Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Meta C++: an extension layer for multi-stage generative metaprogramming

Yannis Lilis[a] Anthony Savidis[ab]

a. Institute of Computer Science, FORTH

b. Department of Computer Science, University of Crete

Abstract Generative metaprogramming is a powerful mechanism for reuse through code manipulation, adaptation and composition. Its practicing is complex, involving development demands similar to any actual system. In C++, metaprogramming is currently practiced with templates, requiring a functional programming style that is in contrast to the imperative object-oriented nature of the main language. Thus, metaprograms bear little resemblance to normal programs, and involve different programming approaches in their development, effectively disabling any possibility for design or code reuse between them. In this paper, we propose *MetaC++*, an extension layer supporting multi-stage generative metaprogramming for C++, offering the full-range of language constructs for implementing compile-time metaprograms and allowing them to share development practices with normal programs. We present its design and implementation, and outline the importance of such an extension to C++ through a number of advanced application scenarios.

Keywords Metaprogramming; Generative Programming; Compile-Time Metaprogramming; Multi-Stage Languages; Language Implementation.

1 Introduction

Multi-stage generative metaprogramming concerns programs encompassing definitions that when evaluated, either at compile-time or at runtime, generate source code that is put in their place. Such definitions handle source code in the form of *Abstract Syntax Trees* (ASTs). Metaprogramming can help achieve various benefits [She01], including performance, partial evaluation, reasoning or validation of object programs, embedding of domain specific languages, and code reuse.

C++ [Str13] in particular has a long history of adopting metaprogramming practices. First was the C Preprocessor [KR88] whose macros allow generating code

Yannis Lilis, Anthony Savidis. *Meta C++: an extension layer for multi-stage generative metaprogramming*. In Journal of Object Technology, vol. 18, no. 1, 2019, pages 1:1–44. doi:10.5381/jot.2019.18.1.a1

through text-based substitutions. Then C++ templates were introduced, offering a Turing Complete [Vel03] functional language interpreted at compile time [AG04, Vel96] as part of the type system, enabling compile-time computations. Template metaprograms have become an essential part of modern C++ programs with well-established libraries [AG04, Ale01] being widely adopted and new ones emerging [SP12, Dio] as the language evolves. Finally, besides variadic templates, C++11 also introduced the `constexpr` specifier, enabling functions or variables to be evaluated at compile time.

Motivation. Despite the aforementioned support, metaprogramming in C++ is still an open issue. The C Preprocessor is inadequate for metaprogramming as it operates at a lexical level. Additionally, template metaprograms involve a fundamentally different programming approach compared to the class-based imperative nature of the normal language. Metaprogramming involves an inherent complexity, while normal C++ programming is also complex on its own. Thus, requiring metaprogrammers to also be proficient in an entirely different programming style that involves custom coding patterns and idioms, places a significant extra burden on them. More importantly though, from a software engineering perspective, the different programming style disables any design or source code reuse for similar problems. For instance, consider the Fibonacci sequence implementations shown on the top part of Figure 1. The runtime version uses just a normal function while the compile-time version requires template classes and recursive template specialization.

Also, any C++ library code, including STL, has to be reimplemented to be used in the context of a template metaprogram. For STL in particular, `boost::mpl` [AG04] containers, iterators and algorithms essentially replicate the functionality of their STL counterparts, while again requiring a different programming style. For example, the bottom part of Figure 1 shows the normal code for creating a vector with integers and its compile-time equivalent that uses the `boost::mpl::vector` sequence, the `boost::mpl::push_back` and `boost::mpl::at_c` metafunctions, the `boost::mpl::int_` integral constant wrapper and type declarations for expressing the compile-time data.

Apart from the different programming style, non-trivial metaprograms also require lengthier and convoluted code. For example, as we will see in section 4, the equivalent of a for loop over a standard container requires an elaborate combination of variadic templates, template template parameters and recursive template specializations. Over-

	Runtime version	*Compile-time version*

Fibonacci sequence	`int fibonacci(int n) {`
	`  if (n==0|| n==1) return 1;`
	`  else return fibonacci(n-1)+`
	`  fibonacci(n-2);`
	`}`
	`printf("%d", fibonacci(5));`
	`// 8, calculated at runtime`
		`template<int n> struct Fibonacci {`
		`  enum { value = Fibonacci<n-1>::value +`
		`  Fibonacci<n-2>::value };`
		`};`
		`template<> struct Fibonacci<0> { enum {value=1};};`
		`template<> struct Fibonacci<1> { enum {value=1};};`
		`printf("%d", Fibonacci<5>::value);`
		`// 8, calculated at compile-time`

Vector of Integers	`#include <vector>`
	`std::vector<int> numbers{1,2};`
	`numbers.push_back(3);`
	`assert(numbers[2] == 3);`
		`#include <boost/mpl/int.hpp>`
		`#include <boost/mpl/vector.hpp>`
		`#include <boost/mpl/push_back.hpp>`
		`#include <boost/mpl/at.hpp>`
		`using namespace boost::mpl;`
		`using temp = vector<int_<1>, int_<2>>;`
		`using numbers = push_back<temp, int_<3>>::type;`
		`static_assert(at_c<numbers, 2>::type::value == 3,`
		`  "assertion failed");`

Figure 1 – Examples of C++ programs and equivalent template metaprograms

all, template metaprogramming involves extremely complicated code patterns that are difficult to write, test, debug and maintain even for experienced C++ developers.

