Structured Program Generation Techniques

Yannis Smaragdakis, Aggelos Biboudis, and George Fourtounis

University of Athens
{smaragd,biboudis,gfour}@di.uoa.gr

Abstract. So, you can write a program that generates other programs. Sorry, ... not impressed. You want to impress me? Make sure your program-generating program only produces well-formed programs. What is "well-formed", you ask? Well, let's start with "it parses". Then let's get to "... and type-checks". You want to really impress me? Give me an expressive language for program generators in which any program you write will only generate well-formed programs.
In this briefing, we will sample the state-of-the-art in program generation relative to the above important goal. If we want to establish program generation as a general-purpose, disciplined methodology, instead of an ad hoc hack, we should be able to check the generator once and immediately validate the well-formedness of anything it might generate. This is a modular safety property for meta-programs, much akin to static typing for regular programs.
Some of the emphasis will be on our own work on "class morphing" (or just "morphing"): the statically-safe adaptation of the contents of a class, depending on other classes supplied as parameters. Along the way, lots of other techniques will be discussed and contrasted, from different template facilities, to syntactically-safe program generation, to program staging techniques.

1 Introduction

A *program generator* (or just *generator*) is a program that generates programs expressed in a high-level language. The language in which the generator is written (commonly called the *host* or the *meta* language) and the output language (commonly called the *object* or *target* language) do not have to be the same, although they often are.[1]

Generators arise in so many practical scenarios that one may wonder whether they deserve a special name, or they are merely "programs". Generators appear as wizards or refactorings in IDEs, as template or macro libraries, as implementations (compilers) of domain-specific languages (DSL), as high-performance optimizing libraries, as modularity (e.g., *aspect-oriented*) mechanisms, as frameworks (e.g., for dependency injection), and much more.

[1] A closely related concept is that of a *program transformer*, which modifies an existing program, instead of generating a new one. The main principles and ideas behind generators and transformers are virtually identical. In this text, we write "generator" to mean "generator or transformer".

Generators exist because of the desire, as old as programming itself, to automate, elevate, modularize or otherwise facilitate program development. In practice, generators are one of many technologies for enabling modularity and software reuse—other examples are binary or source libraries, application frameworks, component technologies, and services. However, generators are often the technique of last resort. They are used for programming automation patterns not covered by other, conventional technologies. Generators offer the potential for more advanced optimizations, syntactic convenience, or static checking than plain libraries or component technologies.

Generators are also an intellectually fascinating topic: what can be more interesting to a computer scientist than computing programs? The canonical sensationalist example is self-generating programs. For example, we can have:

```
((lambda (x) (list x (list 'quote x)))
 '(lambda (x) (list x (list 'quote x))))
```

in Lisp or:

```
main(a){a="main(a){a=%c%s%c;printf(a,34,a,34);}";printf(a,34,a,34);}
```

in C.

The power and appeal of generators comes at a cost, however. Programmers often view program generation technology as low-level and largely ad hoc. A common complaint concerns debugging: an error in the generated program can be very hard to debug and may require full understanding of the generator itself. In more general terms, the fault is due to lack of *modular* reasoning. The generator author cannot easily consider what the generator will do for every input, only for the inputs he/she has tested. The generator user (i.e., the end programmer) should not have to reason about the code produced by the generator, only about the way he/she uses it.

This need is the focus of our briefing. We discuss *structured program generation techniques*, i.e., techniques that can offer guarantees on the correctness (w.r.t. static semantics, i.e., at most type-correctness) of generated programs *before these are generated*, i.e., for all inputs to the generator. We will refer to this property as *modular safety* of a generator.[2]

Ensuring that a generator only produces well-formed programs (typically under some assumptions on the generator input) is practically important and intuitively appealing. Viewed as a type-checking matter, this property is quite similar to static typing of regular programs. Much like in standard static typing, we want to statically check the generator and be sure that no type error arises during its *run time*, which happens to be the *compile time* of the generated program.

If the problem of structured program generation is solved, "program generation" will become mere "programming", raising the level of programming automation without sacrificing high-level, modular reasoning. Consider: if a gen-

[2] This is also occasionally called *meta type-safety*.

erator that can do most useful things that current generators do can also be checked modularly, i.e., for all possible inputs, then why does it matter that it is a generator? The output program will never need to be inspected by the end programmer. Instead, we might prefer to mentally model what the generator does as program generation, while in reality we will not care whether a program is actually ever generated.

In the next sections, we present current program generation mechanisms and levels of modular safety, before focusing on state-of-the-art techniques for modular safety of generators. Our goal is not to offer an exhaustive survey of the literature but to inform of different levels of reasoning power, with selected pioneering or representative work in each one.

2 Kinds of Generation and Program Transformation

There is a large variety of mechanisms that can be used to generate or transform programs—for instance, see Reference [12] for a representative comparison. We briefly survey the general classes of such mechanisms, for reference in future sections.

Generation of programs as text. The most basic technique for program generation is that of producing character strings containing the text of a program, which is subsequently interpreted or compiled. For instance, a common approach for generating database queries (in SQL) inside an imperative language program is via string concatenation—for instance:

```
sqlProg = "SELECT name FROM" + tableName + "WHERE id = " + idVar;
```

Note the distinction between target language identifiers (`name`, `id`) and meta-variables (`tableName`, `idVar`). The latter correspond to parts of the generated program text that may vary, depending on the generator's execution. The former are fixed and need to have a meaning in the context of the generated program. (We discuss the topic of what generated names may refer to in the next section, under "Scoping and Hygiene".)

Text-based program generation is readily available in most programming settings, yet clearly low-level. There is nothing in the generator code to indicate that the string that is being assembled represents a program. Therefore, this program could have errors at any level of program processing (lexical analysis, parsing, static semantics, etc.).

Syntax tree manipulation. A more sophisticated, yet commonly used, technique is to generate the syntax tree of a program, instead of its unstructured text. This requires defining host language concepts that correspond to the syntactic structure of the target language—an idea we will revisit in the next section. For example, our SQL-generating program could be written as follows:

```
sqlProg = new SelectStmt(new Column(name), table,
                         new WhereClause(new Column(id), idVar));
```

The generated program can be produced by either pretty-printing the syntax tree and invoking a traditional language processor (e.g., a compiler for the target language) or by interfacing with the post-parsing stage of such a target language processor.

Code templates, quoting. Generating programs by assembling syntax trees can be tedious, even in languages with pattern-matching constructs [54]. Therefore, several facilities for program generation (e.g., [2,35,43,56], among many) offer the ability to generate program fragments by "quoting" the code to be generated, i.e., using code templates with constant and variable parts. This requires language constructs for generating program fragments in the target language (typically called a *quote*—e.g., "`‘[...]`") as well as for supplying values to fill in holes in the generated syntax tree (typically called an *unquote* or *escape*—e.g., "`#[...]`"). For instance, our earlier code fragment might be written as:

```
sqlProg = ‘[SELECT name FROM #[table] WHERE id = #[idVar]];
```

As can be seen, this approach (often termed *meta-programming with concrete syntax* [54]) approximates the syntactic simplicity of a plain evaluated program, significantly simplifying the program text of the generator.

