruby, Groovy, and Perl offer MOP functionality based on metaclasses. More mainstream languages like C++ and Java do not offer MOPs, but there are various extensions of them that do so.

Iguana [Gowing and Cahill 1996] is a C++ extension that improved the modularity of metalevel architectures by introducing the concept of fine-grained MOPs. It allows multiple reflective object models to coexist in a given application and provides a flexible way of structuring customized metalevels from elementary building blocks, with the protocols themselves providing higher-level building blocks. It also allows reifying various concepts including object creation and deletion, message send, receive and dispatch, and state read and write. These principles were later ported to Java with Iguana/J [Redmond and Cahill 2002]. Iguana and Iguana/J support only one metalevel as opposed to most other systems based on the notion of reflective towers that offer an unbound number of metalevels.

MetaXa [Golm and Kleinöder 1999] features a runtime MOP and introduces the notion of *shadow classes*, as new class versions supporting reflective functionality associated with base-level objects when a metaobject is attached on them. Metaobjects can be stacked on top of each other to form a metaobject chain, thus enabling metaobject composition. Guaraná [Oliva and Buzato 1999] improves metaobject composition using *composer*s, that is, metaobjects enabling one to define modifiable and configurable policies for delegating control to other metaobjects. This enforces a clear separation between the reflective levels of an application and distinguishes between metaobjects that implement metalevel behavior from composers that define policies of composition and organization. Another orthogonal way to support metaobject composition is to adopt a general-purpose object-oriented mechanism, namely, *traits* [Ducasse et al. 2005]. Class properties can be represented as traits, grouping methods as reusable units that can be used to compose classes, and thus metaobjects, being classes themselves, can be composed out of traits. This offers a uniform and safe way of metaobject composition that inherits the automatic and explicit resolution of composition conflicts of normal trait composition.

MetaJ [Douence and Südholt 2001] features a generic reification technique based on program transformation using the notion of reflective towers. The source code representation of language features requiring reification is transformed from the nonreflective interpreter version into the reflective interpreter version, and the latter can be used at runtime to build a reflective tower. This approach is selective and enables fine-grained reification of arbitrary parts of object-oriented metacircular interpreters, with individual objects allowed to be reified independently and program transformation applied to different interpreter definitions. Effectively, this supports deriving different MOPs from the original interpreter definition. *Reflex* [Tanter et al. 2003], an open reflective

Table 3. Classification of Languages Featuring MOPs Based on Taxonomy Subcategories

Classification Dimensions \ Languages with MOPs	CLOS	DeepJava	Green	Groovy	Guaraná	Iguana - IguanaJ	Jasper	MetaJ	MetaXa	Oberon	OpenC++	OpenJava	Python	R-Java	Reflex	Ruby	Smalltalk
Metaclass based	✓	✓				✓	✓	✓			✓	✓	✓				✓
Metaobject based			✓	✓	✓				✓					✓	✓	✓	
Message reification										✓							

Java extension, has a similar philosophy targeting to build or adapt reflective extensions using its two main concepts: a generic MOP with a single method for any system event and class builders that transform normal classes into reflective classes by introducing metaobject invocations. It also supports partial behavioral reflection based on hooksets, where execution points are grouped into composable sets that may crosscut object decomposition, and a configurable link is used to attach metalevel behavior to these sets.

Dynamic shells and *extensions*, adopted in *Green* [Guimarães 1998] and *R-Java* [Tomioka et al. 1998], offer efficient and type-safe MOP functionality for statically typed languages. Shells are pseudo-objects with methods and instance variables that may be attached to other objects, causing messages sent to the actual object to be intercepted by the shell, similar to a delegation-style invocation. Extensions offer similar functionality but for classes, allowing a program to replace the methods of a class and its subclasses by the methods of another class at runtime.

All previous systems involve runtime MOPs; however, systems with compile-time MOPs also exist. For example, in *OpenC++* [Chiba 1995] and *OpenJava* [Tatsubori et al. 2000], metaobjects are available during compilation, providing a compile-time reflection mechanism used to manipulate source code and provide class translation through a method called *type-driven translation*. After parsing the original source, the system generates a class metaobject for each defined class. Then it deploys the class metaobjects to translate the target class to normal language syntax, repeating the process if further metaobjects are involved, and finally sends it to the original language compiler for translation. *Jasper* [Nizhegorodov 2000], a reflective Java syntax processor, offers a compile-time MOP with metaclass extensions that enable specifying custom factories for creating metaobjects, as well as syntactic transformations that manipulate standard metaobjects. Syntactic transformations rely on type-safe syntax templates that can be compiled into bytecode at template-definition time, enabling clearly separating implementing from implemented functionality. Compile-time MOPs, including OpenC++, OpenJava, and Jasper, typically support a single metalevel. *DeepJava* [Kühne and Schreiber 2007] features a compile-time MOP that supports an unbound number of metalevels for describing relations between *clabjects*, that is, elements that act both as objects and as classes. It features a *potency-based deep instantiation* mechanism that specifies the presence of object fields for instances of owning (i.e., lower-level) clabjects and enables creating new classes at runtime just like objects without compromising static type safety. Overall, compile-time MOPs operate as advanced macro systems that perform code transformation based on metaobjects rather than on text or ASTs.

Table 3 classifies the discussed systems featuring MOPs based on taxonomy subcategories.

2.4 Aspect-Oriented Programming

2.4.1 Model Details. Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] is a methodology for modeling crosscutting concerns into modular units called *aspects* (Figure 6). Aspects contain information about the additional behavior, called *advice*, that will be added to the base program by the aspect as well as the program locations, called *join points*, where this extra

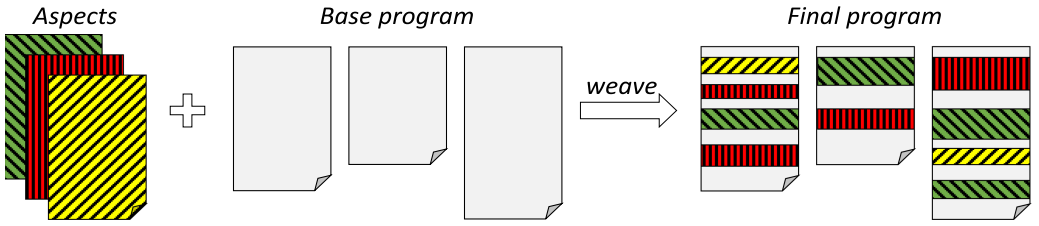

Fig. 6. Separating crosscutting concerns (aspects) and weaving them along with base program functionality based on some matching criteria in order to form the final program.

behavior is to be inserted based on some matching criteria, called *pointcuts*. Combining the base program with aspect code forms the final code incorporating the crosscutting functionality from all aspects (Figure 6, right). This process is called *weaving* and may take place during program compilation or execution. The first option, called *static AOP*, typically uses a separate language and a custom compiler, called *aspect weaver*, which combines the base program with the aspect program to form the final program. In the second option, called *dynamic AOP*, the bytecode contains no direct AOP functionality but is instrumented to be able to weave the aspect code at runtime. No separate language is typically involved, with AOP constructs offered and deployed as host language entities. Static AOP yields better performance as it has less runtime overhead, while dynamic AOP is more flexible allowing one to dynamically plug or unplug aspects at runtime.

AOP ideas originate from reflection and MOPs so there is a close relation between metaprogramming and AOP. In particular, AOP supports metaprogramming as a disciplined program transformation approach that adopts a pattern-matching-based approach to express crosscutting concerns. This may limit expressiveness compared to algorithmic transformations but enables handling metaprogramming tasks involving crosscutting concerns with increased flexibility and abstraction, resulting in more concise, straightforward, and safe implementations.

2.4.2 Representative Languages and Systems. *AspectJ* [Kiczales et al. 2001] was the first language and reference implementation of AOP, extending Java with constructs for aspect definitions, pointcut designators, and advice. *AspectC++* [Spinczyk et al. 2002] brought AOP to C++, adopting AspectJ concepts for similarity and familiarity and adapting them to the C++ syntax and semantics, while also introducing new C++ specific features, such as *generic advice*. AspectJ and AspectC++ are representative static AOP systems, both generating binary code that incorporates AOP functionality. Representative dynamic AOP frameworks for Java include *JAC* [Pawlak et al. 2001], *AspectWerkz* [Bonér 2004] (now merged with AspectJ), *PROSE* [Nicoara and Alonso 2005], and *Spring* [Johnson 2011]. JAC and PROSE both use normal Java classes for specifying aspects, and weavers constitute Java programs deploying aspect objects onto application objects. AspectWerkz and Spring offer similar runtime weaving functionality but define aspects, pointcuts, and advice using Java annotations or external XML configuration files.

AOP can support metaprogramming by inserting code before, after, or around matched joinpoints, as well as introducing data members and methods through intertype declarations. However, it is typically the other way around, as most systems offering AOP support rely on metaprogramming techniques for its delivery, using both runtime and compile-time approaches. The latter is directly correlated with the type of AOP the system provides, that is, static or dynamic.

AspectS [Hirschfeld 2002] and *AspectL* [Costanza 2004] support AOP by building respectively on the runtime MOPs of Smalltalk and Lisp. Reflex also uses its MOP to support AOP [Tanter et al. 2008]. *AspectR* [Bryant and Feldt 2002] is a Ruby library utilizing metaprogramming techniques to

Table 4. Classification of AOP Languages and Systems Based on Taxonomy Subcategories

AOP ianguages and Systems / Classification Dimensions	AOP++	Aspect.Net	AspectC++	AspectJ	AspectL	AspectR	AspectS	Aspect Scheme	Aspect Werkz	Delta	Groovy AOP	Handi-Wrap	JAC	PostSharp	PROSE	Reflex	Spring
Static AOP	✓	✓		✓				✓		✓				✓			
Dynamic AOP			✓		✓	✓	✓		✓		✓	✓	✓		✓	✓	✓

implement AOP through class method adapters. Most Java frameworks for dynamic AOP perform aspect weaving through load-time or runtime bytecode engineering.

On the other hand, *Handi-Wrap* [Baker and Hsieh 2002a] and *AspectScheme* [Dutchyn et al. 2006] are respectively Java and Scheme extensions that use macros to support AOP. *AOP++* [Yao et al. 2005], a C++ AOP framework, uses template metaprogramming to define pointcuts and match joinpoints at compilation time. *Groovy AOP* [Kaewkasi and Gurd 2008] provides a hybrid dynamic AOP implementation based on metaprogramming and bytecode transformation; aspects, pointcuts, and advice are specified at compile time based on a Groovy DSL, while advice is woven into bytecode at runtime using dynamic compilation.

There is also work in the direction of combining AOP with metaprogramming. *Meta-AspectJ* (MAJ) [Zook et al. 2004] is a Java extension to define templates that produce AspectJ code. MAJ adopts generative programming techniques to develop DSLs on top of AspectJ as well as extend AspectJ itself. More recently, Lilis and Savidis [2014] explored the combination of the two practices by introducing AOP support for metaprograms and the entire processing pipeline of a multistage language. They identified three aspect categories to fully support AOP in a multistage language: (1) *prestaging aspects*, used to introduce staging or transform existing stages in the original code; (2) *in-staging aspects*, used to apply typical AOP for stage metaprograms; and (3) *post*staging, used to apply typical AOP transformations in the final program.

Table 4 classifies the discussed AOP systems against the taxonomy subcategories.

2.5 Generative Programming

2.5.1 Model Details. Generative programming is a "software development paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge" [Czarnecki and Eisenecker 2000] (Figure 7). In the context of metaprogramming, generative programming involves programs that manipulate and generate other programs based on some algorithm and a program representation, typically that of an AST. In this sense, they are closely related to macro systems; however, macros tend to blur the distinction between normal code and metacode (typically, only macro definitions require extra syntax), while in generative programming systems code generation directives are clearly marked.

We review languages and systems offering support for generative metaprogramming based on *templates, AST transformations, compile-time reflection, traits,* and *class composition.*

2.5.2 Representative Languages and Systems.

Templates. C++ supports metaprogramming through its template system. Metaprogramming is achieved by instantiating template code (i.e., skeleton classes and functions) with specific type and nontype parameters producing concrete C++ code that is then compiled. No free-form source code generation is allowed. Nevertheless, C++ templates feature a Turing Complete functional language [Veldhuizen 2003], allowing expressing any compile-time computation, meaning that with

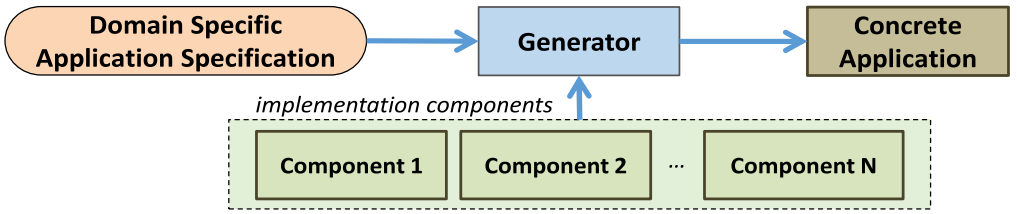

Fig. 7. Using a program generator to automatically create a concrete application from application specification based on elementary, reusable implementation components.

the appropriate metaprogramming logic for type manipulation (e.g., template class composition, class selection), it is computationally possible to express any generative metaprogram. This feasibility is theoretical, as in practice, the implementation complexity involved for elaborate metaprogramming scenarios outweighs the achieved benefits.