Functions qualified as `constexpr` allow the development of metaprograms in a fashion similar to normal programs, but the limitations on the programming elements allowed for compile-time computations disable adopting them for full-scale metaprogramming. In any case, metaprogramming is supported by computing values at compile-time and instantiating template code with concrete types and values, so there is no notion of code as a first class value that would allow supporting code generation, traversal, manipulation or introspection.

On the other hand, multi-stage languages [TS00, Tah04, She99] take the programming task of code generation and support it as a first-class language feature through the use of special syntax, promoting a metaprogramming paradigm where the meta-language is a minimal superset of the host language.

Contribution. In our work, we adopt aspects of multi-stage programming and propose *MetaC++*, a multi-stage extension of C++ that supports generative metaprogramming. Key aspect of our design is that apart from a common syntax, metaprograms should also share common development practices with normal programs, fully reusing C++ as the language for implementing metaprograms. For instance, since classes, modules and libraries can be used in C++ programs, they should also be allowed in metaprograms and indeed in the same manner. Overall, our main contributions are:

- Multi-stage metaprogramming under the complex context of C++
- Generative metaprogramming for C++ without complicated template patterns
- A metaprogramming model for C++ that treats metaprograms as full scale programs developed with common practices, language features and tools
- An implementation[1] on top of the Clang [cla] C++ front-end for LLVM
- An AST Library for generative template metaprogramming in standard C++

In particular, MetaC++ introduces the following features to C++:

- Multi-stage programming where stage code can fully reuse the full C++ language
- Code as first class value using ASTs and corresponding AST composition tags
- `typename` and `template` keyword extensions for disambiguating unknown identifiers
- Integration of staging with existing compile-time evaluation features of C++, such as the preprocessor or templates
- Improved error reporting for C++ metaprograms

The rest of the paper is structured as follows. Section 2 provides background information related to ASTs, quasi-quoting and multi-stage languages. Section 3 introduces MetaC++, detailing its syntax and semantics, and discusses its integration with the normal language. Section 4 presents a case study for generative metaprogramming in standard C++ and compares it with our approach. Section 5 discusses selected application scenarios highlighting the software engineering value of our meta-language, while section 6 elaborates on implementation details. Finally, section 7 reviews related work, and section 8 summarizes and draws key conclusions.

[1] Our implementation is available at `https://github.com/meta-cpp/clang`

2 Background

2.1 ASTs and quasi-quotation

Metaprogramming involves generating, combining and transforming source code, so it is essential to provide a convenient way for expressing and manipulating source code fragments. Expressing source code directly as text is impractical for code traversal and manipulation, while intermediate or even target code representations are too low-level to be deployed. Currently, the standard method for representing and manipulating code is based on ASTs, a notion originating from the *s-expressions* [Ste90] of Lisp. Although ASTs provide an effective method for manipulating source code fragments, manually creating them usually requires a large amount of expressions or statements, making it hard to identify the actually represented source code [WC93]. Thus, ways to directly convert source text to ASTs and easily compose ASTs into more comprehensive source fragments were required. Both requirements have been addressed by existing languages through *quasi-quotation* [Baw99]. Normal quotation skips any evaluation, thus interpreting the original text as code. Quasi-quotation works on top of that, but instead of specifying the exact code structure, it essentially provides a source code template that can be filled with other code. To better illustrate this notion consider the following Lisp macro that generates the multiplication of the argument X by itself.

```
(defmacro square (X) '(* ,X ,X))
(square 5) ; expanded during macro expansion, yields 25
```

Definitions after the *backquote* operator ' are not directly evaluated but are interpreted as a code fragment (i.e. an AST). The *unquote* operator , operates in reverse, escaping the syntactic form and inserting its argument in the expression being created. This way, the invocation `(square 5)` creates the expression `(* 5 5)` that yields 25.

2.2 Multi-Stage Languages

Multi-stage languages extend the multi-level language [GJ95] notion of dividing a program into levels of evaluation by making them accessible to the programmer through special syntax called *staging annotations* [TS00]. Such annotations are used to specify the evaluation order of the various program computations. In this sense, a staged program is a conventional program that has been extended with the proper staging annotations. Here, we will use the term *stage code* or *meta-code* for code that is somehow characterized to be evaluated in a distinct execution stage. Then, the term *stage program* refers to the collection of *stage code* belonging to the same stage.