Macros. Another meta-programming approach of widespread use is *macros*: reusable code templates with pre-set rules for parameterizing them and yielding different generated program fragments. A typical macro only allows substitution of parameters in its body, as opposed to more general program generation control flow—e.g., a loop that generates an unbounded number of statements. A reference facility for high-level macros is the Scheme macro system [46]. A swapping macro, replacing its uses by an expression that swaps two values, can be written as follows:

```
(define-syntax-rule (swap x y)
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))
```

Macro languages can vary greatly in sophistication and are difficult to categorize. A common element is that they blend the distinction between generator and generated program. The generated fragment is typically not treated as a data structure (e.g., a syntax tree) but instead replaces program expressions wherever it occurs. Thus, the usual relationship between meta-program and object program is inverted: in macros, the default, undistinguished program text is as-if generated and augmented/transformed by the output of the generator (i.e., the macro), whereas in typical program generation undistinguished program text is part of the generator and generated code is clearly marked.

Generics. Common (type-)genericity mechanisms in programming languages may occasionally be powerful enough to be considered general-purpose program generation facilities. Genericity refers to the ability to parameterize a code template with different static types. Mechanisms such as C++ templates work by producing specialized code for each concrete type parameter. Furthermore, the specialization mechanism is powerful enough to allow conditional reasoning, and templates can be recursive, thus allowing full Turing-complete computation [53]. C++ templates are also explicitly able to compute over compile-time constants, using regular C++ operators. This capability is used to define a compile-time adder in the following code fragment:

```
template<int X, int Y>
struct Adder {
   enum { result = X + Y };
};
```

There are several standard techniques that have harnessed the expressiveness of C++ templates to yield arbitrary program generation capabilities, as discussed in References [12, 52].

Specialized languages. Beyond the above classes of mechanisms, there are several specialized languages for program generators. For example, *aspect-oriented programming* facilities can be viewed as implicit transformations of a program [28, 29]. This is most evident in features such as *inter-type declarations*:

```
aspect S {
  declare parents:
  Car implements Serializable;
}
```

The above aspect adds a supertype (`Serializable`) to an existing class `Car`. As in more overt program generation/transformation techniques, we can ask the question of what guarantees are offered on the generated program, when the aspect is generic and can apply to yet-unspecified classes.

In Section 4, we will see more examples of languages specifically designed for expressing meta-programs.

3 Kinds of Generator Safety

The main focus of this briefing is on *statically-safe* (or just *safe* for brevity) program generation techniques: certifying the generator as "safe" should guarantee the well-formedness of any generated program.

One can perhaps debate whether the static safety of a generator is an essential feature. After all, the generated program will be checked statically before it runs, so why try to catch the same errors before the program is even generated? The answer is that static checking is not mainly intended to detect errors in the generated program or even errors in the generator input, but errors in the generator itself. Such errors are typically mismatched assumptions: the generator fails to take into account some input case, so that, even though the generator writer has tested the generator under several inputs, other inputs result in badly-formed programs. Although these errors will be detected at compile time of the *generated program*, this is (at least as late as) the generator's *run time*. Thus, errors of program well-formedness, which a programmer would hope to have eliminated once and for all, can arise dynamically, as far as the generator is concerned.

Consider a simple scenario in a realistic generator. The generator examines an input program, and for every class containing a designated method—e.g., `register`—produces registration code that invokes the method. The generated code will fail to compile if the `register` method is private. This is an error in the generator itself! The generator writer has failed to take into account the possibility of private `register` methods. (Multiple fixes may be possible: the generator could ignore non-accessible methods, or the generated code could invoke them indirectly—e.g., via reflection.) Even worse, the generator writer could have extensively tested his/her code with large, realistic inputs, just never with private `register` methods.

As discussed in the introduction, tools that only generate well-formed programs are often called *structured* meta-programming tools. The term "structured" only captures the basic premise, however: there are several levels of well-formedness and we need to distinguish them for purposes of precise characterization.

Lexical and syntactic well-formedness. The first level of static safety for generators is safety with respect to lexical and syntax checking. That is, such safety entails employing techniques for building or checking generators so that any generated/transformed program is guaranteed to pass the lexical analysis and parsing phases of a traditional compiler. A common way to satisfy this property is by encoding the syntax of the object language using the type system of the host language. For instance, consider traditional syntax checking expressed as context-free grammar (CFG) rules. Rules for top-level syntactic categories of an imperative language (statement, declaration, expression) will typically take a form such as that below:

```
AST    ::= Stmt
        |  Expr
        |  Decl
Stmt   ::= IfStmt
IfStmt ::= "if" "(" Expr ")" Stmt
...
```

The above CFG specification can map to types for values that the generator manipulates, as well as constructors for these values. This yields a subtyping hierarchy[3] where types *Stmt*, *Expr*, and *Decl* are subtypes of type *AST* and

[3] Alternatively, one can represent a grammar as an *algebraic data type* (ADT), for equivalent functionality, with respect to our static safety guarantees.

type *IfStmt* is a subtype of *Stmt*. Furthermore, values of type *IfStmt* are created using a constructor that accepts an *Expr* value and a *Stmt* value. If the generator type checks, then the values it manipulates are guaranteed to conform to the induced type constraints, which means that they are fragments of syntactically-correct code in the target language, per the CFG rules. There is no possibility of, e.g., creating an *IfStmt* with a *Stmt* instead of an *Expr* in the condition of the generated `if` code fragment.

Scoping and hygiene. Programming languages typically support variables, which obey scoping rules: each variable is first declared and then used, and the rules define where the declaration starts having effect and what variables are visible at each program point. When an *identifier* (i.e., a name) is used to denote a variable, we say that the identifier is a *reference* to the variable, or that the identifier *binds* to the variable declaration.

A correctness property of great interest for generated programs concerns the appropriate binding of identifiers, i.e., ensuring that produced variable references are bound to the *intended* variable declaration. Consider an example of meta-programming with concrete syntax:

```
expr = ‘[ for (int i = 0; i < #[boundExpr]; i++) { #[bodyExpr] } ];
```

The generated code fragment introduces a new identifier, `i`, in a binding position, i.e., in a declaration. Can this declaration bind references that the generator programmer did not intend? For instance, if `boundExpr` holds an expression from the input program, could this expression refer to a variable `i`, bound to the newly declared `i`? Conversely, can declarations of the input program accidentally bind references in the generated code? Mistakes in binding resolution may or may not appear as static checking errors—e.g., binding to an unintended variable may not be a type error, depending on the declared types and the static semantics of the target language. Thus, the problem is a semantic one: even well-typed programs may have a different meaning than what was intended.