AST Transformations. Most generative programming systems instead offer code templates through quasi-quotation to support AST creation and composition and complement them with AST traversal or transformation features to support metaprogramming functionality.

The *Jakarta Tool Set* (JTS) [Batory et al. 1998] supports creating domain-specific languages using *Jak,* a Java extension that supports AST construction and manipulation, and *Bali*, a parser generator for creating syntactic extensions. A JTS component consists of a Bali grammar file for the extension syntax and a set of Jak files for the extension semantics; the domain-specific program is initially parsed into an AST using the parser generated by the extension syntax and then the AST is modified through the Jak transformation program into a pure Java program.

Groovy [Subramaniam 2013] supports AST creation through the *ASTBuilder* class and AST transformations as classes implementing the *ASTTransformation* interface. Local transformations are applied on any target class through annotations, transforming only the class itself. To offer global transformations (i.e., applied on the entire program code), one may also provide the compiler with an *ASTTransformation* subclass binary to be loaded and executed on the whole program AST. OCaml *PPX* extensions points [White 2013], introduced as a replacement for the Camlp4 preprocessor, operate in a similar way supporting AST quasi-quotation and AST transformations. They constitute external programs, passed to the compiler in binary form, that operate on the program AST by using the special attribute and extension nodes, receiving as input and providing as output OCaml AST in serialized form.

Java annotations processors [Darcy 2006] enable retrieving annotated source code elements such as classes and inspecting their internal structure, effectively offering an AST traversal. Annotation processors are normal Java classes extending the *AbstractProcessor* class, whose binaries are registered with the javac compiler to process annotations present on input source files. Compared to other systems, no AST creation facility is offered, and code generation can only create new Java sources files but not alter the code for existing Java classes. *Project Lombok* [Kimberlin 2010] overcomes this limitation and enables functionality similar to Groovy AST transformations. It offers an extensible collection of annotations featuring useful code generation patterns directly applicable to Java classes. Internally, it assumes that an annotation processor running in javac will be an instance of *JavacAnnotationProcessor* and then uses a nonpublic API to modify the existing AST. Despite relying on a nonstandard and fragile approach, Project Lombok has a large user base as, being just a Java library, it is easy to adopt.

Backstage Java (BSJ) [Palmer and Smith 2011] extends Java with quasi-quotations and algorithmic, context-aware code generation and transformation facilities. A built-in context variable

present in a metaprogram serves as the entry point to the AST and enables AST traversal and modification. Its novel feature is a dependency-driven execution order that retains determinism in case of nonlocal changes based on a difference-based approach that views metaprograms as transformation generators. It executes metaprograms independently on AST copies without observing changes from other metaprograms they do not depend on, recording changes to *edit scripts* and merging them to produce the final program. Merge failures denote metaprogram conflicts that are automatically detected and reported by the system.

Mython [Riehl 2009] is a Python variant allowing extending compilation through a quotation mechanism and code transformations. Mython quotations take an extra parameter used for parsing and extending the compile-time environment, allowing embedding of other languages by specifying compile-time definitions able to translate the embedded code into Python. Such translations return host language ASTs and the possibly modified compile-time environment. New names can be bound in the compile-time environment via a special translation function offered by the compiler. Finally, compiler built-in functions become first-class values, enabling customized and offering support for domain-specific optimizations.

Fan [Hongbo and Zdancewic 2013] is a metaprogramming system for OCaml that features a unified abstract syntax representation defined using polymorphic variants and supports nested quasi-quotes for the full language syntax, allowing them to be overloaded and customized. It also supports syntactic extensions based on *delimited, domain-specific languages* (DDSLs) implemented as libraries. In fact, quasi-quotes are a DDSL library bundled with the compiler.

Stratego/XT [Bravenboer et al. 2008] is a language and toolset for program transformations based on rewrite rules and strategies. It supports concrete object syntax, allowing expressing the transformations in terms of the host language, resembling the quotations of the previously discussed systems. Transformations differ from previous systems as they rely on pattern-based rewrite rules that operate as syntax transformers. Apart from standard rewrite rules, Stratego supports dynamic rewrite rules for context-sensitive transformations, and rewrite strategies to configure rule evaluation. Additionally, it offers extensible transformation modules, as well as tools for generating new ones from specifications, effectively enabling the development of entire program transformation systems. For example, SugarJ [Erdweg et al. 2011] extends Java with library-based syntactic language extensibility using Stratego for syntactic transformations.

Compile-Time Reflection. Apart from code templates, many generative programming systems also offer compile-time reflection features to enable generating code based on existing code structures, while emphasizing static-type safety, that is, ensuring that the generator will always produce well-formed code for any input. *cJ* [Huang et al. 2007] extends Java with compile-time predicates that can be associated with type parameters of generic classes, used across the class body to conditionally configure the generic class instantiation by matching criteria over its type parameters. It differs from other conditional compilation techniques (e.g., C/C++ #ifdef) as it is statically type safe and allows modular type checking of classes independent of their usages. *Genoupe* [Draheim et al. 2005], a C# extension with support for compile-time program generators based on generic types, offers a similar conditional construct and a loop construct that allows iterating over fields or methods of a type parameter and generates code for each match. Genoupe offers a type system with a high degree of static safety; however, it cannot guarantee that the generated code is always well typed. *CTR* [Fähndrich et al. 2006] also extends C# with program generators. It offers compile-time reflection and uses a *transform* construct to implement pattern matching and code generation. It avoids explicit quoting and unquoting to keep new syntactic constructs and keywords to a minimum and offers metavariables that can symbolically represent named program elements. It guarantees type safety as generated code is statically checked to be well-formed even for compiled transform entities. CTR overcomes some of Genoupe's

shortcomings, offering stronger type-safety guarantees and allowing one to extend existing elements as opposed to only generating new ones. *MorphJ* [Huang and Smaragdakis 2011] extends Java with pattern-based reflective declarations and introduces the notion of class morphing as a code generation technique that generate classes based on the structure of other classes. It offers improved expressiveness through nested patterns that can specify positive or negative existential conditions on top of the main pattern without sacrificing safety. It also improves on the type system of CTR as it allows type-checking generic class declarations independently of their instantiations, thus catching any errors directly at the original class definition. Additionally, it does not introduce new constructs as code generation is supported through generic classes. *SafeGen* [Huang et al. 2005] is another metaprogramming system for Java targeting type-safe code generation. It features *cursors*, being symbolic variables matching program elements against syntax predicates (in first-order logic), and *generators* that use *cursors* to output code fragments. Generators are written in a flexible quasi-quote style and together with cursors essentially form a comprehensive rewrite system. SafeGen is more expressive than MorphJ as its reflective declarations allow arbitrary first-order logic expressions compared to the more constrained pattern matching of MorphJ. However, SafeGen's type safety is statically determined by constructing first-order logic sentences and checking their validity through a theorem prover. This results in an undecidable type system that may fail to type valid code generators. A more detailed discussion of statically type-safe program generation techniques and systems is available in Smaragdakis et al. [2017].

Class Composition. There are also systems focusing on composition approaches such as *mixins* and *traits*, for improved flexibility and expressiveness. *Metatrait Java* (MTJ) [Reppy and Turon 2007] is a Java extension allowing compile-time code generation and introspection based on member-level patterns. It introduces user-customizable traits that are parameterized over types, values, and names offering compile-time pattern-based reflection. Traits support a uniform, expressive, and type-safe way for metaprogramming without resorting to ASTs, with their type system incorporating a hybrid of structural and nominal subtyping. *PTFJ* [Miao and Siek 2012, 2014] generalizes MTJ trait functions, combining pattern-based reflection with traits and providing language features for manipulating sets of member declarations like giving them names, manipulating their domains using set operations, and passing them as arguments to traits. Pattern-based reflection is supported at statement level, enabling metaprograms to generate statements. Regarding expressiveness, PTFJ is close to MorphJ. They differ in that PTFJ does not have MorphJ's negative nested patterns, but it can extend a class in place, while MorphJ cannot. *MetaFJig* [Servetto and Zucca 2010, 2014] is a Java variant supporting metacircular (i.e., homogeneous) class composition. It treats classes as first-class values, so that new classes can be defined based on existing classes through a set of primitive composition operators, namely, *sum, restrict, alias*, and *redirect*. Compilation is based on a series of metareduction steps called *compile-time execution* that derive nonconstant class declarations in the context of the current metaprogram. This is a modular process that is guaranteed to be sound by interleaving metareduction steps with type checking that dynamically detects class composition errors. MetaFJig also ensures *metalevel soundness*; this means that no typing errors during compile-time execution can originate from metacode that has already been compiled. MetaFJig is more expressive than MTJ, PTFJ, and MorphJ as it offers the full base language for expressing transformations instead of having a custom pattern-based metalanguage, but at the cost of the compilation process being nondeterministic and with no termination guarantees.

Feature-oriented programming approaches also fall into this category. For example, *FFJ* [Apel et al. 2008] is a feature-oriented extension of Java where features can create new classes or extend existing ones by class refinements, while statically guaranteeing type-safe feature composition.

Table 5 summarizes the classification of discussed systems based on taxonomy subcategories.

Table 5. Classification of Generative Programming Systems Based on Taxonomy Subcategories

Classification Dimensions \ Generative Programming Systems	BSJ	C++	cJ	CTR	D	Fan	FFJ	Genoupe	Groovy	Java	JTS	Lombok	MAJ	MetaFlig	MorphJ	MTJ	Mython	PPX	PTFJ	SafeGen	Stratego/XT	SugarJ
Templates/generics		✓			✓					✓												
AST transformations	✓					✓			✓	✓	✓	✓					✓	✓			✓	✓
Compile-time reflection			✓	✓				✓					✓		✓	✓			✓	✓		
Class composition							✓							✓		✓			✓			

2.6 Multistage Programming

2.6.1 Model Details. Multistage programming (MSP) extends the multilevel programming [Glück and Jørgensen 1996] notion of dividing a program into levels of evaluation and makes them accessible to the programmer through special syntax called *staging annotations* [Taha and Sheard 1997]. Such annotations are introduced to explicitly specify the evaluation order of the program computations, enabling the creation of delayed (i.e., future stage) computations and their execution in the current stage. In this sense, MSP is closely related to partial evaluation and program specialization, as the explicit evaluation order allows specializing generic and highly parameterized programs in a way that does not involve unnecessary runtime overhead. From another perspective, MSP can also be seen as a special case of program generation; delayed computations are practically code fragments in AST form built through quasi-quotation, while execution in the current stage essentially performs in-place code generation (Figure 8). In this sense, MSP is also related to procedural macro systems, where invoking a function that returns a delayed computation and executing it in the current stage resembles macro expansion. The previously discussed syntactic difference applies here too; macro systems use extra syntax for definitions and normal (function-like) syntax for invocations, while in MSP the extra syntax is needed at the call site to execute the returned delayed computation.

Languages adopting MSP typically support an unbound number of stages and are thus called *multistage languages* (MSLs). However, there are also languages offering exactly two stages of evaluation, called *two-stage languages*. MSLs are also categorized as *homogeneous*, when the metalanguage is the same as the object language, or *heterogeneous*, when the metalanguage and object language are different. MSLs with an unbound number of stages are always homogeneous.

2.6.2 Representative Languages and Systems. MSP was first applied in MetaML [Taha and Sheard 1997], extending ML with staging annotations: (1) *brackets*, used to create delayed computations; (2) *escape*, used to combine delayed computations; (3) *run*, used to execute a delayed computation at the current stage; and (4) lift, used to convert a ground value into code. MetaML introduced *cross-stage persistence (CSP)*, as the ability to use in future stages values that are available in the current stage, and *cross-stage safety*, which disallows variables bound at some stage to be used at an earlier stage. MetaML relies on runtime code generation and offers strong type safety guarantees, ensuring that if the generator is well typed, any generated programs are also well typed. *MetaOCaml* [Calcagno et al. 2003], another early MSL, extended OCaml with staging features and operated as a compiled MetaML variant, implemented through a combination of ASTs, gensym, and runtime reflection. It made CSP explicit in the language and introduced the notion of *environment classifiers* [Taha and Nielsen 2003], which are special identifiers annotating code fragments and variable declarations, whose scoping mechanism ensures that certain code fragments are closed and safely runnable.