Staging was originally focused at runtime, where the main stage (i.e. the normal program) determines the "next" stage code to be evaluated during its execution. For example, consider the following *MetaML* code exhibiting the use of staging annotations.

```
val code = <5>;
val square = <~code * ~code>;
val result = run square; (* evaluated at runtime, yields 25 *)
```

Brackets `<_>` create delayed computations thus constructing code fragments (i.e. ASTs). Then, *escape* `~_` allows combining smaller delayed computations to construct larger ones by splicing its argument in the surrounding brackets (i.e. combines ASTs). Thus, the second assignment of the above code creates the delayed computation `<5*5>`. Finally, `run` evaluates the delayed computation in the current stage (i.e. performs code generation for the given AST), which in our example evaluates to 25.

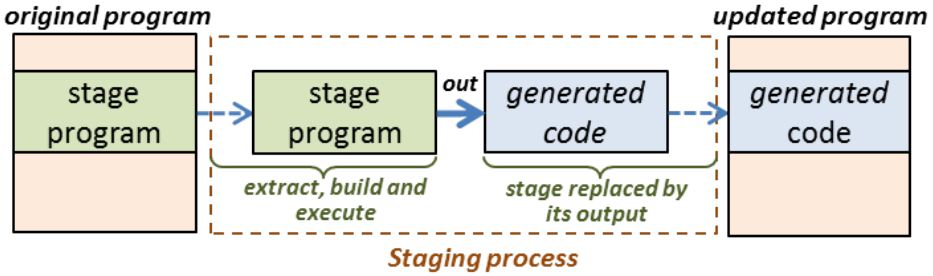

Figure 2 – Process of compile-time staging in multi-stage languages (only 2 stages shown)

Staging can also be applied during compilation, where "previous" stage code is evaluated at compile-time to change the main stage code. This notion is depicted in Figure 2, while an example written in *Template Haskell* [SJ02] is provided below.

```
square :: Expr -> Expr
square x = [| $x * $x |]
result = $(square [|5|]) -- evaluated at compile-time, yields 25
```

Quasi-quote brackets `[| _ |]` again create ASTs, while the splice annotation `$` plays a dual role; within quasi-quotes it combines ASTs, acting similar to *escape*, while outside of them it evaluates the expression and splices the result in its place, acting similar to `run`, but with its evaluation occurring during compilation.

3 Meta C++

MetaC++ is a multi-stage extension of C++ that adopts compile-time staging and the integrated metaprogramming model [LS15]. We continue by briefly introducing the programming model, outlining the staging syntax and semantics of our language and discussing C++ specific extensions required for generative metaprogramming.

3.1 Programming Model

Most multi-stage systems offer only the notion of staged expressions that are evaluated in isolation, separated from other code present in the same stage. There is neither the notion of a collective stage program, nor language support in the form of statements (e.g. assignments, control flow) or definitions (e.g. variables, functions, classes) to realize such a notion. Pure functional languages such as Template Haskell are stateless and can thus use definitions across stages, setting virtually no distinction between runtime and compile-time environments for code evaluation. When state is involved, there is a need for clearly separated stages, each with its own definitions and state.

In the integrated metaprogramming model, independent snippets of stage code at the same nesting, involving any language construct (e.g. expressions, statements, definitions), are concatenated following their order of appearance in the main source and treated as a *unified program*, with a lexically-scoped control flow, shared program state, and the scoping rules of the main language. The concatenated stage fragments may contain multiple code generation directives, so an integrated metaprogram behaves as having multiple input and output locations within its *enclosing program*. We use

the term enclosing program and not main program, as for nesting levels above one the resulting integrated metaprograms are hosted within other integrated metaprograms.

The integrated metaprogramming model compared to fragmented stage code reflects a fundamental methodological shift concerning transformations. In particular, we treat transformations as any other program function. Effectively, since stage fragments at the same nesting are related by transforming the same enclosing program, it seems an unreasonable decision to physically separate them into distinct programs or modules as it serves no particular goal and only complicates the engineering of metaprograms.

The target is to enable software engineering of metaprograms in a way similar to normal programs, adopting all normal language features and programming practices for their implementation. In this sense, a stage program in MetaC++ may be structured using functions, classes, modules and libraries, while performing operations like typical file I/O, network connections and communication, loading of DLLs, etc.

The way integrated stage programs are assembled and generate code for their enclosing program resembles HTML generating systems such as PHP, ASP and JSP. For example, in JSP, scripting elements placed alongside HTML code constitute meta-code to be inserted and executed in the JSP page's servlet class. However, such systems are heterogeneous (i.e. the metalanguage is different from the object language), operate on source text at a lexical level and support only one level of code generation, while MetaC++ is homogeneous (i.e. the metalanguage is the same as the object language), operates on code in AST form and supports multi-stage generation.

Examples exhibiting the integrated metaprogramming model are presented after first discussing the staging syntax and semantics, in section 3.4 and later in section 5.

3.2 AST Tags

Such tags allow converting source text into ASTs, *involve no staging*, and are translated into calls that create ASTs by parsing source text or combining other ASTs together.