The absence of unintended name binding is typically called *hygiene*. Questions of hygiene have been studied in depth, over some-30 years of research in meta-programming. The same issues arise in generics (e.g., C++ templates perform hygienic renaming), in quoting/meta-programming with concrete syntax [45,51], and, most prominently, in macros [10,17,32,50].

Macros have been the first setting where the question of hygiene has been examined [32]. In macros, there is a clearly designated part of the generated program that comes from the input program, and another that comes from the macro definition. Therefore, the issue of hygiene takes a simple form: a macro system is hygienic if the familiar *lexical scoping* rules (i.e., an identifier is bound to a declaration in its lexical context) are obeyed. Hygienic macros are a fundamental feature of Scheme [17,46]. Racket, a Scheme descendant, goes even further by implementing the whole language using code generation via hygienic macros [50].

Hygienic meta-programming systems enforce their hygiene property automatically—typically by performing variable renaming to eliminate ambiguity. Therefore, hygiene is a rather orthogonal property to the rest of the mechanisms we discuss in this briefing. Most of the research in hygienic mechanisms has focused on designing and implementing the right scoping mechanisms for meta-programs, not on finding errors (with no possibility of automatic fixes) in the generator code.

Full well-formedness. Extrapolating from the above kinds of statically-safe program generation, it is reasonable to ask how easy it is to achieve full static safety. That is, to write generators in such a way that every generated program is guaranteed to pass any static check in the target language. This is a hard property to ensure: static checks beyond the syntax phase (i.e., in type checking or other semantic analysis) require context information, which is tough to maintain by merely analyzing the generator. A rich host language can express generators with arbitrarily complex structure, whose control-flow paths map to different static contexts of generated programs.

To see the problem in an example, consider a program generator that emits programs depending on two input-related conditions:

```
if (pred1())
  emit( ‘[int i;] );
...
if (pred2())
  emit( ‘[i++;] );
```

If, for some input, `pred2` does not imply `pred1` (or if the first `if` is unreachable), then the generator can emit the reference to variable `i` without having generated the definition of `i`. This is an error in the generator and it should be the responsibility of the infrastructure for generator development to prevent such errors. (Of course, it is rather easy to catch this error at generation time of the `i++` fragment, but this just shifts the blame: the generator does not produce an invalid program, but fails to produce anything.)

As the above example shows, static safety for generated programs corresponds to arbitrarily complex properties of the generator's control- or data-flow. In our example, determining the reachability of the statement emitting the declaration of variable `i` is a complex program analysis property.

4 Mechanisms for Fully Structured Generation

In order to achieve guarantees of full static safety for generators, we need to place restrictions on what generators are expressible in a language or development setting. As in any other kind of approach for establishing complex program properties, the restrictions can take many forms: we can limit the expressiveness of the host language for generators via disciplined syntactic or type-system constructs, we can permit only generators that successfully pass an analysis phase, and more. We see such mechanisms next.

4.1 Multi-Stage Programming

A simple way to ensure the static safety of generated programs is to map them one-to-one to fragments of the generator's code. As a result, the generator and the generated program can be viewed as one, are type-checked by the same type system, and some parts of the program are merely evaluated later (i.e., generated). This approach is commonly called *staging* [27] and the program is called *multi-stage* or *staged*. Multi-Stage Programming (MSP) is the general paradigm for writing staged programs. MSP is a powerful and general approach, yet with significant limitations—e.g., a statement of the generator code is not allowed to produce arbitrarily many declarations in the generated program.

MSP views program generation as the addition of extra stages to regular programs. Instead of a plain execution stage (possibly internally broken into multiple stages, such as compilation and object execution) we have at least a generation stage and an execution phase. Programmers make use of a set of language constructs that introduce more stages for explicitly annotated segments of their code, so that these segments are evaluated at different times. At a later time, the previously (partially) evaluated, in earlier stages, parts of the program can be replaced by simpler constructs, such as constant values and statements with linear control flow.

The origins of staging constructs are found in languages like *MetaML* [49]: a statically-typed, multi-stage programming language, as an extension of Standard ML/NJ [1]. MetaML introduced four language constructs:

– the meta-brackets that delay a computation e.g., `<40+2>`. Evaluation cannot happen and the computation is considered *frozen*. These are *future-stage* computations, and can be thought of as generated code.
– the *escape* operator, `~x` for some variable `x`, can be used only inside meta-brackets—e.g., `<40+~x>`. This operator permits calculations at the current stage and splices the result inside the delayed expression for later use. This allows evaluation steps to take place during program generation, i.e., to vary the generated code.
– `run x` forces the evaluation of a meta-bracket expression. Essentially, it compiles the computation at run-time and runs it to produce the result.
– `lift x` allows the conversion of a value—the result of the evaluation of an expression that does not contain a function—into code.

Consider a function whose body contains a mix of staged and unstaged parts. What happens when we evaluate that method with an argument list consisting of some known values and some to-be-supplied at a later stage? MSP evaluates the unstaged and escaped parts of the program, utilizing information that is available at the current stage. Then it produces a residual program that is going to be evaluated at a subsequent stage, when the rest of the parameters are available. This is similar to a *partial evaluation* (PE) of the program [11, 25], where the unstaged and escaped parts are evaluated, with staged parts left for later evaluation. (Staging and partial evaluation are closely related concepts [24, 27]: staging can be seen as instructing a partial evaluator as to what parts of the program to partially evaluate. Conversely, automatic partial evaluation can be seen as computing the staging annotations automatically—a step called *binding-time analysis* in the PE literature [26].)

Applications of staging include the implementation of domain-specific languages [18], building compilers from interpreters [48] and the "finally tagless" approach to building efficient interpreters [6].

We next review staging in more detail via two modern staging implementations.

BER MetaOCaml. MetaOCaml [5] is a bytecode MSP compiler for OCaml and BER MetaOCaml [30] is its continuation: a heavily re-factored version of the MetaOCaml compiler that is more extensible and easier to integrate with releases of the regular OCaml compiler.[4]

We illustrate staging via the folklore example of a simple power function, which has been used for demonstrating partial evaluation (and staging) since at least 1977 [14]. The power function is defined recursively using the basic method of exponentiation by squaring. If the exponent is even we square the result of raising `x` to half the given power. Otherwise, we reduce the exponent by one and we multiply the result by `x`.

```
let even n = (n mod 2) = 0;;
let square x = x * x;;
let rec power n x =
  if n = 0 then 1
  else if even n then square (power (n/2) x)
  else x * (power (n-1) x);;
```

We can stage the above function in the MetaOCaml code below to produce power functions specialized for a certain `n`—e.g., 5. The staged version of the function is identical to the original, with the mere addition of staging annotations/constructs. BER MetaOCaml has three of the four MetaML constructs mentioned earlier: meta-brackets, escape and run. In this example, although `n` is statically known, `x` remains a variable: its value will only be known at a later evaluation stage.

```
open Runcode;;
let even n = (n mod 2) = 0;;
let square x = x * x;;
let rec powerS n x =
  if n = 0 then .<1>.
  else if even n
       then .<square .~(powerS (n/2) x)>.
       else .<.~x *  .~(powerS (n-1) x)>.;;

let power5 = !. .<fun x -> .~(powerS 5 .<x>.)>.;;
```

[4] More historical details about the evolution path of MetaOCaml can be found at "A brief history of (BER) MetaOCaml", `http://okmij.org/ftp/ML/MetaOCaml.html#history` .