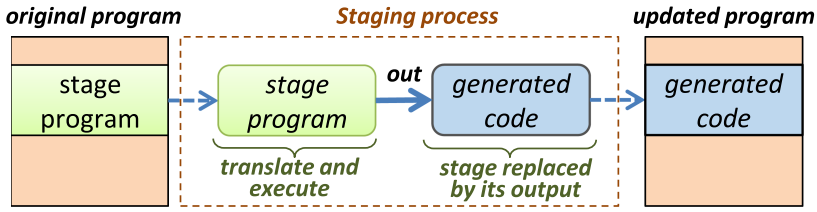

Fig. 8. MSP: Current stage execution of delayed computations performs in-place code generation.

The relation between MSP and macros was identified early, with macros realized as multistage computations in *MacroML* [Ganz et al. 2001]. MacroML extended ML with generative macros, featuring static type checking and hygienic macro expansion. It allowed introducing new language constructs capable of binding variables and supported recursive macros but not higher-order macros or macro-generating macros. Later, Yallop and White [2015] proposed an OCaml extension for compile-time generative macros by combining MetaOCaml constructs (quoting and splicing) and a new keyword for macro expansion. The system was termed modular macros, as macros become part of the module system. Recently, the Dotty Scala compiler offered uniform support for both compile-time macros (similar to MacroML) and MSP runtime code generation (similar to MetaML) [Stucki et al. 2018]. The system relies on a level-counting *phase consistency principle* and supports generating and splicing both terms and types.

MSP has also been studied in the context of imperative languages. For example, *Metaphor* [Neverov and Roe 2006] is a C# variant that offers MSP features. It can statically guarantee type safety of generated code and integrates staging features with the C#-based refection system to allow code generation based on type information. It also treats both types and code as first-class values, enabling generation and analysis of both code and types. Combining MSP with imperative features is difficult, due to scope extrusion, in which a variable may accidentally end up being used in a different scope than the one it was bound in. To this end, Kameyama et al. [2009] introduced $\lambda_1^{\emptyset}$, a two-stage language with two delimited continuation operators, namely, *shift* and *reset*, able to statically ensure that all generated code is well formed. The language was based on a two-level calculus with control effects and a sound type system. The key idea to prevent scope extrusion was to restrict control effects to the scope of generated binders, that is, treat them as control delimiters. This means that code in quasi-quotes should not have observable side effects. This is relaxed in *Mint* [Westbrook et al. 2010], an MSP Java extension. In Mint, specific terms can be declared as *weakly separable,* meaning they do not have observable side effects that involve code values. With escaped terms being weakly separable, Mint can guarantee that no code value leaves the scope where it is generated, retaining type safety and making the system more expressive. Restrictions still apply as requiring final classes for nonlocal operations within escapes excludes a big part of the standard Java library. Rhiger [2012] further improved lexically scoped stage computation by introducing $\lambda^{[]}$ and a type system that supports multiple stages, evaluation under future-stage binders, and open code manipulation.

MSLs are typically homogeneous; nevertheless, Eckhardt et al. [2005] introduces the notion of *implicitly heterogeneous* MSP where object language and metalanguage are different but metalanguage constructs are first evaluated to object language constructs, enabling one to subsequently generate third-party object language code based on source-to-source translation specifications. This approach maintains a homogeneous MSP experience and its type safety guarantees and allows generators to be applicable to different object languages without requiring any changes. Conversely, *MetaHaskell* [Mainland 2012] provides a framework for supporting *explicitly heterogeneous metaprogramming* with multiple object languages. It offers a type system for an

idealized metalanguage and object language pair that guarantees that any closed object language term produced by a well-typed metaprograms is well typed. The framework is modular, enabling one to add new object languages with only minor modifications to the host language, to support type-level quantification over object language types and propagate type equality constraints. *Terra* [DeVito et al. 2013] is another case of heterogeneous metaprogramming, where Lua is used as a metalanguage to generate low-level language code for high-performance computing. The evaluation of Lua and the generation of Terra code share the same lexical environment and variable references are hygienic across the two languages, while execution between them is separated to avoid performance issues from high-level Lua features.

There are many MSLs offering exactly two stages of evaluation or supporting staging at compile time. For example, 'C, DynJava, and Jumbo are two-stage languages, as their quasi-quote operators allow specifying code fragments in the original source language level. C++ can also be seen as a two-stage language, separating template instantiation (first stage) from nontemplate code translation (second stage). *Template Haskell* [Sheard and Peyton Jones 2002] is also a two-stage language that extends Haskell with compile-time metaprogramming facilities through quasi-quotes and splicing. The first stage takes place at compile time, evaluating the top-level splices to generate object-level code, while the second stage takes place at runtime, where the object-level code is executed free of any metalevel computations. Other languages applying MSP during compilation include *Converge* [Tratt 2008], *Metalua* [Fleutot 2007], and *Delta* [Lilis and Savidis 2015]. Converge deploys the approach of Template Haskell in a dynamically typed object-oriented language. Metalua extends Lua with metaprogramming support, featuring a different underlying philosophy of strictly separating metalevels and shifting across them through staging annotations. Delta is another untyped language supporting compile-time MSP. Apart from the basic MSP constructs, it introduces extra staging constructs for expressing metalevel statements and definitions and features a new metaprogramming model where code segments of the same stage nesting are not treated as isolated staged expressions, but as an integrated program. The model views metaprograms equal to normal programs and proposes adopting common practices and tools for their development.

Languages with runtime staging focus particularly on providing static type safety guarantees, as code generation occurs at runtime and at that point it is too late to report any type errors. Type safety, though, comes at the cost of expressiveness; many useful generative metaprograms cannot be expressed, while metaprograms cannot operate as code analyzers, as allowing destructing or traversing quoted expressions may cause the type safety guarantees to be violated. Offering more expressiveness sometimes means offering less type safety guarantees. For example, Viera and Pardo [2006] propose an MSL with intentional analysis that relaxes the static safety in order to allow inspecting the structure of its object programs and destructing them into its subparts based on a pattern-matching mechanism. Recently, *Squid* [Parreaux et al. 2017], a Scala macro library supporting MSP, improved on this aspect by providing pattern matching on code and rewriting of open code through *speculative rewrite rules* while maintaining type safety. Languages with compile-time staging typically also offer fewer type safety guarantees, as they are allowed to report type errors during compilation and instead focus more on expressiveness. Nevertheless, this is not always the case; for example, in MetaFJig, where metareduction steps can be seen as stages, the *metalevel soundness* property ensures that operations on already compiled classes are type safe (as long as the input is also well formed) without reducing expressiveness with respect to arbitrary AST manipulation.

An alternative approach that supports MSP without explicit annotations has also been explored in *Lightweight Modular Staging (LMS)* [Rompf and Odersky 2010]. LMS is a library-based approach for MSP in Scala that avoids quasi-quotes and instead distinguishes between binding times based on types. Essentially, while MSP provides staging support for all language constructs by default but

Table 6. Classification of Multistage Languages Based on Taxonomy Subcategories

Classification Dimensions \ MSLs	'C	Converge	Delta	Dotty Scala	DynJava	Jumbo	LMS	MacroML	Meta Haskell	MetaML	Meta OCaml	Metalua	Metaphor	Modular Macros	Mint	Mython	Squid	Template Haskell	Terra	λ_1^{ϕ}	$\lambda^{\square}$
Two stage	✓				✓	✓			✓									✓	✓	✓	
Multistage		✓	✓	✓			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓				✓
Homogeneous	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
Heterogeneous									✓							✓			✓		

requires the programmer to annotate staged code, LMS involves no explicit staging but requires staged type operations to be provided by the programmer as traits.

Table 6 classifies the discussed MSP systems against the taxonomy subcategories.

3 PHASE OF EVALUATION

Metaprograms can also be categorized by when they are evaluated. We distinguish between three phases that a metaprogram can be evaluated at: (1) before compilation, as a preprocessing step; (2) during compilation; and (3) during execution. Each option has its own characteristics, implying various advantages and disadvantages, while their adoption is not exclusive. In theory, any combination of them is viable; however, in practice most metalanguages offer only one or two of the options, each of which may support different metaprogramming practices or level of expressiveness. Additionally, the phase of evaluation does not necessarily dictate the adoption of a particular metaprogramming model; however, there is a correlation between the two with most metaprogramming models typically encountered in one or two evaluation phases.

3.1 Preprocessing-Time Evaluation

3.1.1 Evaluation Characteristics. The first option for applying metaprogramming is before the translation of the normal program, during a preprocessing phase of the original source file. In this case, which we call *preprocessing-time metaprogramming* (PPTMP), metaprograms present in the original source are evaluated during the preprocessing phase and the resulting source file contains only normal program code and no metacode (Figure 9). Thus, translation can reuse the language compiler or interpreter without the need for any extensions. Systems operating this way are called *source-to-source preprocessors*, as both their input and output are in the form of source code. In cases where the original language compiler is available as a library, it can also be bundled with the preprocessor to operate as a back end that directly generates binary code, blurring the distinction between preprocessing-time and compile-time evaluation. The distinction between the two is practical and involves mostly design and implementation issues (modularity, extensibility, ease of development). There is no theoretical barrier regarding the expressiveness of each approach, as a PPTMP system need not operate only on a textual level, but may be fully aware of the language syntax and semantics and be as sophisticated as a compiler. In fact, there are cases such as C++ where early compilers were PPTMP systems translating C++ code to C code. In our discussion, systems that output source code and require an external compiler invocation are classified as PPTMP, while systems integrating the language compiler invocation are classified as compile-time evaluation systems.

3.1.2 Languages and Systems. All lexical macro systems fall into this category, as they are language agnostic and need to operate before the translator, with CPP and M4 being representative examples. CPP is used to expand all macro invocations within a C/C++ file and generate

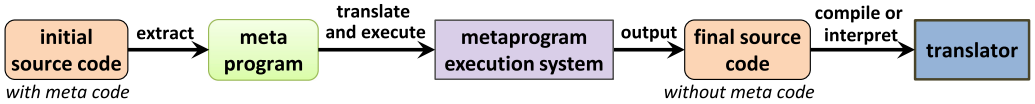

Fig. 9. Typical example of preprocessing-time metaprogramming.

a macro-free source code version that is then compiled by a C/C++ compiler (modern compilers apply the preprocessing phase as part of the compilation). Similarly, M4 copies its input to the output, expanding macros as it goes. On the other hand, syntactic macros can operate at any phase, including before translation, when they function as source preprocessors. Camlp4 operates on Ocaml source files, parsing and analyzing their source code and outputting a preprocessed source file that is sent for translation. Similarly, in sweet.js and ExJs, macro invocations in the input source are expanded to output pure JavaScript. In the Marco macro system, a Marco program along with external inputs (e.g., additional object language fragments) are used to generate a source file containing only object language code. Finally, the Racket macro-expansion step is also an example of a source-to-source compiler that takes the source code in one language and outputs source code in another one [Hasu and Flatt 2016].

Reflection, MOPs, and AOP are typically not encountered in PPTMP systems. An exception is the *Reflective Java* [Wu and Schwiderski 1997] preprocessor that generates reflection stub classes to support reflective method invocations through a MOP. Also, OpenC++ and OpenJava, while featuring compile-time MOPs, may be invoked as source-to-source preprocessors that generate pure C++ code and Java code, respectively. Regarding AOP, source preprocessing is mostly limited to systems adopting source-level weaving. For example, the AspectJ 1.0 compiler can be run in preprocessor mode, generating Java sources with the woven code, which are then compiled with a Java compiler. Similarly, AspectC++ transforms its input files into pure C++ files containing the woven code based on the source code transformation system PUMA. Delta aspectual transformations also fall into this category, as the aspect weaver operates as a source-to-source preprocessor that generates Delta source files that are then supplied to the compiler.

Many systems that support PPTMP fall into the category of generative programming, where metaprograms act as code generators that receive their input as source code specified in the metalanguage and translate it into object language source code. All such cases involve syntactic transformations, where input is parsed into AST, which is transformed by the metaprogram, and then unparsed to produce the final source code. For example, in JTS, the input source code, written in Jak, is parsed into an AST that is modified through a Jak transformation program and then finally unparsed into a Java program. Fan and Stratego/XT operate similarly, with the AST transformation taking place respectively through syntactic extensions and rewrite rules. The Genoupe compiler parses input files in Genoupe ASTs, transforms them into C# ASTs, and unparses them to output corresponding C# source files. SugarJ operates similarly, parsing the extended Java sources into a mixed SugarJ and extension node AST that is desugared to a pure SugarJ AST from which the Java part is extracted and unparsed. SafeGen metaprograms also operate on ASTs, creating and then generating Java code based on matching input entries. cJ translates code with type conditionals into Java classes and interfaces using generics, while MTJ and PTFJ generate pure Java classes incorporating the trait functionality. Similarly, MAJ generates pure Java code that uses an AST library for expressing AOP constructs.