Quasi-quotes (written `.<...>.`) may be inserted around language elements, such as class or function definitions, expressions, statements, etc., to convey their AST form and are used to create ASTs from source text. For instance, `.<1+2>.` represents the AST for the source text `1+2`. Quasi-quotes can be nested at any depth (AST representing other ASTs) to allow forms for multiple levels of staging. Identifiers within quasi-quotes are resolved in the context where the respective AST is inserted, while hygienic macros [KFFD86] are also supported through the notation `$id` that introduces contextually unique identifiers. Quasi-quotes may also include preprocessor directives to allow generating code containing `#define`, `#include`, etc. Such directives are not expanded within the quasi-quotes, but are treated as AST values.

Escape (written `.~(expr)`) is used only within quasi-quotes to prevent converting the source text of `expr` into an AST and evaluate it normally. Practically, escape is used on expressions already carrying AST values which need to be combined into an AST being constructed via quasi-quotes. For example, assuming `x` already carries the AST value of `.<1>.`, the expression `.<.~x+2>.` evaluates to `.<1+2>.`. We also support the escaped expression to carry a numeric, boolean or string value, in which case, the value is automatically converted to its corresponding AST value as if it were a constant. For instance, if `x` is `1`, then the expression `.<.~x+2>.` evaluates to `.<1+2>.`.

Quasi-quotes (and any escapes they contain) are translated into calls that create ASTs by parsing source text or combining other ASTs together. In particular, they are translated to calls of the internal compiler function `meta::quasiquotes`. For example, `.<.~x+.~y>.` is translated to `meta::quasiquotes(".~x+.~y", 2, x, y)`.

3.3 Staging Tags

Staging tags generally imply compile-time evaluation of the associated source code, and are essential in supporting staging. Syntactically, they define the boundaries between stage code fragments and introduce stage nesting.

Generate (written `.!(expr)`) is used for evaluating the stage expression `expr` and inserting its value (that must be of an AST type) into the enclosing program by replacing itself. It effectively performs in-place code generation, operating analogously to JSP expressions tags (<%= `expr` %>). Generate tags are allowed within quasi-quotes, but are just AST values that are not directly evaluated. This allows expressions carrying an AST with a generate tag to be inserted into the enclosing program, meaning generate tags may generate further generate tags, thus supporting meta-generators.

Execute (written `.&stmt`) defines a stage `stmt` representing any single statement, local definition or block in the language. It operates analogously to JSP scriptlet tags (<% `stmts` %>) that insert code into the service method of the JSP page's servlet. Any definitions introduced are visible only within stage code. *Execute* tags can also be nested (e.g. `.&.&stmt`), with their nesting depth specifying the exact stage program they will appear in. Additionally, *execute* tags can be quasi-quoted and be converted to AST form, meaning their generation will introduce further staging.

Define (written `.@defs`) introduces stage `defs`, which syntactically represent any global program unit in the language (e.g. function or class definitions, namespaces). It operates analogously to JSP declaration tags (<%! `decls` %>) that introduce declarations in the JSP page's servlet class, but also allows introducing classes and namespaces. Definitions introduced are visible only in stage code, while nested *define* tags (e.g. `.@.@def`), like nested *execute* tags, specify the stage the `defs` will appear in.

Preprocessor directives may appear after *execute* or *define* tags (e.g. `.@#include`, or `.&#define`) enabling file inclusion, macro definitions and conditional compilation in stage code. Stage preprocessor directives are expanded in the stage program and do not affect normal program code or interfere with normal preprocessor directives.

Below we show the typical staged power written in MetaC++. It is selected (and a bit verbose) to illustrate the syntax and semantics of our language. Motivating examples for generative metaprogramming using MetaC++ are presented in section 5.

```
.@AST* ExpandPower(unsigned n, AST* x) { //stage function definition
  if (n == 0) return .<1>.;
  else return .<.~x*.~(ExpandPower(n-1, x))>.;
}
.@AST* MakePower(unsigned n, AST* name){ //stage function definition
  AST* expr=ExpandPower(n,.<x>.);//call function of same stage nesting
  return .<int .~name (int x){ return .~expr; }>.;       //apart from
} //expressions,statements,declarations and names can also be escaped
.&AST* power = MakePower(3, .<power3>.); //stage variable declaration
.!(power);//generation directive accessing the stage variable power
//code generation result: int power3(int x) { return x*x*x*1; }
```

The adoption of three staging tags instead of the typical single code generation tag (e.g. MetaML `run` or Template Haskell splice) is essential to follow the integrated metaprogramming model. In particular, the *execute* and *define* tags play the role of stage statements and definitions and have nothing to do with code generation that is only performed through generate tags. For example, the code `.&f()` is different from `.!(f())` as the former only invokes the stage program function f (possibly

affecting global stage program data), while the latter invokes it and uses its result (presumably an AST value) to perform code generation. We could achieve some *execute* functionality using the *generate* form by performing the necessary action and returning an empty AST to be inserted, but that would require introducing extra stage functions to accommodate statements. More importantly though, without *execute* we would not be able to introduce local definitions for stages, such as variables, lambdas, and new local types (e.g. synonyms or classes). This is important as local stage program definitions serve the same role as local definitions in normal programs. Additionally, *execute* and *define* tags cannot be combined in a single overloaded tag that will operate differently based on its argument. The reason is that C++ allows both local and global definitions for various elements (e.g. variables, classes, types) so a single tag could not unambiguously differentiate between the two options and forcing one option over the other would limit expressiveness, as indicated by the code below.

```
.@AST* x = .<1>.;             //global stage variable declaration
.&AST* x = .<2>.;             //local stage variable declaration
int y = .!(.<.~(::x)+.~x>.); //accessing local&global stage variables
//code generation result: int y = 1 + 2;
```