Note the structure of the above code. The return value of the `powerS` function is a staged computation, but it includes a part that can be evaluated, which is the recursive application. The result (i.e., the code/AST of the result) is spliced back into the staged computation of the final value, `power5`: a specialized function, to be available to later stages.

The operator `!.` is aliased to `Runcode.run`—the "run" functionality of MetaOCaml. This function compiles the staged lambda function, and links it back as the (specialized) code to be executed in the body of the `power5` function. Simply put, `!.` transfers our code from the world of representations of functions, `(int -> int) code`, to the world of functions, `(int -> int)`.

BER MetaOCaml lets us inspect the code that is generated from our staged algorithm, using the `print_code` function. The result looks like the following snippet. As the reader observes, the recursive applications are performed at compile-time, partially evaluating the function. The result is the residue program below:

```
fun x -> x * (square (square (x * 1)))
```

In the example, the `square` function is referred from a future-stage computation using an identifier (`square`) bound at the present stage. This function is characterized as *cross-stage persistent*.

Lightweight Modular Staging. Rompf et al. introduced MSP support in Scala with *Lightweight Modular Staging* (LMS) [41]. In LMS, the programmer has just one staging construct available, in the form of a user-level type. To indicate that, e.g., an expression does not have a current-stage integer value but a future-stage integer value, the user changes the declared type of the expression from `Int` to `Rep[Int]`. The unary abstract type constructor `Rep[_]` indicates future-stage values. Types for other values, as well as the exact version of (overloaded, current or future stage) operators are inferred.

LMS follows a library-based approach, relying on a special and extensible version of the Scala compiler called *Scala-Virtualized* [39]. In Scala-Virtualized, a Scala program is represented in terms of function calls, e.g., the control-flow construct `do b while (c)` is represented by the `__doWhile(b, c)` function call. In this way, all interesting program statements are mapped to function calls. Even method calls are represented as infix functions—e.g., `x.a(y)` as `infix_a(x,y)`. This technique permits operations to be added to the type `Rep[T]` which also supports every method of a bare `T`. Staging `power` in LMS resembles the following code:

```
def even (n: Int) = n % 2 == 0
def square (x: Rep[Int]) = x * x
def powerS (n : Int, x : Rep[Int]) : Rep[Int] = {
  if (n == 0) 1
  else if (even(n)) square(powerS(n/2, x))
  else x * powerS(n-1, x)
}
```

```
def powerTest(x : Rep[Int]) : Rep[Int] = powerS(5, x)
```

The input that needs to be designated as *future-stage* is the base `x`. This is the dynamic part of this snippet. All other is static input and will participate in compile-time evaluation. The type constructor `Rep` in, e.g., `Rep[T]` has the property that all operations on T are applicable to `Rep[T]` as well and operations on it will be generated later.

The core difference of LMS from other staging approaches is that *binding times (i.e., stages) are distinguished only by types*. The simplicity of the type-based approach to staging has been a significant boost for LMS, and owes much to the power of the Scala type system. In LMS applications, regular computations are routinely switched to staged computations with small, local changes to declared types.

Practical Notes. The two modern incarnations of staging concepts (BER MetaOCaml and LMS) integrate several practical enhancements and have seen significant applications. In both technologies, the user has multiple ways to generate code. For example, in LMS, CUDA code can be generated instead of Scala, and in BER MetaOCaml the user can compile directly to native code instead of the bytecode-generating, `Runcode.run` function. A notable recent application, with versions for both BER MetaOCaml and LMS, is the Strymonas library [31], which offers highly-optimized streaming functionality (e.g., `map`, `filter`, `zip` combinators), often over an order-of-magnitude faster than conventional libraries. More generally, staging, using LMS, has been proposed as a key part of an ambitious development methodology for high performance without sacrificing abstraction [40]. The methodology has seen several instances of successful application. It centers around the creation of domain-specific languages (DSLs) that obtain high-performance implementations via interpreters staged to become (effectively) compilers.

4.2 Class Morphing

Class morphing is a technique for writing program generators that take classes as input and generate new type-safe classes, based on the structure of the input ones. Morphing has been implemented as MorphJ [16, 19–21]: a language that adds compile-time reflection capabilities to Java. A programmer is able to capture compile-time patterns and encode them in (meta-)classes.

In MorphJ, a generator `Gen` taking as input a class `C` corresponds to a meta-class `Gen` parameterized by the input `C`, similar to Java generics (`Gen<C>`). From this perspective, morphing is a strong generalization of generics: for different values of the type parameters of a class `Gen`, radically different contents may be generated, which are the output of the generator. The body of the generator class then contains regular Java code mixed with MorphJ annotations describing the *reflection* (i.e., content-inspection) patterns that guide generation.

Each pattern is associated with a generative scenario. In the following example, the `LogMe` generator accepts as a parameter a class `X` and produces a subtype of `X` that logs the returns of calls to `X`'s (non-`void`) methods:

```
class LogMe<class X> extends X {
  <R,A*>[m] for ( public R m(A) : X.methods )
  public R m (A a) {
    R result = super.m(a);
    System.out.println(result);
    return result;
  }
}
```

The second line in the above (meta-)class is a *static for-loop* over methods of class `X` (designated `X.methods`) that match a pattern, `public R m(A)`, where `R` and `A` are type parameters (`A` can match any number of type parameters, as indicated by the `A*` syntax in its declaration) and `m` is a name parameter, as indicated by its distinct declaration syntax (`[m]`). Other facilities for inspecting the contents of type parameters (e.g., iterating over fields) are defined similarly in MorphJ.

In another example, a morphed class `Listify` may statically iterate over all the methods of another, unknown, type, `Subj`, pick those that have a single argument, and offer analogous "lifted" methods: whenever `Subj` has a method with argument `A`, `Listify` accepts a `List<A>`. (The implementation of every method in `Listify` can then, e.g., iterate over all list elements, and manipulate them using `Subj`'s methods.)

```
class Listify<Subj> {
  Subj ref;
  Listify(Subj s) {ref = s;}

  <R,A>[m] for (public R m(A): Subj.methods)
  public R m (List<A> a) {
    ... /* e.g., call m for all elements */
  }
}
```

Observe here that, in contrast to the previous example, this generator does not generate a subtype of `Subj` but an unrelated class that internally uses a `Subj` object.

In general, MorphJ offers program transformation capabilities but with modular type-safety guarantees: type-checking (via MorphJ) the code of `Listify` guarantees that all the classes it may produce (for any type `Subj`) also type-check (via the plain Java type system).