3.2 Compilation-Time Evaluation

3.2.1 Evaluation Characteristics. Another option for applying metaprogramming is during the compilation of a target program (Figure 10). This approach, called *compile-time metaprogramming*

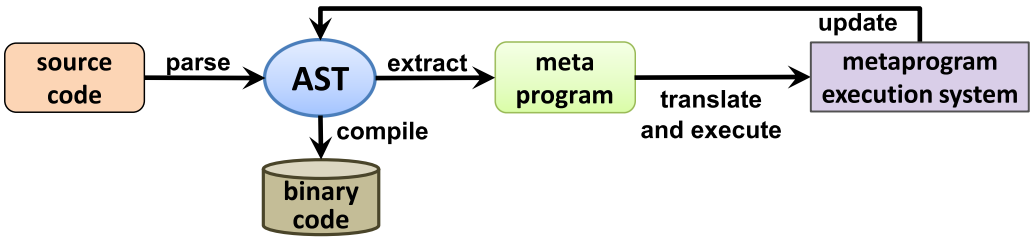

Fig. 10. Typical example of compile-time metaprogramming.

(CTMP), implies that the language compiler is extended to handle metacode translation and execution. Such extensions may take various forms like *compiler plugins, syntactic additions, procedural or rewrite-based AST transformations*, or *multistage translation*. Also, metacode execution can take two forms: *interpreting* the source metacode, that is, having a metalanguage interpreter within the compiler, or *compiling* the source metacode to binary and then executing it. Finally, all of the discussed metaprogramming models may be encountered in CTMP systems.

3.2.2 Languages and Systems. There are various macro systems operating during compilation. Many of them are supported as compiler plugins, where macros are programs that are compiled separately and used during the compilation of a target program. For example, Dylan, Nemerle, Maya, JSE, and Scala all compile their macros separately to binary (DLLs or Java bytecode) and use it as a compiler extension plugin. Racket macros can also be considered extensions to the compiler that expand syntax into existing forms. In Honu, macro bodies are compile-time expressions, separated from runtime as in Racket. Another option is evaluating metaprograms by an interpreter executed at compile time. MS^2 is such an example, as it uses an embedded interpreter for a subset of C. Many compiled versions of Lisp dialects also fall into this category, as they typically use an interpreter for compile-time macro-expansion. Finally, there are also systems that extend their compilation process to allow AST transformations. For example, the macro system of <bigwig> supports AST transformations using grammars and substitution.

There are also a few systems featuring compile-time MOPs. In OpenC++, code for metaobjects is separately compiled into binary and then dynamically loaded as a compiler plugin during the compilation of the client program. OpenJava generates Java bytecode for the compile-time metaobjects, uses that to perform a source-to-source translation to Java, and then uses the standard Java compiler to generate the final bytecode. Jasper offers metaclasses that allow extending its parsing, syntax-to-syntax transformations, and pretty-printing facilities. Metalevel code is compiled into Java bytecode and executed at compile time to override default MOP instances. In DeepJava, for every metalanguage class the compiler generates a set of Java classes to match type and instance facet definitions, effectively creating the clabject metalevel relation structure, as well as generates code to instrument the clabject instantiations at runtime.

Compile-time AOP systems typically involve compilers and weavers operating at bytecode. For example, since version 1.1, AspectJ uses bytecode weaving; normal code and aspects are compiled into binary form and woven together to produce bytecode files run on a Java Virtual Machine (JVM). Other compile-time bytecode weaving systems include PostSharp [Fraiteur 2008] and Aspect.NET [Safonov and Grigoriev 2005]. There are also other CTMP systems supporting AOP. For instance, AOP++ relies on template metaprogramming and uses the C++ type system to express pointcuts and match joinpoints at compile time. Moreover, Handi-Wrap and AspectScheme use compile-time macros to generate code that enables runtime aspect weaving.

Many systems supporting CTMP fall into the category of generative programming. C++ templates are instantiated as part of the type system, while *constexpr* expressions and functions are

evaluated at compile time by a C++ interpreter. An interpreter also handles compile-time function execution in D [Alexandrescu 2010]. In Groovy, AST transformations are performed at compile time, while transformation code must already be present in binary form before compiling the target program. OCaml PPX extensions and Java annotation processors operate similarly, being supplied respectively as external programs and jars to the compiler invocation. In BSJ, a successfully merged edit script is applied to the original AST, and the result is serialized into Java source code and then compiled to bytecode using the Java compiler. In MetaFJig, each compile-time metareduction step translates code into Java, compiles it to bytecode with the Java compiler, and executes it in the JVM to update the compile-time program representation.

Systems offering compile-time reflection are typically generative programming systems, as the reflection-based metacode eventually generates object language code. For instance, the transform constructs of CTR are separately compiled as DLLs and used at client code through C# attributes. After compiling client code into intermediate language, the compiler applies any matching transforms to update it and then generates the client code binary. MorphJ generic classes are compiled into nonstandard annotated bytecode files that are used as templates for expansion; when instantiated with a particular type during the compilation of a program that uses them, a copy of the annotated bytecode is modified based on the reflective iterator information, producing a valid Java bytecode file for the class instantiation.

There are also various two-stage or multistage languages supporting CTMP. Template Haskell relies on Glasgow Haskell Compiler's built-in bytecode compiler and interpreter to run splice expressions and generate code at compile time. Metalua metalevels are executed at compile time through a custom interpreter. In Converge, for each splice expression, the compiler generates a temporary module containing a function with the splice expression and other necessary definitions, compiles it into bytecode, loads it in the VM, and invokes the splice function to generate the AST value to replace the splice expression. Mython quotations refer to translation functions executed at compile time by compiling the input AST to Python bytecode and then evaluating that bytecode in the given environment. Delta uses the nesting of the staging tag annotations to assemble the entire source code for a specific stage and compiles and executes it with the original compiler and VM to collectively transform the target program. The process is repeated for further stage code; otherwise, the final program is normally compiled.

3.3 Execution-Time Evaluation

3.3.1 Evaluation Characteristics. Metaprogramming can also be applied during program execution (Figure 11), a process referred to as *runtime metaprogramming* (RTMP). Supporting such a form of metaprogramming involves extending the language execution system and offering runtime libraries to enable dynamic code generation and execution. RTMP is the only case where it is possible to extend the system based on runtime state and execution, for example, using new classes whose code is delivered over the network during execution. For interpreted languages, RTMP is closely coupled with the existence of an *eval* function that interprets a dynamic source text in a lexically scoped context. For interpreted languages with macro systems (e.g., some Lisp and Scheme implementations), macros are expanded and their replacement is directly evaluated, interleaving macro invocations with normal program execution during the single interpretation step (no separate compilation and execution). Apart from these, other metaprogramming models typically found in RTMP systems include reflection, MOPs, AOP, and MSP.

3.3.2 Languages and Systems. In Java and C#, the compiler and loader are offered as libraries that can be used at runtime to compile, load, and deploy any dynamic code. In 'C, quoted expressions are statically compiled into code-generating functions that use custom runtime systems for

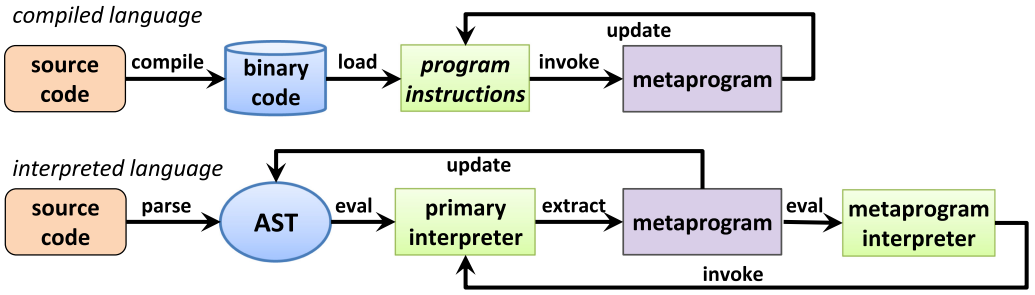

Fig. 11. Execution-time metaprogramming in compiled (top) and interpreted (bottom) languages.

code generation. DynJava similarly generates generator classes that at runtime produce executable code through a bytecode manipulation library. Jumbo combines a quotation syntax with a compiler API and allows creating and manipulating code values that generate and execute Java bytecode at runtime. Mnemonics relies on advanced typing features of Scala, such as type inference and function types, to generate method bodies in Java bytecode at runtime.

Most metaprogramming systems featuring MOPs fall in the category of RTMP. In both the Smalltalk MOP and the CLOS, metaobjects are available during execution and can be accessed and manipulated to alter the program behavior. The same applies for other languages with MOPs, including Python, Ruby, Groovy, and Perl, as well as various Java extensions. For the latter, two approaches are adopted to support the MOP functionality. The first is to extend the JVM to support metaobjects, as in R-Java, MetaXa, and Guaraná. Iguana/J also extends the runtime engine, but instead of a modified JVM, it uses a native dynamic library loaded through the nonstandard JIT compiler interface. The second approach is to use the standard JVM and deploy a bytecode engineering library to change the implementation of a class at runtime. For example, Kava [Welch and Stroud 1999] implements its MOP functionality using a load-time transformation framework, while Reflex uses the Javassist framework for load-time structural reflection. Finally, there is the option of providing the MOP functionality through an interpreter, as in the reflective Java interpreter MetaJ.

AOP systems operating at runtime are those supporting dynamic AOP, that is, where bytecode is instrumented to be woven during loading or execution. One option for implementing such a weaving process is to rely on a runtime MOP, as in AspectS, AspectL, and Reflex. A second option is to deploy a bytecode engineering library, as in JAC and *Handi*-Wrap, which use Javassist and BCEL, respectively, to alter bytecode during class loading. Another option is to rely on execution system built-in facilities or custom extensions. For example, AspectWerkz uses a custom class loader to weave joinpoints at class loading, while PROSE hooks code for joinpoints using the JVM Debugger Interface or the JIT compiler. In Groovy AOP, advice code is woven into bytecode at runtime using dynamic compilation. Finally, Spring AOP uses either JDK dynamic proxies or CGLIB to create proxies for given target objects.

In traditional MSLs like MetaML and MetaOCaml, code generation occurs during program execution. MetaML relies on an interpreter that can construct, type, and run object programs during interpretation. MetaOCaml uses a modified OCaml compiler that translates staging constructs into operations that build and manipulate ASTs, invoke the compiler at runtime, and execute the result. Metaphor and Mint adopt a similar approach. Metaphor and Mint translate brackets and escapes into a series of AST composition operations that recreate the AST at runtime, while the run operator compiles and executes the code at runtime using C# and Java dynamic code generation features. In LMS, staged code operations are specified as AST node traits, while

runtime code generation involves collecting all related ASTs, emitting the respective source code wrapped as a class definition, compiling it, loading the generated class file, and finally returning a newly instantiated object to perform the operation.

4 METAPROGRAM SOURCE LOCATION

Metaprograms can further be classified based on their source location. In this context, they may be embedded within the program they intend to transform as part of its source code or they may be located externally as separate transformation programs. We further elaborate on this classification and present systems within each category.

4.1 Embedded in the Subject Program

Metaprograms are typically part of the program they transform, often blended with normal code. Macro definitions and invocations are placed together with normal program code. Generative programming systems also usually have their code templates and generators located alongside normal program code. The same applies for MSLs, where staged code and code generation directives are distinguished from normal program code through staging annotations. Even when metaprogramming involves reflection systems or MOPs, any reflective behavior or interaction with metaobjects is specified within the program itself. Finally, in the case of static AOP, the metalanguage typically extends the host language, so aspect code is mixed with normal code, while in the case of dynamic AOP, aspect-oriented constructs are offered and deployed as host language entities, so they are again part of the subject program.

For a metaprogram present within the subject program, there are three transformation options (Figure 12) with respect to the location context: (1) *context unaware*, where the metaprogram generates code to be directly inserted in its place without any context information (like macro invocations); (2) *context aware*, where the metaprogram is aware of the location context and may transform any code fragment related to that context; and (3) *global*, where the metaprogram globally transforms or affects the entire program regardless of its location.