The example may seem contrived but it is simply the staged equivalent of the following C++ code that uses a global variable and a local variable that shadows it.

```
int x = 1;
void f() { int x = 2; int y = ::x + x; } //y = 1 + 2
```

Moreover, *generate* tags and *escape* tags operate in a similar way, both taking AST values as arguments and inserting them at the context of use; for *generate* tags the context of use is the enclosing program, thus performing code generation, while for *escape* tags it is the enclosing quasi-quote, thus performing AST combination. If *generate* tags were not allowed within quasi-quotes, the two tags could be combined in an overloaded tag that would operate as an *escape* tag within quasi-quotes and as a *generate* tag outside of them. However, disallowing *generate* tags within quasi-quotes means offering no support for meta-generators, and thus limits expressiveness. We consider the extra syntax to be minimal and well-worth the added expressiveness of meta-generators, so we keep both tags with their originally discussed semantics.

3.4 Staging Loop

The staging loop takes place after the original source code has been parsed into an AST and is responsible to evaluate meta-code and produce a modified AST that consists of pure C++ code and can be normally compiled. Each stage program consists of code at the same stage nesting with their order of appearance in the main source, while the evaluation order of stages is inside-out, i.e. from most to least nested. Thus, the staging loop is repeated until no further stages exist and involves three steps: (i) determining the maximum stage nesting level; (ii) assembling the stage program for this nesting level; and (iii) building and executing the assembled stage program.

The maximum stage nesting is initially computed by traversing the AST and counting the encountered staging tags. This computation should be repeated at the beginning of every stage evaluation since the maximum stage nesting may be increased if the evaluation of the last stage has generated further meta-code. Then, we perform a depth-first traversal to collect the AST nodes representing code located under staging tags at the maximum nesting. For example, in the original source of Figure 3, only

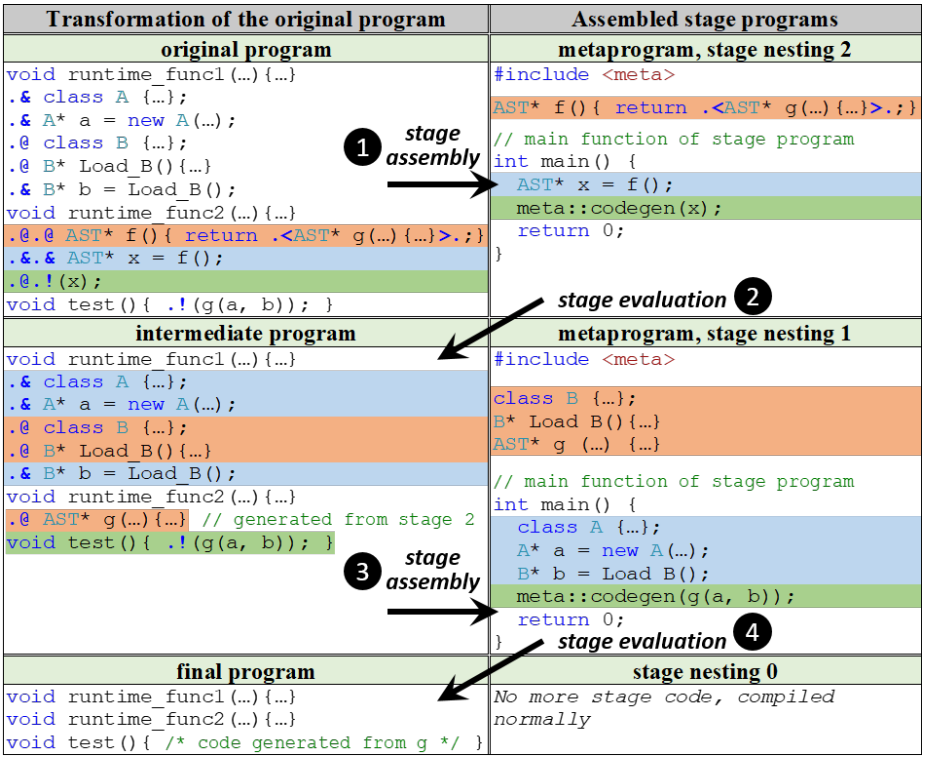

Figure 3 – Staging loop example with two stages showing stage assembly and evaluation