Type-checking a MorphJ program/generator to guarantee the static safety of all possible generated classes is based on determining the *uniquness* of declarations and the *validity* of references. Uniqueness means that for each generated declaration of a variable or method in morphed code, the MorphJ type system needs to ensure that the declaration does not conflict with others in the same scope. Validity means that each reference to an identifier needs to map to an appropriate, type-correct morphed or pre-existing declaration in the morphed code. Because of static for-loops, a single identifier in morphed code (e.g., `m` in the `LogMe` class, above) maps to possibly many generated identifiers. Therefore, the checking of uniquness and validity needs to process the reflection patterns of static for-loops. This checking is done in MorphJ via the well-known concept of *unification* in patterns: for uniqueness, any two declaration-generating patterns in the same scope should never unify, while for validity, the pattern producing a reference should be a specialization (i.e., one-way unification) of a pattern producing a corresponding declaration. The pattern-based type-checking mechanism is mixed with subtyping and conventional type reasoning in the full MorphJ type checker [20]. The decidability of type-checking also hinges on expressiveness limitations placed on the MorphJ program: static for-loops cannot be nested (although a single for-loop can contain a nested, secondary pattern, with type variables bound in the primary pattern), and there is only a limited compile-time conditional statement [19].

The generator programmer has several facilities for influencing the result of MorphJ type-checking. The simplest one is that of stating subtyping constraints on type parameters (e.g., `C<X extends I>`), as in regular Java. Additionally, the programmer can add explicit static prefixes or suffixes to generated identifiers, to ensure their uniqueness—the *a*#*b* notation designates identifier concatenation with one of *a*, *b* being a constant.

Morphing enables great expressiveness when added to a conventional language. In fact, morphing can even simulate inheritance by offering safe delegation over classes. This requires some extra functionality, namely the addition of a single keyword (`subobject`) [16]. To see why such an extension might be needed, consider this example of a logger generator similar to `LogMe` given above:

```
class Logger<Subj> {
  Subj ref;
  Logger(Subj s) {ref = s;} // initialize

  <R,A*>[m] for (public R m(A) : Subj.methods)
  public R m (A a) {
    System.out.println("method " + m.name + " called with arg " + a);
    return ref.m(a);
  }
}
```

The morphed methods defined here report when they are called but the delegation will occur only during external-client calls to methods of `Logger<C>`, for some `C`. Any further calls happening inside the parent are not logged and thus the morphed class does not behave like a class that would override each method with a logged one. The problem is the lack of the *late binding* property. To remedy this, the object targeted by delegation must be identified using the `subobject` keyword:

```
class Logger<Subj> {
  subobject Subj ref;
  ... // as before
}
```

This small change ensures that method calls are late-bound so that a generated class, `Logger<C>` behaves like a true subclass of the original, `C`. However, the addition of `subobjects` has significant repercussions. For instance, two references to the same subobject (i.e., aliases), via different access paths can behave differently [16].

4.3 Shortcomings and More Power

Staging and morphing are broad approaches that can ensure statically-safe program generation, by focusing on specific (albeit broad) classes of program generation tasks. For more power, there have been several approaches that allow arbitrary program generation constructs, yet impose a discipline (based on type-checking or other analysis) over how these constructs are composed. We see some interesting such mechanisms next, noting the differences from staging and morphing, both in style and in expressive power.

SafeGen. SafeGen [22] is a meta-programming language for writing generators of Java code that are statically guaranteed to produce type-safe code.

The SafeGen language targets generators that can be written as transformations using reflection, i.e., inspection of the structure of existing code. SafeGen can thus be used for tasks similar to those that class morphing targets. SafeGen handles generated names and can guarantee that generated definitions have fresh names, not clashing with existing ones.

Compared to multi-stage languages and class morphing, SafeGen is more expressive in principle, as it permits the specification of a generator via arbitrary constructs. For example, it permits generators that generate and use names more freely than the scoping of multi-stage languages allows; it also enables generators that test arbitrary logic formulas over reflective properties, compared to the more constrained way of handling the same properties in morphing.

However, the type system of SafeGen is undecidable and it depends on an automated theorem prover for discharging proof goals for common cases. This means that more generators can be written and proved correct in SafeGen, but the user can also write generators that SafeGen cannot prove correct automatically and will report a “possible error”.

A simple example in SafeGen (that could also be written using morphing) is that of a generator that takes a Java class as input and produces a Java interface of `void` methods that have the same name as the methods in the input class:

```
#defgen MakeInterface (Class c) {
  interface I { #foreach(Method m : MethodOf(m,c)) { void #[m] (); } }
}
```

In the code above, #defgen declares the generator, the #foreach syntax iterates over the methods of the input class, and #[m] uses the name of the meta-variable m in the generated code. This generator may seem too simple, but is buggy: since Java permits method overloading, two methods of the input class can have the same name and thus the generator may generate an interface with duplicate method declarations. SafeGen catches this error as it cannot prove that the generated output will always be type-safe.

A working SafeGen example is that of a generic delegator, which was also given in Section 4.2, using morphing:

```
#defgen MakeDelegator ( input(Class c) => !Abstract(c) ) {
  #foreach( Class c : input(c) ) {
    public class Delegator extends #[c] {
      #foreach(Method m : MethodOf(m, c) & !Private(m)) {
        #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
          return super.#[m](#[m.ArgNames]);
} } } } }
```

Although this is essentially a morphing example, the flavor of operators offers a glimpse of actual program manipulation with SafeGen. Static reflective iteration is supported, as well as arbitrary generation-time nesting of primitives (e.g., nesting of #foreach loops), compile-time conditionals, identifier manipulation, etc. The type system of the target language is fully encoded in input for the automated theorem prover that SafeGen employs as part of its checking. Part of this input is constant and encodes assumptions of the language (e.g., the single-inheritance nature of Java), while another part is custom-generated by translation of the generator specification, to encode the structure of the generated code. Finally, the generator specification is used to produce a logical sentence that establishes the generated program's well-formedness for unknown generator inputs. The automated theorem prover then attempts to prove this sentence.

In all, despite its power, the SafeGen approach suffers from lack of programmer control, due to the undecidable nature of the checking process. It is not easy to know when a generator fails to type-check due to a bug vs. due to limitations in automated formal reasoning.

Ur. Instead of translating a generator specification into a logical sentence for an automated theorem prover (as in SafeGen), one can attempt to enlist a powerful type system that can simultaneously express conventional type-level properties of a program and the logical structure of a generator under unknown inputs. This typically entails the use of *dependent types*: types that can use program expressions as terms. An advantage of this approach is that the user can improve the theorem proving ability of the system by just applying better type annotations (though these can be arbitrarily complex).