Context Unaware. This involves metaprograms that need not know the code generation context, but only any input parameters that are used by the transformation logic to generate an AST that replaces the original metacode (e.g., macro invocation or code generation directive). Context-unaware generation is very common in metalanguages, while for most macro systems (e.g., CPP, List, Scheme macros), generative programming systems (e.g., C++ templates, CTR, MorphJ) and MSLs (e.g., MetaML, Template Haskell, Metalua) it is the only available option.

Context Aware. This case is more general, supporting scenarios where the transformation logic takes into account the metaprogram location context. It typically involves providing access to the respective program AST node and allowing it to be traversed for read or write operations. Essentially, this AST node constitutes an extra input (or in-out) parameter to the metaprogram. This means that the metaprogram may transform code fragments not merely in a single context, but at multiple different locations reachable from the initial context. For example, a metaprogram applied on a method for synchronization could both introduce a mutex member variable and add the proper locking and unlocking code in its body. Context-aware generation is not widely adopted and is currently offered by only a few metaprogramming systems.

Groovy AST transformations, Java annotation processors, BSJ metaprograms, Nemerle macros, and PPX extensions are all applied through annotations contextually placed just before their targets that direct the compiler to apply the metaprogram logic supplying the annotated element AST. BSJ also allows using the built-in *context* variable to get the AST of its enclosing code entity and use it to transform its code. The inverse macros for Scala capture the AST of the code following the macro invocation until the end of the containing block and pass it as an argument to the

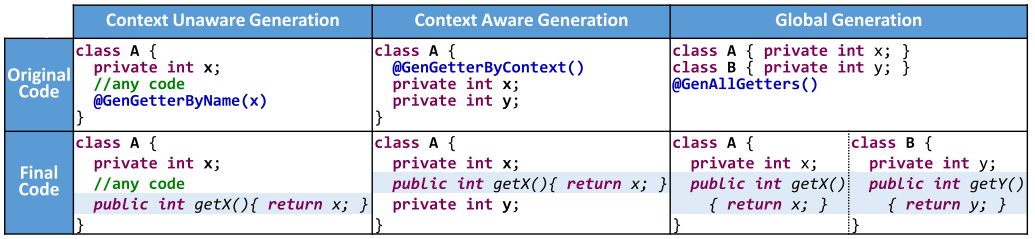

Fig. 12. The three transformation options for metaprograms embedded in a subject program with examples of original (annotated as @generator) and generated code (highlighted).

macro transformation method that can traverse and transform this AST argument and eventually return a modified version that will substitute the original. Python decorators are functions that can be applied on classes or functions through annotations to adapt their definitions. During parsing, annotations are desugared to calls of the decorator function with the annotated element as argument. The desugaring process itself or the resulting code do not involve any explicit context; nevertheless, from the perspective of the programmer the overall decoration process uses contextually placed annotations for code transformation. Delta supports context-aware code generation through its staged library function *std::context* that retrieves the parent AST of its actual call site in the context of the enclosing stage (or main program) being transformed, thus offering an entry point for AST iteration and manipulation.

Overall, each system may provide different options for accessing context, ranging from read-only to entirely mutable, but they all offer a specific source location as a metaprogram input.

Global. The global case relates to scenarios that collectively introduce, transform, or remove functionality for the entire program and is typically encountered in systems with reflection, MOPs, and AOP. For example, applying runtime compilation (e.g., in Java or C#) to generate a new class globally affects the state of the execution system and has the same effect regardless of the source location of the code performing the runtime compilation. Most MOPs also fall in the global case; metaobjects typically globally affect the structure or behavior of all related objects within the system. This is not always the case though; for example, the fine-grained MOP of MetaJ allows reifying individual objects independently, operating in a context-sensitive fashion. Introducing code through AOP advice can also globally transform a program. Being able to add restrictions to global transformations can also achieve a context-aware behavior. For example, depending on the AOP approach (i.e., static or dynamic) and the specified pointcuts, it is possible to transform code for specific classes instead of transforming the code for all classes altogether.

Global transformations are not necessarily limited to reflection, MOPs and AOP, or cases of RTMP. In fact, any PPTMP or CTMP system that provides access to the full program AST, either directly or indirectly through access to enclosing scopes and namespaces, can naturally introduce global transformations. This can also be seen as a context-aware case where the context is the entire program. Groovy global AST transformations constitute a representative example, where a target transformation is applied on the whole program AST, potentially affecting the entire program code. The same applies for Delta, where AST nodes have access to their parents, and thus any AST node retrieved from *std::context* may be used to effectively affect the entire program AST. This is not the case for BSJ, where despite AST nodes having access to their parents, metaprograms are restricted to affect only the declarations they appear in to limit nonlocal changes. Java annotation processors and project Lombok typically target annotated elements, thus operating in a context-aware fashion, but can also navigate across the entire program AST to globally perform read-only or read-write operations, respectively.

4.2 External to the Subject Program

Metaprograms may also be defined and applied externally to their subject program. In this case, they are specified as separate transformation programs applied through PPTMP systems or supplied to the compiler together with the target program to be translated as extra parameters.

Stratego/XT relies on externally provided rewrite rules to transform its input program, operating as a source-to-source preprocessor. In SafeGen, a metaprogram uses cursors to iterate on entities of the subject program that is supplied separately to the SafeGen compiler and then outputs code fragments to a generated Java source file through generators. Delta and its *Sparrow* IDE support applying AST transformation programs to a target program for AOP. Before compiling a target program, the build system invokes a separate weaver program to apply any transformation programs through PPTMP and then invokes the compiler on the updated program. The global transformations of Groovy are packaged as jars and supplied to the compiler during its invocation. The compiler parses the target program to AST, applies on it the supplied transformations, and uses the updated AST to continue its translation.

Many AOP scenarios can also be considered as external with respect to the target program, as aspects may be introduced to a program retrospectively or they may be dynamically plugged and unplugged from the system. For instance, an AspectJ source file may not introduce any Java classes, but only declare aspects targeting to transform other Java classes. Thus, the metaprogram may be entirely separate from the host program, which in fact may be ignorant of the crosscutting concern, and the transformation only occurs during the invocation of the AspectJ compiler. The same applies for dynamic AOP approaches; aspect code that can be added and removed on the fly at runtime may be part of the same execution system, but with respect to its source location it will typically be part of some other subsystem and source file.

5 RELATION TO THE OBJECT LANGUAGE

A prominent aspect for categorizing metaprogramming systems is the relation between the object language and the metalanguage. Effectively, every metaprogramming language can be seen in two layers. The first layer concerns the basic object language, and the second layer, which we will refer to as the metalayer, relates to the metaprogramming elements for implementing the metaprograms. In particular, there are a few languages that were designed from scratch to have a metalayer, while for others the metalayer has been introduced later, independently from the object language. Following, we study the relationship between the metalayer elements (metalanguage) and the object language. In this context, there may be no distinction between the metalanguage and the object language, the metalanguage may be an extension of the object language, or the metalanguage may be an entirely new language. In the first case, there is a single language used for both object code and metacode. The second case is similar, with the metalanguage offering all object language functionality along with some extensions for metaprogramming. A typical example of such extensions are the quasi-quotation constructs offered by most metalanguages. In the third case, the metalanguage is a different language from the object language, using different and typically fewer constructs from the object language.

Here we should make a couple of notes regarding the classification for languages extended for metaprogramming support. In some cases, the extended language used for specifying metacode may also be used for normal code, making the classification between the first or second category a bit subtle. Taking MetaML as an example, on the one hand it extends ML (i.e., can be classified in the second category), while on the other hand it can also play the role of both the object language and metalanguage, so it can be classified in the first category. Essentially, the classification relates to which language is considered as the object language, the original one or the extended

one. We classify such cases with respect to the original language, that is, as language extensions. In other cases, despite extending the original language with constructs for metaprogramming, the metalanguage is not necessarily an extension of the object language; in fact, in many cases the metalanguage consists only of these custom constructs and cannot use constructs of the host language. As such, we classify such cases as using custom metalanguages.

In case the metalanguage is the same as the object language or an extension of it, a further design choice is whether to reuse the runtime system of the host language or use a custom one. This dilemma is typically encountered in languages supporting CTMP, as in languages supporting RTMP it is usually straightforward to reuse the execution system or extend it to support the metaprogramming functionality. In this context, an ad hoc runtime may be easier to implement but involves maintenance issues as any changes on the object-level runtime system should be duplicated to the metalevel runtime system. On the other hand, reusing the same runtime as part of the compiler requires modular design and implementation, while it is typically restricted for bytecode-compiled languages; for natively compiled languages the only option is to use a custom runtime, such as a custom interpreter or JIT compiler.

5.1 Metalanguage Indistinguishable from the Object Language

Metaprogramming systems in which the metalanguage is indistinguishable from the object language fall in two categories. First, object language and metalanguage may use the same constructs through the same syntax. Second, metalanguage constructs may be modeled using object language syntax and applied through special language or execution system features.

A typical example of the first category is Lisp, whose macro system utilizes the full language itself to specify the transformation logic. In Lisp, there is no distinction between code and data, so any operation that can be performed on data can also be performed on code using the same execution system. Scheme syntax-case macros and their derivatives (e.g., Honu and sweet.js *case* macros) operate similarly, offering the full language functionality for syntax transformation. Nemerle macros and PPX extensions also use object language features for their implementation. Similarly, Groovy AST transformations and Java annotation processors are normal classes implementing a specific interface and can use all object language syntax and features. Nemerle macros, Groovy AST transformations, Java annotation processors, and PPX extensions are all compiled to binary (DLLs, Java bytecode, OCaml binary) and loaded by their compilers, reusing the host language execution system. C++ *constexpr* expressions and functions also use C++ syntax. In C++11, *constexpr* functions offered limited syntax, allowing a single return statement. C++14 relaxed many restrictions and supported more language constructs (e.g., local variables and loops for iteration rather than recursion), but *constexpr* functions are still limited to a subset of C++ syntax as specific requirements must be met (e.g., cannot use try-catch blocks). This also applies for compile-time function execution in D [Alexandrescu 2010]; only functions with portable and side-effect-free code may be run at compile time. Regarding their execution system, both C++ constexprs and D compile-time functions are evaluated by custom interpreters.

Examples of the second category are many systems supporting MOPs and AOP. MetaXa, Guaraná, Kava, Reflex, and MetaJ model metaobjects through normal language syntax. For their execution, MetaXa and Guaraná use a modified JVM version, Reflex and Kava reuse the standard JVM along with bytecode engineering libraries, and MetaJ relies on a custom interpreter. JAC and PROSE model AOP constructs as Java objects, while AspectWerkz, Spring, and PostSharp use Java annotations and C# attributes to specify AOP functionality in object language syntax. AspectS, AspectR, AspectL, and AspectScheme are libraries that deliver AOP functionality without language or execution system extensions. AspectS and AspectL utilize the language MOPs, AspectR uses method wrapping, and AspectScheme uses macros. It should be noted that in some AOP cases,

pointcuts are specified using strings based on an AspectJ-based pattern-matching language; thus, arguably a custom metalanguage is used as well.

LMS and Mnemonics also use language and execution system features for metaprogramming. LMS MSP facilities are specified using trait classes for AST nodes and dynamic compilation, and Mnemonics' dynamic code generation relies on type inference and function types.

5.2 Metalanguage Extends the Object Language

When a language is extended with metaprogramming features, the aim is to exploit the original language features in developing metaprograms, offering a metalanguage that is an extension of the base language rather than a restriction and that reuses well-known features instead of adopting custom programming constructs. Such language extensions typically involve new syntax and functionality used to differentiate normal code from metacode and how the latter is deployed.

A common example of such syntax are the quasi-quote constructs found in many macro and generative programming systems. For example, the MS^2 macro language is a C extension that offers a template substitution mechanism. 'C also extends C with quasi-quote operators while adding extra type constructors and some special forms for handling dynamic code. Jak, JSE, DynJava, Jumbo, MAJ, and BSJ are all Java extensions with quasi-quote constructs to support creating and manipulating ASTs. JSE offers an extra pattern-matching construct based on Dylan's rewrite rules, while MAJ quasi-quotes express AspectJ code instead of Java. Scala macros also rely on AST creation and manipulation and provide quasi-quotes as a convenient syntax for expressing ASTs. Finally, Mython extends Python with an extensible quotation syntax and Fan extends OCaml with quotation syntax to support DDSLs.

Another typical case of metalanguages that extend their object language are two-stage and multistage languages, where object-level and metalevel syntax are separated through staging annotations. MetaML and MetaOCaml extend ML and OCaml with the basic staging elements: brackets, escape, and run. Similarly, Metaphor and Mint respectively extend C# and Java with brackets and escape, while the run construct is offered as a method available on code objects. Template Haskell extends Haskell with quasi-quote and splice operators, while Converge adds similar extensions for its Python-like object language. Metalua extends Lua with operators for shifting across metalevels, while Delta extends its object-language syntax with the three basic MSP constructs along with extra operators for expressing metalevel statements and definitions.