nodes under staging tags of nesting 2 (*top left*, highlighted) are considered for stage assembly. These nodes are then used to generate the stage program code. Nodes from *execute* and *define* tags are used as they are while pruning their staging tag node from the main program AST, i.e. they are consumed by the stage they target and are not available in any other stage (even stages introduced by meta-generators). For nodes originating from generate tags, apart from the associated expr, we also need a special invocation that will internally handle the required AST modification, replacing the generate node with the result of its evaluation. For this purpose we deploy a library function offered by our meta-compiler (`meta::codegen`) that is linked only in stage programs. For example, the `.!(x)` expression of the original source of Figure 3 leads to the `meta::codegen(x)` invocation in the stage program of nesting 2 (*top right*) that in turn generates the `AST* g(...){...}` function for stage 1 (*middle left*).

The collected AST nodes are then assembled to form the stage program (Figure 3, *stage assembly* arrows numbered 1 and 3). Code from *define* tags represents global definitions and declarations so it is placed in global scope, while code from *execute* and *generate* tags essentially constitutes the executable part of the stage program so it is placed within the body of a generated main function (the main function of the stage program that is unrelated with the main function of the normal program). In both cases, the assembled code fragments preserve their relative order of appearance in the original source text so as to follow standard C++ scoping rules.

Selecting the stage code of the maximum nesting level at each iteration, yields an assembled stage program that contains no meta-code. Additionally, both quasi-quoting and code generation functionality is handled through internal meta-compiler

library functions, ensuring that metaprograms are syntactically just standard C++ programs that use the meta-compiler as part of their execution environment. Finally, the executable part of the stage code is placed within a main function just to make the stage program a normal executable C++ program. Examples of stage programs, assembled by the MetaC++ compiler, are shown in the right part of Figure 3. Overall, stage programs can be compiled using the original language compiler and executed using the original language runtime environment.

When a stage program is executed (Figure 3, *stage evaluation* arrows numbered 2 and 4), it affects the original program through the `meta::codegen` calls, transforming its AST into a modified version, called *intermediate program*, that is used for the next staging loop iteration. Eventually, when an intermediate program contains no more stage code it constitutes the *final program* (Figure 3, *bottom left*) that is the result of the staging loop and is then compiled using the original language compiler.

3.5 Extended Syntax Disambiguation

In the context of generative metaprogramming, it is common to encounter unknown identifiers without resulting to invalid code. For example, quasi-quoted code that will be inserted at some source location will typically involve identifiers (e.g. types or variables) that are visible at that source location. However, the quasi-quotes themselves may syntactically reside in an entirely unrelated declaration context or scope, causing the used identifiers to be unknown within quasi-quotes. The same applies for normal program code that refers to identifiers generated by meta-code. Conceptually, identifiers introduced by a generate tag should be visible to subsequent code as if they were part of the original source; however, syntactically they do not exist prior to stage execution, resulting in unknown identifiers.

For C++, the latter poses a significant challenge as its context-sensitive grammar does not allow unambiguously parsing code with unknown identifiers. In particular, there may be different ways to parse a code segment based on whether an unknown identifier names a variable, a type or a template, as shown in the following example.

```
X * x;
A < B > c;
```

In the first line, if `X` refers to a type, then the statement declares a variable named `x` with type pointer to `X`, otherwise it is a multiplication between variables `X` and `x`. In the second line, if `A` is a class template then the statement declares a variable `c` of type `A<B>`, while if all identifiers are variables we have a weird yet valid expression that tests if `A` is less than `B` and then if the result is greater than `c`.

Standard C++ has a similar issue with dependent names [Cpp] within templates, where types and expressions may depend on template parameters types. We revisit the above example for code present within a template that has a type parameter `T`.

```
T::X * x;
T::A < T::B > c;
```

Since `T` is a template parameter, we do not know if `T::X`, `T::A` and `T::B` name variables, types or templates, again resulting in ambiguous parsing. C++ solves this problem by allowing the programmer to explicitly disambiguate the intended use through the `typename` and `template` keywords. As shown below, using `typename` treats the qualified identifier as a type, using `template` treats it as a template, while using no additional keyword treats it as a variable.

```
typename T::X * x;               //X and B are types
T::template A<typename T::B> c; //A is a template
```

In our work, we extend the use of the `typename` and template `keywords` to be valid even for unqualified identifiers, semantically referring to type and template definitions that will be available after the staging process (e.g. they may be generated by meta-code). An unknown identifier is considered to be a type if prefixed by `typename`, a `template`, if prefixed by template, or a variable otherwise. In this sense, our original example contains a multiplication and a series of comparison operations. If we wanted the code to express variable declarations, we would instead write:

```
typename X * x;             //X is a type
::template A<typename B> c; //A is a template and B is a type
```

Apart from referring to unknown identifiers, this syntax is actually valid standard C++; since C++11 both keywords can be used outside of templates, while since C++17 the `template` disambiguator is allowed if the left part of the scope resolution operator refers to a namespace [Cpp], which in the last example is the global namespace.