The Ur system [7] for program generation adopts this principle. Ur permits the declaration of generators that can be generic on their input while still producing only well-formed output. Ur's metaprogramming model is based on type-level computation and type-level records, following a functional programming style. The input of a generator defined in Ur can contain records of values or types and such records can be taken apart or built in the body of the generator, in a safe way, using functional programming machinery (e.g., higher-order functions such as `map` and `fold`). The type system keeps track of the origin and manipulation structure of records and values, much like the SafeGen type system, earlier, kept track of the patterns used to produce definitions and references to identifiers.

Targeting pragmatic applications and ease of use, Ur is using a restricted form of dependent types, combined with ad-hoc logic for common cases (such as special provers for the inference of intermediate proofs of a particular shape or automatic code transformations, such as `map` fusion). The safety guarantees of Ur assume that the writer of the generator dedicates some effort in writing type annotations and reasoning about output safety using the record-specific features of the language. On the other hand, the *user* of the generator can be spared this effort as Ur's heuristics fill in many holes, resulting in simple-to-use generators.

Although Ur offers type safety based on its records reasoning, the data format output by a generator may be subject to additional well-formedness constraints, such as the need for sanitization in HTML and SQL to address code injection attacks. In the case of HTML and SQL, Ur has been extended with additional functionality that guarantees this well-formedness, resulting in Ur/Web, a domain-specific language for web application development, implemented on top of Ur as a special library with extra rules for parsing and optimization [7, 9]. Other generated data formats and their needs would need a similar extension of Ur.

As an example, consider a dynamic webpage that defines a generic `sum` function that sums an arbitrary list (record) of integers and then calls it to sum three lists of integers (one of them being the empty list):[5]

```
fun sum [fs ::: {Unit}] (fl : folder fs) (x : $(mapU int fs)) =
    @foldUR [int] [fn _ => int]
     (fn [nm :: Name] [rest :: {Unit}] [[nm] ~ rest] n acc => n + acc)
     0 fl x

fun main () = return <xml><body>
  {[sum {}]}<br/>
  {[sum {A = 0, B = 1}]}<br/>
  {[sum {C = 2, D = 3, E = 4}]}
</body></xml>
```

In this example, `main` is the entry point of the dynamic web page, which calls `sum` three times. (The `nm` and `~` syntax found in the body of `sum` is part of the Ur/Web support for record manipulation.) The results of `sum` calls are then embedded in well-formed XML. The `sum` function is declared using a fold, as in standard

[5] The two pieces of code here are contained in Ur/Web distribution version 20150520 as demos `sum` and `tcSum`.

functional programming practice. The interesting part is the ability to iterate over the (unknown) fields of any record and to modularly assert that the iteration (i.e., `sum`) is well-defined, no matter what record is supplied as input. What sets Ur/Web apart from other Web frameworks is the amount of type information inferred: only `x` is explicitly given in the calls to the function; everything else is inferred. In particular, Ur/Web manages to infer the folding mechanism used (the `folder`). While in principle the type inference of Ur/Web is not complete, in practice it addresses many common cases encountered during Web development.

The example above was the definition of summing for integer records. The following example shows how to define a generic `sum` in a similar fashion:

```
fun sum [t] (_: num t) [fs ::: {Unit}] (fl: folder fs) (x: $(mapU t fs)) =
    @foldUR [t] [fn _ => t]
     (fn [nm :: Name] [rest :: {Unit}] [[nm] ~ rest] n acc => n + acc)
     zero fl x

fun main () = return <xml><body>
  {[sum {A = 0, B = 1}]}<br/>
  {[sum {C = 2.1, D = 3.2, E = 4.3}]}
</body></xml>
```

This example uses type classes, another feature of functional programming [55], to define `sum` on number-like data, i.e., values of types in the `num` type class. In this way, `sum` can be applied to records of integers and floats, with the same ease of development (inference) as before.

4.4 Other Techniques

There are several other techniques that are close relatives of the ones we discussed in the previous sections. We mention some of them for completeness and as starting points for further study.

The Genoupe system [13, 34] allows expressing generators in an extended version of C#, using constructs similar to the static `#foreach` of SafeGen. Like SafeGen, the system allows arbitrary expressiveness (e.g., nesting of static for loops, static conditionals, and more) but, in contrast to SafeGen, does not encode the full complexity of the meta type-safety question in its type-checking. In SafeGen, this complexity mandated the use of a rich logic and an automated theorem prover. In contrast, the Genoupe checking is done through standard type-system techniques and is even more restrictive than MorphJ's. For instance, there is no way to generate declarations under a set of conditions and generate references to them under stricter conditions—the type system treats expressions in static constructs as opaque values that can only be compared for equality.

The compile-time reflection (CTR) facility [15] is a close relative of morphing, yet presents a different tradeoff in the design space. It introduces the concept of a self-contained transformation (instead of merging meta-programming features inside generic classes, as in MorphJ) and sacrifices some modular type safety: the system catches invalid references, though not duplicate definitions. The work has been recently extended with addition of features from morphing, and applied to several different program elements, such as pattern-based traits and reflection at the statement level [36, 37].

In recent work on active libraries, Servetto and Zucca propose MetaFJig*, a rich meta-language for safe reflection with nested class support and composition operators [42]. These new features do not have a counterpart in classical morphing. In Section 4.5 we discuss interesting research avenues that incorporate such features.

Concepts of static safety have also arisen in the context of refactoring transformations, with work by Steimann and von Pilgrim [47]. An interesting aspect of this work is that it treats the program as a constant (i.e., does not guarantee the safety of a refactoring for all possible input programs) yet attempts to solve a hard problem—namely, to compute the constraints that need to hold in the post-transformation state of the program for the refactoring to have been semantics-preserving.

Recent work in the literature has focused on offering static safety guarantees for macro systems and other syntactic extension mechanisms. Lorenzen and Erdweg [33] propose a syntactic language extension facility that offers *type-based* syntax desugaring (allowing the desugaring specifications to employ type information) while guaranteeing automatically that desugarings only generate well-typed code. Chlipala's Bedrock system [8] is a relative of the Ur approach of Section 4.3. Bedrock introduces "certified low-level macros", for an assembly-level target language. These are highly expressive macros, allowing the implementation of a full C-like language stack. However, safety guarantees carry the cost of some manual verification effort by the programmer. The host language in Bedrock is the functional programming language of the Coq proof assistant [3]. In this setting, safety properties are also low level, guaranteeing the absence of invalid jumps or bad memory reads/writes in the resulting machine code.

4.5 Remarks and Future

The techniques we saw in the previous sections cover several points in the space of expressiveness/static safety tradeoffs. Perhaps the techniques with the easiest path to mainstream adoption are the ones that enforce clear, up-front expressiveness restrictions, yet support general as well as popular program generation patterns. Staging and morphing (Sections 4.1 and 4.2) are the clearest such instances. Both of them have significant expressiveness limitations and addressing such limitations is the topic of active research.