Object language extensions may also take the form of using additional syntax to specify how metalevel code should be deployed. For example, the reflective systems Reflective Java and R-Java both extend Java with extra keywords used to specify the reflective classes. Also, many systems featuring MOPs have the same syntax for metalevel and base-level classes and objects and use extra syntax only for their associations. OpenC++, Iguana, OpenJava, and Iguana/J all use C++ and Java code to express metalevel and base-level functionality while offering extra syntax to associate classes with metaclasses. In DeepJava, base-level and metalevel objects are distinguished by their potency values, while additional syntax is used for dynamic class creation. Maya extends Java with syntax for introducing and specifying syntax transformations. Furthermore, Handi-Wrap relies on Maya to extend Java with dynamic method wrappers used for AOP. Finally, MetaFJig extends Java with a set of primitive class composition operators.

5.3 Metalanguage Different from the Object Language

Using a different language for metaprogramming targets minimizing the gap between metalanguage concepts and their realization (i.e., implementation in metalanguage syntax). The metalanguage syntax and constructs are selected to better reflect the metalanguage concepts to ease their use in developing metaprograms and enable them to become more concise and understandable.

However, separating the metalanguage from the object language may lead to different development practices and disable the potential for design or code reuse between them, significantly affecting the software engineering process. From a usability perspective, it also requires learning and being proficient in two languages instead of one.

A case exemplifying how radically different *a custom metalanguage* may be compared to the object language is C++. The object language is a compiled imperative language with object-oriented features, while the metalanguage is an interpreted purely functional language. Thus, C++ templates require an entirely different programming style and involve custom coding practices that deviate from common programming styles. For instance, a loop requires advanced features like recursive and partial template specializations. Other systems that use a custom metalanguage include systems featuring macros, static AOP, and generative programming.

Lexical macro systems such as CPP and M4 are ignorant of the object language and naturally use custom syntax to specify the macro transformation logic. However, many syntactic macros also use custom metalanguages. Scheme syntax-rules macros and their derivatives (e.g., ExJs and sweet.js *rule* macros) use a declarative pattern-based language that is separate from the main language. Dylan's macro system is a pattern-matching template substitution system that differs from typical Dylan syntax. <bigwig> macros rely on declarative concepts while targeting a C/Java-like object language. Finally, Camlp4 uses custom syntax to operate on OCaml grammars.

Static AOP systems such as AspectJ and AspectC++ also use custom languages for specifying AOP constructs. Aspect definitions are similar to class definitions and advice resembles method definitions, but pointcuts involve custom syntax with new keywords that constitutes a special-purpose pattern-matching language. Aspect.NET also introduces a custom metalanguage (Aspect.Net.ML) for AOP functionality. It can be used across .Net languages and the Aspect.NET preprocessor converts metalanguage annotations to .Net Attributes for the object language.

Various generative programming systems also adopt custom languages for metaprogramming. Some of them introduce new language constructs not present in the object language. For example, in SafeGen, the metalanguage consists mainly of the cursors and generators constructs that respectively iterate and match over input program entities and output generated code. Similarly, CTR relies on the *transform* construct to write metacode for inspection and object code generation. Moreover, MTJ and PTFJ use traits and trait functions that allow parameterizing over names and types. Other metalanguages select constructs and syntax that *resemble object language syntax*. This aims to blur the distinction between the object level and the metalevel and enable the programmer to think in terms of an extended object language rather than two separate languages. For example, cJ offers a single static conditional construct whose syntax resembles Java generic types and wildcards. Genoupe and MorphJ offer static conditional and iteration constructs matching those of their object languages. Genoupe is closer to its object language, as its conditionals resemble C# expressions, while MorphJ deviates from Java syntax as it uses nested patterns that act as existential conditions.

Finally, there are cases where a metalanguage supports *multiple object languages* by design, so naturally it is different from the object languages. Stratego/XT is a representative example as it supports generic program transformation. Marco and MetaHaskell also fall into this category as they enable writing metaprograms for a variety of object languages.

Table 7 classifies metalanguages that are different from their object languages based on the subcategories of the "Relation to the Object language" dimension of the taxonomy.

6 DISCUSSION

We have presented a wide range of metaprogramming languages and systems that adopt various metaprogramming models, support several possible evaluation phases and metaprogram source

Table 7. Classifying Metalanguages Different from Their Object Languages against the Taxonomy

Classification Dimensions \ Metalanguages Different from Their Object Languages	<bigwig>	AspectJ	Aspect.Net	AspectC++	C++	cJ	CPP	CTR	Camlp4	Dylan	ExJs	Genoupe	Groovy AOP	M4	MTJ	Marco	MetaHaskell	MorphJ	PTFJ	SafeGen	Scheme	Stratego/XT	SugarJ	sweet.js	Terra
Custom metalanguage	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓				✓	✓	✓		✓	✓	✓
Resembling object language						✓						✓						✓							
Multiple object languages																✓	✓					✓			

locations, and have varying relations to the object language. Table 8 summarizes the language and system classification based on our taxonomy.

Languages and systems are equally distributed across metaprogramming models, indicating that all models have received significant research focus, and highlighting the diversity of approaches within the field of metaprogramming. Most languages also adopt a single model for their metaprogramming needs. Here we note that we mostly focus on models inherently supported within the language and not the ones offered as language extensions or libraries. For example, macro systems can be adopted for generative programming, while MOPs can be used as the basis for AOP libraries. Additionally, popular languages like Java have extensions and libraries supporting all models (e.g., Maya offers macros, MetaXa offers a MOP, and MorphJ and Mint support, respectively, generative and multistage programming). In any case, we categorize based on the base language features and classify extensions or libraries separately.

As previously noted, there is a strong correlation between the metaprogramming model and the phase of the metaprogram evaluation. All macro, static AOP, and generative programming systems can operate as a preprocessing step or during compilation. On the contrary, most reflection systems, MOPs, MSP systems, and dynamic AOP systems are evaluated during execution. Additionally, almost all systems evaluate their metaprograms during a single phase. The few exceptions that support more than one phases involve languages and systems offering multiple metaprogramming models, with each phase supported to match a different model.

Focusing on specific models, most macro systems are syntactic, procedural, and hygienic; this certifies the insufficiency of lexical macros, highlights the need for programmatic generation or transformation, and validates the advantage of having hygienic macro expansion. AOP systems are balanced across static and dynamic ones, matching the need for efficiency or flexibility based on the use case. Most generative programming systems feature AST transformation and compile-time reflection, while the majority of them support code templates and quasi-quotes. Most MSP systems are homogeneous and multistage. Two-stage homogeneous languages exist but do not seem to have some theoretical barrier for imposing the stage limit, rather implementation issues. Finally, two-stage heterogeneous languages also exist, with valid use cases.

In terms of source location, most languages featuring macros, generative programming, or multistage programming support context-unaware source code generation, while reflection systems, MOPs, and AOP globally affect all program code. Put another way, most CTMP systems offer context-unaware generation, while most RTMP systems offer global generation or transformation. There are only a few systems supporting context-aware source code generation and transformation, and even fewer systems where metaprograms are applied externally (along with AOP systems that can also operate as such). The latter shows a clear tendency to have metaprograms collocated in the object program they intend to operate on.

Regarding the relation between object language and metalanguage, there is a balance across the viable options. There are various cases where a single language plays both the role of object

Table 8. Classification of Metaprogramming Languages and Systems Based on the Taxonomy

Classification Dimensions / Metaprogramming Languages and Systems	*Metaprogramming Model*						*Evaluation Time*			*Source Location*				*Relation to Object Language*		
	Macro System	Reflection System	MOP	AOP	Generative Programming	Multistage Programming	PPTMP	CTMP	RTMP	Context Unaware	Context Aware	Affects All Source	Applied Externally	Indistinguishable	Extension	Different
CPP, M4, Camlp4, Marco	✓						✓			✓						✓
Common Lisp	✓	✓	✓					✓	✓	✓		✓		✓		
Scheme	✓	✓						✓	✓	✓		✓		✓		✓
Racket	✓	✓					✓	✓	✓	✓		✓		✓		✓
MS^2, JSE, Maya	✓							✓		✓					✓	
<bigwig>	✓							✓		✓						✓
Dylan	✓		✓					✓		✓		✓				✓
ZL, Honu, Scala, Gestalt	✓							✓		✓				✓		
sweet.js, ExJs	✓						✓			✓				✓		✓
Nemerle, Inverse Macros	✓							✓		✓	✓			✓		
Java		✓			✓			✓	✓		✓	✓		✓		
C#, Javaassist, BCEL, Self, Dart		✓							✓			✓		✓		
'C, DynJava, Jumbo		✓				✓			✓	✓					✓	
Python		✓	✓						✓	✓	✓	✓		✓		
Ruby, Javascript		✓	✓						✓	✓		✓		✓		
Mnemonics		✓							✓	✓				✓		
Iguana, IguanaJ, R-Java, Green			✓						✓			✓			✓	
Guaraná, Kava, MetaJ, MetaXa, Oberon, Smalltalk			✓						✓			✓		✓		
Reflex			✓	✓					✓			✓		✓		
Reflective Java			✓				✓					✓			✓	
OpenC++, OpenJava			✓				✓	✓				✓			✓	
Jasper, DeepJava			✓					✓				✓			✓	
AspectJ				✓			✓	✓	✓			✓	✓			✓
AspectWerkz, AspectL, AspectR, JAC, AspectS, Spring, PROSE, Handi-Wrap				✓					✓			✓	✓	✓		
AspectC++				✓			✓					✓	✓			✓
Groovy AOP				✓					✓			✓	✓			✓
Aspect.Net				✓			✓	✓				✓				✓
AspectScheme, PostSharp, AOP++				✓				✓				✓		✓		
C++					✓			✓		✓				✓		✓
D					✓			✓		✓				✓		
JTS, Fan, MAJ					✓		✓			✓					✓	
Stratego/XT					✓		✓						✓			✓
SugarJ					✓		✓			✓			✓			✓
Groovy			✓		✓			✓	✓		✓		✓	✓		
Lombok, PPX					✓			✓			✓			✓		
SafeGen		✓			✓		✓						✓			✓
cJ, MTJ, PTFJ					✓		✓			✓						✓
Genoupe		✓			✓		✓			✓						✓
CTR, MorphJ		✓			✓			✓		✓						✓
Mython					✓	✓		✓		✓					✓	
MetaFJig					✓	✓		✓		✓					✓	
BSJ					✓			✓			✓				✓	
FFJ					✓			✓		✓						✓
MetaML, MetaOCaml, Metaphor, Mint, Squid, $\lambda_1^{\emptyset}$, $\lambda^{[]}$						✓			✓	✓					✓	
MacroML, Modular Macros	✓					✓	✓			✓					✓	
Dotty Scala Compiler	✓					✓		✓	✓	✓					✓	
Template Haskell, Converge, Metalua						✓		✓		✓					✓	
MetaHaskell, Terra						✓				✓						✓
Delta		✓		✓		✓	✓	✓	✓	✓	✓		✓		✓	
LMS						✓			✓	✓				✓		

language and metalanguage, but also many other cases where the metalanguage is an extension of the object language or a new language altogether. Custom metalanguages involve fewer constructs than their object languages and provide specific code transformation or generation features, typically offering a limited form of expressiveness. When more expressiveness is required, the metalanguage is typically an extension of the object language or the object language itself. For example, Scheme syntax-rules macros constitute a separate language that is capable of only pattern-based substitutions, while syntax-case macros use the full language to enable more powerful transformations. Nevertheless, there are custom metalanguages that are Turing Complete (e.g., C++ templates, Dylan macros, and the Marco macro system). Finally, the relation to the object language does not have a strong correlation to the metaprogramming model, as there are examples of all categories for almost all models. MSLs are an exception to this, as the base languages are extended with staging annotations to deliver MSP functionality.

7 CONCLUSIONS

Metaprogramming has been, is, and will continue to be an important research topic as it constitutes a flexible and powerful reuse solution for the ever-growing size and complexity of software systems. Its adoption throughout the years, the constantly increasing support offered by programming languages, and the amount of metacode growing exponentially in recent years are clear indications for what lies ahead.

In this article, we introduced a taxonomy of metaprogramming languages and systems and presented a survey of metaprogramming languages based on the taxonomy. Our main classification involved the metaprogramming model adopted by the language consisting of widely adopted practices for metaprogramming: macro systems, reflection and dynamic code generation, MOPs, AOP, generative programming, and MSP. We described each model, discussed the relations and connections with other models, and presented various metaprogramming languages and systems, classifying them based on the most fitting model(s).