Another extension for the `typename` keyword is that we allow it to appear in the context of a constructor initializer list to disambiguate between member initializers and base class initializers. In standard C++, a dependent name appearing within a constructor initializer list may only refer to a base class, so it is implicitly a type and there is no need for disambiguation. However, in the context of metaprogramming, an unknown identifier within a constructor initializer list may refer either to an unknown base class or an unknown class member. Using the `typename` keyword the identifier is treated as a type, denoting a base class initializer, otherwise the identifier is treated as a variable, denoting a member initializer. For instance, the following AST represents a constructor definition of class (or struct) `X` that has a base class `A` and a member `b`.

```
.<typename X() : typename A(), b() {}>.
```

3.6 Integrating with Compile-Time Evaluation Features of C++

It is important for the staging infrastructure to integrate well with other compile-time evaluation features of C++, as they also support metaprogramming. In particular, we consider the integration of our staging infrastructure with the preprocessor, the template system, the `constexpr` specifier and the `static_assert` declarations.

3.6.1 Preprocessor

As previously discussed, the staging process takes place after parsing the original source code. This would imply that any preprocessor directives encountered in the source text have already been taken into account and expanded accordingly. However, invoking the preprocessor for the entire source file without taking the staging process into consideration may be problematic. For instance, we cannot include a header file (e.g. a standard library header) in both normal and stage code as the second inclusion would be skipped due to the conditional compilation guards (`#ifndef` - `#define` - `#endif`) of the header file. Instead, a *staging-aware* preprocessing step is required in which every source code fragment is aware of its stage nesting and takes into account only directives declared for that particular stage nesting. This means that a stage program may freely use preprocessor directives without interfering with other stages or the main program, perfectly aligning with the intent to support stages with all

normal programming features of the language. Overall, the result of staging-aware preprocessing encompasses the changes introduced by preprocessor directives (i.e. included files, conditional compilation and macro expansions) for all stages, contains no further preprocessor directives, and is the source code given as input to the parser.

Integrating the staging infrastructure with the preprocessor also requires supporting meta-code in header files. A typical scenario is generating a class definition that needs to be included in several source files. When such a header file is included in a source file, any stage definitions it contains will become part of the source file code and thus be taken into account in the staging process. Naturally, we expect any code generated this way to remain the same across different inclusions of the header file, ensuring that source files using it end up with a consistent view. Including a header file that contains meta-code is even possible from within a stage directive; the staging tag associated with the include directive is applied on all definitions included from the header file, increasing their stage nesting by 1, thus maintaining their evaluation order in the staging process. This is shown in Figure 4 where the original program includes