Staging requires a one-to-one mapping between code fragments of the generator and the generated code. In multi-stage languages, one cannot escape an identifier in a binding position. For instance, it is not possible to generate the definition of a variable whose name will be determined at generator run-time, as in:

```
emit( ‘[ int #name; ] );
```

In a recent position paper [23], Inoue et al. argue that the "next stage of staging" will need to lift constructs to the type level. Allowing the generation of binding instances with variable names ("splicing binders") is identified as a major challenge. This is indeed the focus of morphing mechanisms: reasoning about generated declarations and their references, without knowing what the declared names will be until generator run-time. Therefore, an interesting direction for both morphing and staging are to combine forces, in language designs that enable meta type-safety for some of the most common kinds of program generation.

At the same time, morphing is evolving to acquire further functionality, for reflecting over classes. Recent work [4] presents *universal morphing*: an extension of morphing to permit patterns iterating over types. This ability captures morphing functionality at a much larger granularity than before, and enables several interesting programming abstractions. Examples include iterating over all supertypes of a class, over all its nested classes, over all classes in a given set, etc. Such static iteration can generate new classes that subclass or reference input classes, emulate a subset of their supertypes while adding new ones, morph over their members (via standard morphing patterns), etc. This enables, for instance, highly generic *mixin layers* [44]: parametric components containing classes that each inherit from a corresponding class in an unknown super-component supplied as a parameter. The ability to iterate over classes can also be used to support type constructor polymorphism and higher-kinded types, as in the work of Moors et al. [38].

In Section 4.4 we mentioned MetaFJig* as a source of features that are still not satisfactorily handled by other meta-programming systems. MetaFJig* defines a language that supports reflection over classes: the user can write reflective code that generates expressions of a "class" type (i.e., class definitions are seen as expressions) and there are operators over classes (such as sum). Universal morphing can also support versions of these concepts: it supports safe reflection over nested classes, and class operators can be encoded through morphing. For an example of this encoding, the sum operator can be defined as a class `Sum<A,B>` that contains methods of both `A` and `B`, guarantees the absence of method name clashes, and supports recursive summing.

5 Conclusions

Programming language design evolves with the invention of new kinds of abstraction. *Procedural abstraction* was introduced in the 50s and 60s and ushered in the era of structured programming languages. *Type abstraction* or *polymorphism*, in the 70s and 80s, brought about modern functional and object-oriented languages. These advances were foreshadowed by program generation techniques that attempted to achieve the same expressiveness benefits with much lower-level concepts and no safety guarantees. Before there were structured procedures, there were macros that achieved similar benefits in many cases. Before there was polymorphism, there were generators that produced isomorphic code for different types, or copied one type's definitions into another. Generators are inevitable every time existing abstraction mechanisms are not enough. Conversely, getting generators to support an abstraction pattern with full static safety means that they are no longer “generators”: it becomes a mere implementation detail whether a program is indeed generated as part of supporting an abstraction.

Therefore, the question of how to make generators statically type safe (i.e., statically checkable for all possible, unknown, inputs) is central to the future of programming language design. A technique from this solution space may be behind the next major evolution of programming languages. For instance, the morphing approach exemplifies *structural abstraction*: code can be agnostic of the structure of other program elements, yet interface with them correctly. This matches the widespread low-level practice of generating code via reflecting over existing classes or modules.

Given the importance and appeal of the underlying problem, it is no surprise that the field is active and diverse. This briefing gave an overview from a viewpoint that we hope illuminates rather different and typically disconnected approaches.

Acknowledgments This work was funded by the Greek Secretariat for Research and Technology under the “MorphPL” Excellence (Aristeia) award.

References

1. Appel, A., MacQueen, D.: Standard ML of New Jersey. In: Programming Language Implementation and Logic Programming. pp. 1–13. Springer (1991), `http://www.springerlink.com/index/YMU0P7QN06713188.pdf`
2. Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: tools for implementing domain-specific languages. In: Proceedings Fifth International Conference on Software Reuse. pp. 143–153. IEEE, Victoria, BC, Canada (1998), `citeseer.nj.nec.com/171171.html`
3. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer Publishing Company, Incorporated, 1st edn. (2010)
4. Biboudis, A., Fourtounis, G., Smaragdakis, Y.: jUCM: Universal Class Morphing (position paper). CoRR abs/1506.05270 (2015), `http://arxiv.org/abs/1506.05270`
5. Calcagno, C., Taha, W., Huang, L., Leroy, X.: Implementing multi-stage languages using ASTs, gensym, and reflection. In: Generative Programming and Component Engineering. pp. 57–76. Springer (2003), `http://link.springer.com/chapter/10.1007/978-3-540-39815-8_4`
6. Carette, J., Kiselyov, O., Shan, C.c.: Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. Journal of Functional Programming 19(5), 509–543 (Sep 2009), `http://dx.doi.org/10.1017/S0956796809007205`
7. Chlipala, A.: Ur: Statically-typed Metaprogramming with Type-level Record Computation. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 122–133. PLDI '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1806596.1806612`

8. Chlipala, A.: The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 391–402. ICFP '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2500365.2500592
9. Chlipala, A.: The Ur/Web Manual (2015), http://www.impredicative.com/ur/manual.pdf
10. Clinger, W., Rees, J.: Macros that work. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 155–162. POPL '91, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/99583.99607
11. Consel, C., Danvy, O.: Tutorial notes on partial evaluation. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 493–501. POPL '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/158511.158707
12. Czarnecki, K., O'Donnell, J.T., Striegnitz, J., Taha, W.: DSL Implementation in MetaOCaml, Template Haskell, and C++. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation, Lecture Notes in Computer Science, vol. 3016, pp. 51–72. Springer Berlin Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-25935-0_4
13. Draheim, D., Lutteroth, C., Weber, G.: A type system for reflective program generators. In: Glück, R., Lowry, M.R. (eds.) Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3676, pp. 327–341. Springer (2005), http://dx.doi.org/10.1007/11561347_22
14. Ershov, A.P.: On the partial computation principle. Information processing letters 6(2), 38–41 (1977), http://www.sciencedirect.com/science/article/pii/0020019077900783
15. Fähndrich, M., Carbin, M., Larus, J.R.: Reflective Program Generation with Patterns. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering. pp. 275–284. GPCE '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1173706.1173748
16. Gerakios, P., Biboudis, A., Smaragdakis, Y.: Forsaking Inheritance: Supercharged Delegation in DelphJ. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 233–252. OOPSLA '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2509136.2509535
17. Herman, D., Wand, M.: A Theory of Hygienic Macros. In: Drossopoulou, S. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 4960, pp. 48–62. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78739-6_4
18. Herrmann, C.A., Langhammer, T.: Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. Science of Computer Programming 62(1), 47 – 65 (2006), http://www.sciencedirect.com/science/article/pii/S0167642306000736
19. Huang, S.S., Smaragdakis, Y.: Expressive and Safe Static Reflection with MorphJ. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 79–89. PLDI '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1375581.1375592