Then we classified metaprograms by their phase of evaluation, distinguishing three possible options: preprocessing-time evaluation, compilation-time evaluation, and execution-time evaluation. We examined each option, discussing advantages, disadvantages, and requirements imposed on the translation or execution system, and explained the categorization for every discussed metaprogramming language and system. We also highlighted the correlation between the metaprogramming model of the language and the phase of the metaprogram evaluation.

Another classification category we presented was based on the metaprogram source location, distinguishing metaprograms embedded within the program they operate on and metaprograms located externally from their subject program. For the former, we further identified three possible options for transforming the target program: (1) context unaware, that is, replacing only the metacode invocation with the generated code; (2) context aware, that is, affecting the metacode invocation and code fragments related to that context; and (3) global, that is, affecting the entire program regardless of the metaprogram location.

Finally, we classified metaprogramming languages based on the relation between the metalanguage and the object language, separating them into three categories in which the metalanguage is (1) identical to the object language, (2) an extension of the object language, and (3) an entirely different language.

Overall, our work aims to put the metaprogramming landscape in order and hopefully become a stepping stone for future research efforts in the field of metaprogramming.

REFERENCES

Andrei Alexandrescu. 2010. *The D Programming Language*. Addison-Wesley Professional.

Nada Amin and Tiark Rompf. 2017. Collapsing towers of interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (December 2017), 33 pages. DOI: https://doi.org/10.1145/3158140

Apache Commons. 2006. BCEL - Byte code engineering library. Retrieved January 2019 from https://commons.apache.org/proper/commons-bcel/.

Sven Apel, Christian Kästner, and Christian Lengauer. 2008. Feature featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*. ACM, New York, 101–112. DOI: https://doi.org/10.1145/1449913.1449931

Kenichi Asai. 2014. Compiling a reflective language using MetaOCaml. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE'14)*. ACM, New York, 113–122. DOI: http://dx.doi.org/10.1145/2658761.2658775.

Kevin Atkinson and Matthew Flatt. 2011. Adapting scheme-like macros to a c-like language. In *Workshop on Scheme and Functional Programming (Scheme'11)*. Retrieved January 2019 from http://scheme2011.ucombinator.org/papers/Atkinson2011.pdf.

Jonathan Bachrach and Keith Playford. 1999. *D-expressions: Lisp power, Dylan style*. Technical Report. Retrieved January 2019 from https://people.csail.mit.edu/jrb/Projects/dexprs.pdf.

Jonthan Bachrach and Keith Playford. 2001. The Java syntactic extender (JSE). In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. ACM, New York, 31–42. DOI: http://doi.acm.org/10.1145/504282.504285

Jason Baker and Wilson Hsieh. 2002a. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM, New York, 86–95. DOI: http://doi.acm.org/10.1145/508386.508396

Jason Baker and Wilson Hsieh. 2002b. Maya: Multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*. ACM, New York, 270–281. DOI: http://doi.acm.org/10.1145/512529.512562

Don Batory, Bernie Lofaso, and Yannis Smaragdakis. 1998. JTS: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*. IEEE Computer Society, Washington, DC, 143–153. DOI: http://dx.doi.org/10.1109/ICSR.1998.685739

Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 88–99. University of Aarhus. Invited talk. Retrieved January 2019 from https://3e8.org/pub/scheme/doc/Quasiquotation%20in%20Lisp%20(Bawden).pdf.

Alan Bawden and Jonathan Rees. 1988. Syntactic closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP'88)*, 86–95. DOI: http://doi.acm.org/10.1145/62678.62687

Jonas Bonér. 2004. What are the key issues for commercial AOP use: How does AspectWerkz address them? In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*. ACM, New York, 5–6. DOI: http://dx.doi.org/10.1145/976270.976273

Claus Brabrand and Michael I. Schwartzbach. 2002. Growing languages with metamorphic syntax macros. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*. ACM, 31–40. DOI: http://doi.acm.org/10.1145/503032.503035

Gilad Bracha and David Ungar. 2004. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming*, Systems, Languages, and Applications *(OOPSLA'04)*. ACM, New York, 331–344. DOI: http://dx.doi.org/10.1145/1028976.1029004

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1–2 (2008), 52–70. DOI: http://dx.doi.org/10.1016/j.scico.2007.11.003

Avi Bryant and Robert Feldt. 2002. AspectR - Simple aspect-oriented programming in Ruby. Retrieved January 2019 from http://aspectr.sourceforge.net/.

Eugene Burmako. 2013. 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)*. ACM, New York, Article 3, 1–10. DOI: http://doi.acm.org/10.1145/2489837.2489840.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE'03). Springer LNCS* 2830, 57–76. DOI: http://dx.doi.org/10.1007/978-3-540-39815-8_4

Guido Chari, Diego Garbervetsky, and Stefan Marr. 2016. Building efficient and highly run-time adaptable virtual machines. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*. ACM, New York, 60–71. DOI: https://doi.org/10.1145/2989225.2989234

Shigeru Chiba. 1995. A metaobject protocol for C++. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. ACM, New York, 285–299. DOI: http://doi.acm.org/10.1145/217838.217868

Shigeru Chiba. 1998. Javassist - A reflection-based programming wizard for java. In *Proceedings of the OOPSLA'98 Workshop on Reflective Programming in C++ and Java*. Retrieved January 2019 from http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=2404E80FF2C51CC1892AD4539970A569?doi=10.1.1.39.7059&rep=rep1&type=pdf.

William Clinger and Jonathan Rees. 1991. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'91)*. ACM, New York, 155–162. DOI: http://doi.acm.org/10.1145/99583.99607

Pascal Costanza. 2004. A short overview of AspectL. In *Proceedings of the European Interactive Workshop on Aspects in Software*. EIWAS. Retrieved January 2019 from http://www.p-cos.net/documents/aspectl-short-final.pdf.

Ryan Culpepper and Matthias Felleisen. 2010. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*. ACM, New York, 235–246. DOI: https://doi.org/10.1145/1863543.1863577

Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing, New York.

Robertas Damaševičius and Vytautas Štuikys. 2008. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37, 2 (2008), 124–132. Retrieved January 2019 from http://www.itc.ktu.lt/index.php/ITC/article/view/11931/6598.

Joe Darcy. 2006. Java Specification Request 269: Pluggable annotation processing API. Retrieved January 2019 from http://jcp.org/en/jsr/detail?id=269.

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, 105–116. DOI: https://doi.org/10.1145/2491956.2462166

Tim Disney, Nathan Faubion, David Herman, and Cormac Flanagan. 2014. Sweeten your JavaScript: Hygienic macros for ES5. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS'14)*. ACM, New York, 35–44. DOI: https://doi.org/10.1145/2661088.2661097

Rémi Douence and Mario Südholt. 2001. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation* 14, 1 (2001), 7–34. DOI: http://dx.doi.org/10.1023/A:1011549115358

Dirk Draheim, Christof Lutteroth, and Gerald Weber. 2005. A type system for reflective program generators. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05). Springer LNCS* 3676, 327–341. DOI: http://dx.doi.org/10.1007/11561347_22

Stéphane Ducasse, Nathanael Schärli, and Roel Wuyts. 2005. Uniform and safe metaclass composition. *Computer Languages Systems and Structures* 31, 3–4 (2005), 143–164. DOI: http://dx.doi.org/10.1016/j.cl.2004.11.003

Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. 2006. Semantics and scoping of aspects in higher-order languages. *Science Computer Programming* 63, 3 (2006), 207–239. DOI: http://dx.doi.org/10.1016/j.scico.2006.01.003

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1993. Syntactic abstraction in scheme. *Lisp and Symbolic Computation* 5, 4 (1993), 295–326. DOI: http://dx.doi.org/10.1007/BF01806308

R. Kent Dybvig. 2009. *The Scheme Programming Language* (4th ed.). MIT Press.

Jason Eckhardt, Roumen Kaiabachev, Emir Pašalić, Kedar Swadi, and Walid Taha. 2005. Implicitly heterogeneous multi-stage programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, Springer LNCS 3676, 275–292. DOI: http://dx.doi.org/10.1007/11561347_19

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based syntactic language extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. ACM, New York, 391–406. DOI: https://doi.org/10.1145/2048066.2048099

Manuel Fähndrich, Michael Carbin, and James R. Larus. 2006. Reflective program generation with patterns. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*. 275–284. DOI: http://doi.acm.org/10.1145/1173706.1173748

Matthew Flatt. 2002. Composable and compilable macros: You want it when? In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*. ACM, New York, 72–83. DOI: http://doi.acm.org/10.1145/581478.581486

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming* 22, 2 (2012), 181–216. DOI: http://dx.doi.org/10.1017/S0956796812000093

Fabien Fleutot. 2007. Metalua Manual. Retrieved January 2019 from http://metalua.luaforge.net/metalua-manual.html.

Gael Fraiteur. 2008. AOP on .NET – PostSharp. Retrieved April 1, 2015, from https://www.postsharp.net/aop.net.

Steven E. Ganz, Amr Sabry, and Walid Taha. 2001. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM, New York, 74–85. DOI: http://doi.acm.org/10.1145/507635.507646

Robert Glück and Jesper Jørgensen. 1996. Fast binding-time analysis for multi-level specialization. In *Proceedings of the 2nd International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer LNCS 1181, 261–272. DOI: http://dx.doi.org/10.1007/3-540-62064-8_22

Michael Golm and Jürgen Kleinöder. 1999. Jumping to the meta level: Behavioral reflection can be fast and flexible. In *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99)*. Springer LNCS 1616, 22–33. DOI: http://dx.doi.org/10.1007/3-540-48443-4_3

Brendan Gowing and Vinny Cahill. 1996. *Meta-Object Protocols for C++: The Iguana Approach*. Technical Report. University of Bologna. Retrieved January 2019 from https://www.scss.tcd.ie/publications/tech-reports/reports.96/TCD-CS-96-16.pdf.

José de Oliveira Guimarães. 1998. Reflection for statically typed languages. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Springer LNCS 1445, 440–461. DOI: http://dx.doi.org/10.1007/BFb0054103

Tero Hasu and Matthew Flatt. 2016. Source-to-source compilation via submodules. In *Proceedings of the 9th European Lisp Symposium on European Lisp Symposium (ELS'16)*, Article 7, 8.

Robert Hirschfeld. 2002. AspectS - aspect-oriented programming with squeak. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe'02)*. *Springer LNCS* 2591, 216–232. DOI: http://dx.doi.org/10.1007/3-540-36557-5_17

Markus Hof. 2000. Composable message semantics in Oberon. In *Modular Programming Languages (JMLC'00)*. Springer LNCS 1897, 11–25. DOI: https://doi.org/10.1007/10722581_2

Zhang Hongbo and Steve Zdancewic. 2013. Fan: Compile-time metaprogramming for OCaml. Retrieved January 2019 from http://zhanghongbo.me/fan/_downloads/metaprogramming_for_ocaml.pdf.

Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2005. Statically safe program generation with SafeGen. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*. *Springer LNCS* 3676, 309–326. DOI: http://dx.doi.org/10.1007/11561347_21

Shan Shan Huang, David Zook, and Yannis Smaragdakis. 2007. cJ: Enhancing Java with safe type conditions. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development (AOSD'07)*. ACM, New York, 185–198. DOI: http://dx.doi.org/10.1145/1218563.1218584

Shan Shan Huang and Yannis Smaragdakis. 2011. Morphing: Structurally shaping a class by reflecting on others. *ACM Transactions on Programming Languages and Systems* 33, 2, Article 6, 44. DOI: http://doi.acm.org/10.1145/1890028.1890029

Rod Johnson. 2011. Aspect oriented programming with spring. Retrieved January 2019 from http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html.

Chanwit Kaewkasi and John R. Gurd. 2008. Groovy AOP: A dynamic AOP system for a JVM-based language. In *Proceedings of the 2008 AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'08)*. ACM, Article 3. DOI: http://doi.acm.org/10.1145/1408647.1408650

Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the stage: Staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, 111–120. DOI: http://doi.acm.org/10.1145/1480945.1480962

Sam Kamin, Lars Clausen, and Ava Jarvis. 2003. Jumbo: Run-time code generation for Java and its applications. In *Proceedings of the 1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO'03)*, 48–56. DOI: http://doi.ieeecomputersociety.org/10.1109/CGO.2003.1191532

Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language* (2nd ed.). Prentice-Hall, Englewood Cliffs, NJ.