20. Huang, S.S., Smaragdakis, Y.: Morphing: Structurally Shaping a Class by Reflecting on Others. ACM Transactions on Programming Languages and Systems 33(2), 6:1–6:44 (Feb 2011), `http://doi.acm.org/10.1145/1890028.1890029`
21. Huang, S.S., Zook, D., Smaragdakis, Y.: Morphing: Safely shaping a class in the image of others. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). pp. 399–424. Springer-Verlag (Aug 2007)
22. Huang, S.S., Zook, D., Smaragdakis, Y.: Statically safe program generation with safegen. Science of Computer Programming 76(5), 376 – 391 (2011), `http://www.sciencedirect.com/science/article/pii/S0167642308001111`
23. Inoue, J., Kiselyov, O., Kameyama, Y.: The next stage of staging. In: Proceedings of the 17th workshop on Programming and Programming Languages (PPL) (2015)
24. Jones, N.D.: An introduction to partial evaluation. ACM Computing Surveys 28(3), 480–503 (Sep 1996), `http://doi.acm.org/10.1145/243439.243447`
25. Jones, N.D., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: The generation of a compiler generator. SIGPLAN Notices 20(8), 82–87 (Aug 1985), `http://doi.acm.org/10.1145/988346.988358`
26. Jones, N.D., Sestoft, P., Søndergaard, H.: Mix: A self-applicable partial evaluator for experiments in compiler generation. LISP and Symbolic Computation 2(1), 9–50 (1989), `http://dx.doi.org/10.1007/BF01806312`
27. Jørring, U., Scherlis, W.L.: Compilers and staging transformations. In: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 86–96. POPL '86, ACM, New York, NY, USA (1986), `http://doi.acm.org/10.1145/512644.512652`
28. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001: Proceedings of the 15th European Conference on Object Oriented Programming. pp. 327–353. Springer-Verlag, London, UK (2001)
29. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997: Proceedings of the 11th European Conference on Object Oriented Programming, vol. 1241, pp. 220–242. Springer-Verlag, Heidelberg, Germany, and New York (1997), `citeseer.ist.psu.edu/article/kiczales97aspectoriented.html`
30. Kiselyov, O.: The Design and Implementation of BER MetaOCaml. In: Functional and Logic Programming, pp. 86–102. Springer (2014), `http://link.springer.com/chapter/10.1007/978-3-319-07151-0_6`
31. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 285–299. POPL '17, ACM, New York, NY, USA (2017), `http://doi.acm.org/10.1145/3009837.3009880`
32. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic Macro Expansion. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming. pp. 151–161. LFP '86, ACM, New York, NY, USA (1986), `http://doi.acm.org/10.1145/319838.319859`
33. Lorenzen, F., Erdweg, S.: Sound type-dependent syntactic language extension. In: Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 204–216. POPL '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2837614.2837644`
34. Lutteroth, C., Draheim, D., Weber, G.: A type system for reflective program generators. Science of Computer Programming 76(5), 392–422 (2011), `http://dx.doi.org/10.1016/j.scico.2010.12.002`

35. Mainland, G.: Why It's Nice to Be Quoted: Quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 73–82. Haskell '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1291201.1291211
36. Miao, W., Siek, J.: Pattern-based Traits. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. pp. 1729–1736. SAC '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2245276.2232057
37. Miao, W., Siek, J.: Compile-time Reflection and Metaprogramming for Java. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation. pp. 27–37. PEPM '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2543728.2543739
38. Moors, A., Piessens, F., Odersky, M.: Generics of a higher kind. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications. pp. 423–438. OOPSLA '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1449764.1449798
39. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation. pp. 117–120. PEPM '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2103746.2103769
40. Rompf, T., Brown, K.J., Lee, H., Sujeeth, A.K., Jonnalagedda, M., Amin, N., Ofenbeck, G., Stojanov, A., Klonatos, Y., Dashti, M., Koch, C., Püschel, M., Olukotun, K.: Go Meta! A case for generative programming and DSLs in performance critical systems. In: Ball, T., Bodík, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA. LIPIcs, vol. 32, pp. 238–261. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015), http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.238
41. Rompf, T., Odersky, M.: Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. pp. 127–136. GPCE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1868294.1868314
42. Servetto, M., Zucca, E.: A meta-circular language for active libraries. Science of Computer Programming 95, Part 2, 219 – 253 (2014), http://www.sciencedirect.com/science/article/pii/S0167642314002317, selected and extended papers from Partial Evaluation and Program Manipulation 2013
43. Sheard, T., Jones, S.P.: Template Meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 1–16. Haskell '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/581690.581691
44. Smaragdakis, Y., Batory, D.: Implementing layered designs with mixin layers. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). pp. 550–570. Springer-Verlag LNCS 1445 (1998), citeseer.nj.nec.com/smaragdakis98implementing.html
45. Smaragdakis, Y., Batory, D.: Scoping constructs for program generators. In: Generative and Component-Based Software Engineering Symposium (GCSE). pp. 65–78. Springer-Verlag, LNCS 1799 (1999), earlier version in Technical Report UTCS-TR-96-37
46. Sperber, M., Dybvig, R.K., Flatt, M., van Straaten, A., Findler, R., Matthews, J.: Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press, New York, NY, USA, 1st edn. (2010)

47. Steimann, F., von Pilgrim, J.: Constraint-based refactoring with foresight. In: Proceedings of the 26th European Conference on Object-Oriented Programming. pp. 535–559. ECOOP'12, Springer-Verlag, Berlin, Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-31057-7_24`
48. Taha, W.: A Gentle Introduction to Multi-stage Programming. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation, pp. 30–50. No. 3016 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (2004), `http://link.springer.com/chapter/10.1007/978-3-540-25935-0_3`
49. Taha, W., Sheard, T.: Multi-stage Programming with Explicit Annotations. In: Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. pp. 203–217. PEPM '97, ACM, New York, NY, USA (1997), `http://doi.acm.org/10.1145/258993.259019`
50. Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., Felleisen, M.: Languages As Libraries. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 132–141. PLDI '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1993498.1993514`
51. Tratt, L.: Domain Specific Language Implementation via Compile-time Metaprogramming. ACM Transactions on Programming Languages and Systems 30(6), 31:1–31:40 (Oct 2008), `http://doi.acm.org/10.1145/1391956.1391958`
52. Veldhuizen, T.: Expression templates. C++ Report 7, 26–31 (1995)
53. Veldhuizen, T.: C++ templates are Turing complete. Tech. rep., Indiana University (2003)
54. Visser, E.: Meta-programming with concrete object syntax. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering. pp. 299–315. GPCE '02, Springer-Verlag, London, UK, UK (2002), `http://dl.acm.org/citation.cfm?id=645435.652697`
55. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 60–76. POPL '89, ACM, New York, NY, USA (1989), `http://doi.acm.org/10.1145/75277.75283`
56. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ programs with meta-AspectJ. In: Generative Programming and Component Engineering (GPCE). pp. 1–18. Springer-Verlag (October 2004)