Gregor Kiczales, Jim Rivieres, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Springer LNCS 1241, 220–242. DOI: http://dx.doi.org/10.1007/BFb0053381

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*. Springer LNCS 2072, 327–354. DOI: http://dx.doi.org/10.1007/3-540-45337-7_18

Michael Kimberlin. 2010. Reducing boilerplate code with project Lombok. Retrieved January 2019 from http://jnb.ociweb.com/jnb/jnbJan2010.html.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP'86)*. ACM, New York, 151–161. DOI: http://doi.acm.org/10.1145/319838.319859

Thomas Kühne and Daniel Schreiber. 2007. Can programming be liberated from the two-level style: Multi-level programming with deepjava. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. ACM, New York, 229–244. DOI: https://doi.org/10.1145/1297027.1297044

Byeongcheol Lee, Robert Grimm, Martin Hirzel, and Kathryn S. McKinley. 2012. Marco: Safe, expressive macros for any language. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12). Springer LNCS* 7313, 589–613. DOI: http://dx.doi.org/10.1007/978-3-642-31057-7_26

Yannis Lilis and Anthony Savidis. 2014. Aspects for stages: Cross cutting concerns for metaprograms. *Journal of Object Technology* 13, 1 (2014), 1–36. DOI: http://dx.doi.org/10.5381/jot.2014.13.1.a1

Yannis Lilis and Anthony Savidis. 2015. An integrated implementation framework for compile-time metaprogramming. *Software Practice and Experience* 45, 6 (2015), 727–763. DOI: http://dx.doi.org/10.1002/spe.2241

Fengyun Liu and Eugene Burmako. 2017. *Two approaches to portable macros.* Technical Report, EPFL.

Pattie Maes. 1987. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87).* ACM, New York, 147–155. DOI: http://dx.doi.org/10.1145/38765.38821

Geoffrey Mainland. 2012. Explicitly heterogeneous metaprogramming with metahaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP'12).* ACM, New York, 311–322. DOI: http://doi.acm.org/10.1145/2364527.2364572

Stefan Marr, Chris Seaton, and Stéphane Ducasse. 2015. Zero-overhead metaprogramming: reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).* ACM, New York, 545–554. DOI: https://doi.org/10.1145/2737924.2737963

Weiyu Miao and Jeremy Siek. 2012. Pattern-based traits. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12).* ACM, New York, 1729–1736. DOI: http://doi.acm.org/10.1145/2245276.2232057

Weiyu Miao and Jeremy Siek. 2014. Compile-time reflection and metaprogramming for Java. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14).* ACM, New York, 27–37. DOI: https://doi.org/10.1145/2543728.2543739

Gregory Neverov and Paul Roe. 2006. Experiences with an object-oriented, multi-stage language. *Science of Computer Programming* 62, 1 (2006), 85–94, DOI: http://dx.doi.org/10.1016/j.scico.2006.05.002

Angela Nicoara and Gustavo Alonso. 2005. Dynamic AOP with PROSE. In *Proceedings of the International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05).* Retrieved January 2019 from http://www.deutsche-telekom-laboratories.com/~angela/papers/AngelaNicoara-ASMEA2005.pdf.

Dmitry Nizhegorodov. 2000. Jasper: Type-Safe MOP-based language extensions and reflective template processing in java. In *Proceedings of ECOOP 2000 Workshop on Reflection and Metalevel Architectures: State of the Art, and Future Trends.* ACM Press.

Yutaka Oiwa, Hidehiko Masuhara, and Akinori Yonezawa. 2001. DynJava: Type safe dynamic code generation in java. In *Proceedings of the 3rd JSSST Workshop on Programming and Programming Languages (PPL'01).* Retrieved January 2019 from http://loome.cs.illinois.edu/CS498F10/readings/dynjava.pdf.

Alexandre Oliva and Luiz Eduardo Buzato. 1999. The design and implementation of Guaraná. In *Proceedings of the 5th Conference on USENIX Conference on Object-Oriented Technologies & Systems – Vol. 5 (COOTS'99).* USENIX Association, Berkeley, CA, 1–15. Retrieved January 2019 from https://www.usenix.org/legacy/events/coots99/full_papers/oliva/oliva.pdf.

Zachary Palmer and Scott F. Smith. 2011. Backstage Java: Making a difference in metaprogramming. In *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'11).* ACM, New York, 939–958. DOI: https://doi.org/10.1145/2048066.2048137

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017. Unifying analytic and statically-typed quasiquotes. *Proceedings of ACM Programming Languages* 2, POPL, Article 13, 33 pages. DOI: https://doi.org/10.1145/3158101

Emir Pasalic. 2004. *The Role of Type Equality in Meta-Programming.* Ph.D. Dissertation. OGI School of Science & Engineering. Advisor(s) Timothy E. Sheard. AAI3151199. Retrieved January 2019 from http://web.cecs.pdx.edu/~sheard/papers/EmirsThesis.pdf.

Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. 2001. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION'01). Springer LNCS* 2192, 1–24. DOI: http://dx.doi.org/10.1007/3-540-45429-2_1

Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 1999. C and TCC: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 2 (1999), 324–369. DOI: http://doi.acm.org/10.1145/316686.316697

Jon Rafkind and Matthew Flatt. 2012. Honu: Syntactic extension for algebraic notation through enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering (GPCE'12).* ACM, 122–131. DOI: http://doi.acm.org/10.1145/2371401.2371420

Daniel de Rauglaudre. 2003. Camlp4 – Tutorial. Retrieved January 2019 from http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html.

Barry Redmond and Vinny Cahill. 2002. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02). Springer LNCS* 2374, 205–230. DOI: http://dx.doi.org/10.1007/3-540-47993-7_9

John Reppy and Aaron Turon. 2007. Metaprogramming with traits. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07). Springer LNCS* 4609, 373–398. DOI: http://dx.doi.org/10.1007/978-3-540-73589-2_18

Morten Rhiger. 2012. Staged computation with staged lexical scope. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'12). Springer LNCS* 7211, 559–578. DOI: http://dx.doi.org/10.1007/978-3-642-28869-2_28

Jonathan Riehl. 2009. Language embedding and optimization in Mython. In *Proceedings of the 5th Symposium on Dynamic Languages (DLS'09)*. 39–48. DOI: http://doi.acm.org/10.1145/1640134.1640141

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*. ACM, New York, 127–136. DOI: http://doi.acm.org/10.1145/1868294.1868314

Johannes Rudolph and Peter Thiemann. 2010. Mnemonics: Type-safe bytecode generation at run time. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*. ACM, New York, 15–24. DOI: http://doi.acm.org/10.1145/1706356.1706361

Vladimir O. Safonov and Dmitry A. Grigoriev. 2005. Aspect.NET – An aspect-oriented programming tool for Microsoft.NET. In *Proceedings of IEEE Regional Conference 2005*. Retrieved January 2019 from https://sites.google.com/site/aspectdotnet/5.pdf.

Marco Servetto and Elena Zucca. 2010. MetaFJig: A meta-circular composition language for Java-like classes. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*. 464–483. DOI: http://doi.acm.org/10.1145/1869459.1869498

Marco Servetto and Elena Zucca. 2014. A meta-circular language for active libraries. *Science of Computer Programming* 95, P2 (2014), 219–253. DOI: http://dx.doi.org/10.1016/j.scico.2014.05.003

Tim Sheard. 2001. Accomplishments and research challenges in metaprogramming. In *Proceedings of the 2nd International Workshop on Semantics, Application and Implementation of Program Generation (SAIG'01)*. Springer LNCS 2196, 2–44. DOI: http://dx.doi.org/10.1007/3-540-44806-3_2

Tim Sheard and Simon Peyton Jones. 2002. Template metaprogramming for Haskell. *SIGPLAN Notices* 37, 12 (December 2002), 60–75. DOI: http://dx.doi.org/10.1145/636517.636528.

Kamil Skalski, Michal Moskal, and Pawel Olszta. 2004. Metaprogramming in Nemerle. Retrieved January 2019 from http://pdfs.semanticscholar.org/c3b2/e92909de69b162c064a211a2c56947542373.pdf.

Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis. 2017. Structured program generation techniques. In *Grand Timely Topics in Software Engineering (GTTSE'15)*. Lecture Notes in Computer Science, 10223. Springer. DOI: https://doi.org/10.1007/978-3-319-60074-1_7

Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. 2002. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT'02)*, 53–60.

Venkat Subramaniam. 2013. Programming groovy 2: Dynamic productivity for the Java developer. *Pragmatic Bookshelf* (1st edition).

Guy L. Steele. 1990. *Common Lisp: The Language* (2nd ed.). Digital Press.

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'18)*. ACM, New York, 14–27. DOI: https://doi.org/10.1145/3278122.3278139

Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'97)*, ACM, New York, 203–217. DOI: http://doi.acm.org/10.1145/258994.259019

Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*. ACM, New York, 26–37. DOI: http://dx.doi.org/10.1145/604131.604134

Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. 2003. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*. ACM, New York, 27–46. DOI: http://doi.acm.org/10.1145/949305.949309

Éric Tanter. 2004. *Reflection and Open Implementations*. Technical report, University of Chile.

Éric Tanter, Rodolfo Toledo, Guillaume Pothier, and Jacques Noyé. 2008. Flexible metaprogramming and AOP in Java. *Science of Computer Programming* 72, 1–2 (2008), 22–30. DOI: http://dx.doi.org/10.1016/j.scico.2007.10.005

Michiaki Tatsubori, Shigeru Chiba, Marc-Olivier Killijian, and Kozo Itano. 2000. OpenJava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering*. Springer LNCS 1826, 117–133. DOI: http://dx.doi.org/10.1007/3-540-45046-7_7

Elisa Tomioka, José de Oliveira Guimarães, and Antonio Francisco do Prado. 1998. R-Java, A Reflective Java Extension. Retrieved January 2019 from http://www.cyan-lang.org/jose/green/rjava/rjava-sbc.pdf.

Laurence Tratt. 2008. Domain specific language implementation via compile-time metaprogramming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 6 (2008), 1–40. DOI: http://doi.acm.org/10.1145/1391956.1391958

Kenneth J. Turner. 1994. *Exploiting the m4 macro language.* Technical Report CSM-126, Department of Computing Science and Mathematics, University of Stirling, Scotland. Retrieved January 2019 from http://www.cs.stir.ac.uk/~kjt/research/pdf/expl-m4.pdf.

Todd Veldhuizen. 2003. C++ templates are Turing complete. *Technical Report, Indiana University.* Retrieved January 2019 from http://port70.net/~nsz/c/c%2B%2B/turing.pdf.

Marcos Viera and Alberto Pardo. 2006. A multi-stage language with intensional analysis. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06).* ACM, New York, 11–20. DOI: http://doi.acm.org/10.1145/1173706.1173709

Ken Wakita, Kanako Homizu, and Akira Sasaki. 2014. Hygienic macro system for Javascript and its light-weight implementation framework. In *Proceedings of ILC 2014 on 8th International Lisp Conference (ILC'14).* ACM, New York, 12–21. DOI: http://doi.acm.org/10.1145/2635648.2635653

Daniel Weise and Roger Crew. 1993. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI'93).* ACM, New York, 156–165, 1993. DOI: http://doi.acm.org/10.1145/155090.155105

Ian Welch and Robert Stroud. 1999. From Dalang to Kava - the evolution of a reflective java extension. In *Proceedings of the 2nd International Conference on Reflection (Reflection'99).* Springer LNCS 1616, 2–21. DOI: http://dx.doi.org/10.1007/3-540-48443-4_2

Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. 2010. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10).* ACM, New York, 400–411. DOI: http://doi.acm.org/10.1145/1806596.1806642

Leo White. 2013. Extension points for ocaml. *OCaml Users and Developers Workshop.*

Zhixue Wu and Scarlet Schwiderski. 1997. Reflective Java: Making Java even more flexible, 456–458. Retrieved January 2019 from https://pdfs.semanticscholar.org/b7df/83217f05f1d2d8be3ec7ecaeda72781dc7e2.pdf.

Jeremy Yallop and Leo White. 2015. Modular macros. Retrieved January 2019 from https://www.cl.cam.ac.uk/~jdy22/papers/modular-macros.pdf.

Hiroshi Yamaguchi and Shigeru Chiba. 2015. Inverse macro in Scala. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'15).* ACM, New York, 85–94. DOI: http://dx.doi.org/10.1145/2814204.2814213

Zhen Yao, Qi-long Zheng, and Guo-liang Chen. 2005. AOP++: A generic aspect-oriented programming framework in C++. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05).* Springer LNCS 3676, 94–108. DOI: http://dx.doi.org/10.1007/11561347_8

David Zook, Shan Shan Huang, and Yannis Smaragdakis. 2004. Generating Aspectj programs with meta-Aspectj. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, 1–18, DOI: http://dx.doi.org/ 10.1007/978-3-540-30175-2_1

Received November 2017; revised June 2019; accepted July 2019