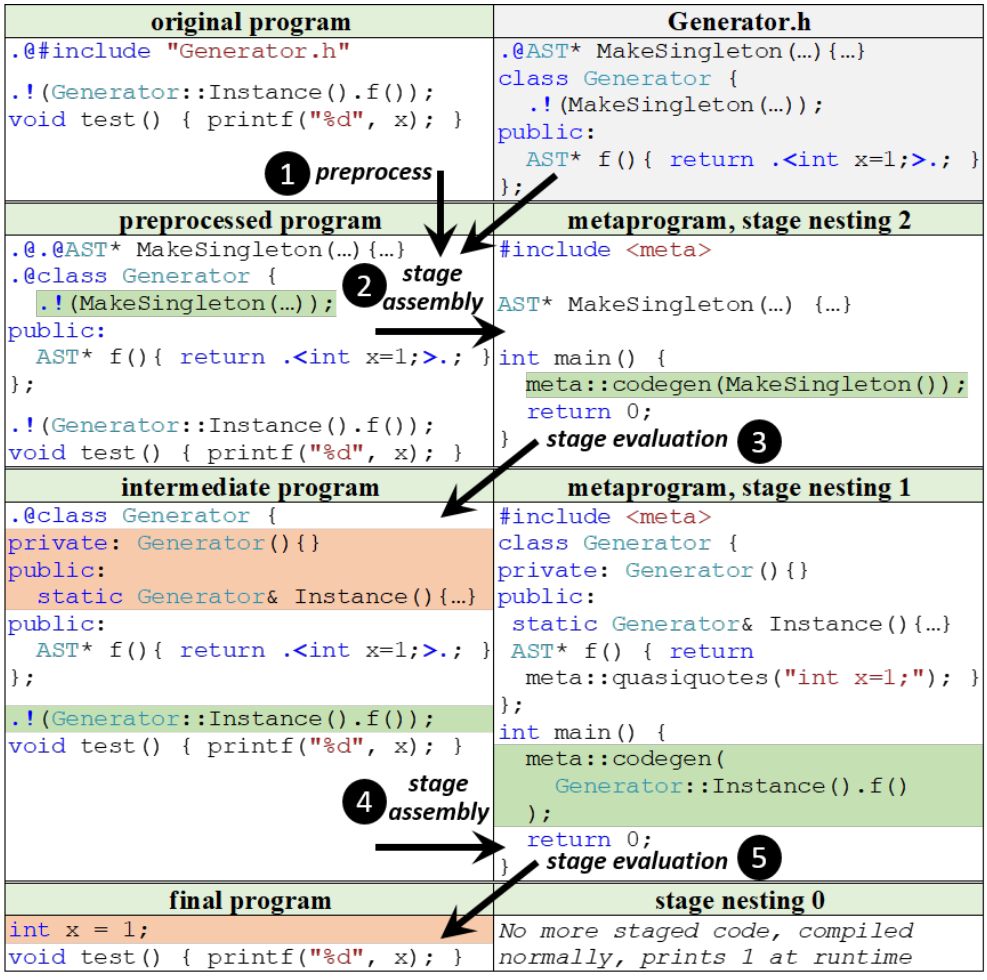

Figure 4 – Stage assembly and evaluation example involving stage preprocessor directives and headers with meta-code; generation directives and their outcomes are highlighted

the header file *Generator.h* within stage code and uses the `Generator` class for code generation. Code within *Generator.h* also uses staging to turn the Generator class into a singleton using the stage function `MakeSingleton`. Thus, the preprocessed program has a maximum stage nesting of 2 and involves two stages before it takes its final form.

3.6.2 Templates

The staging process takes places before template instantiation, so any template definitions are present in the program AST and can be considered for use in stage code. Stage template definitions will become part of some stage program, and they will be instantiated as a normal part of that stage program's translation, without requiring any further action by the staging system. Additionally, template definitions or arguments for template instantiation may be generated by a metaprogram. Such generated code becomes part of the main program AST and any corresponding instantiations are handled later as part of the normal final program compilation.

As part of integrating staging with the template system, we also consider variadic templates and parameter packs in particular. Staging tags within a template definition are always at a different stage nesting and therefore cannot interact with parameter packs. On the other hand, AST tags do not involve staging and may interact with parameter packs of a variadic template, as shown by the code below.

```
template<typename... T> void a (T... t) {
  f(.<1 + .~t>. ...); //quasi-quote pattern
  g(.<.~(h(t))...>.); //escape pattern
}
```

During translation, the call `a(.<2>., .<3>.)` will instantiate the template, expanding the parameter packs as if the code was originally written as `f(.<1+2>., .<1+3>.)` and `g(.<.~(h(.<2>.)), .~(h(.<3>.))>.)`. Supporting this functionality requires some extra handling due to quasi-quotes and escapes being translated to internal compiler function calls that parse source text. This source text should be available when performing the stage assembly, however, the parameter pack expansion occurs later, during template instantiation. To resolve this issue, we deploy another internal compiler function, called `meta::escape_pack`, responsible for retrieving the size of the pack, and the pack itself during its execution, in order to generate the associated source text based on the given pattern and finally parse that text to AST. The name of the parameter pack is available during stage assembly, so it is possible to generate code for both retrieving the size of the pack through the `sizeof...` operator and expanding the pack in the context of the `meta::escape_pack` call. For instance, in the above example the quasi-quoted code `.<.~(h(t))...>.` is translated to `meta::quasiquotes(".~(h(t))...", 1, meta::escape_pack("(h(t))", sizeof...(t), t...))`. For the example invocation, the inner call to `meta::escape_pack` becomes `meta::escape_pack("(h(t))", 2, .<2>., .<3>.)` that will internally call `meta::quasiquotes(".~(h(t)), .~(h(t))", 2, .<2>., .<3>.)` to generate the desired AST.

3.6.3 *constexpr* and *static_assert*

The `constexpr` specifier declares functions or variables that can be evaluated at compile time and thus be used in a context requiring a compile-time constant expression. Due to the adoption of the integrated metaprogramming model, `constexpr` is orthogonal to the staging infrastructure. A `constexpr` specifier located within meta-code will simply become part of some stage program and will enable its compile-time evaluation

during the compilation of that stage program. Similarly, any `constexpr` specifiers located within normal code or introduced through code generation will be part of the final program and will be available for compile-time evaluation during its compilation.

`static_assert` declarations can be used for compile-time assertion checking. For meta-code using `static_assert` declarations checking occurs normally during the compilation of the stage, while for generated `static_assert` declarations it occurs during the compilation of the transformed main program. `static_assert` declarations within templates are always checked upon instantiation. For any other `static_assert` declarations, i.e. found in non-stage and non-dependent contexts, checking occurs during parsing of the original program if the associated expression involves no staging or unknown identifiers, otherwise it is deferred for the compilation of the final program.

3.7 Compile Error Reporting for Metaprograms and Generated Programs

A source of criticism related to templates involves their cryptic error messages that may require digging across several levels of instantiations to locate the error cause.

MetaC++ offers improved compile-error reporting for metaprograms by adopting the techniques discussed in [LS13]. In particular, the AST of every stage program along with the updated version of the main AST it produces are unparsed to generate source code files that are stored as additional output files of the compilation, effectively providing a glass-box view of the staging process. Then, to provide a precise error report, the compiler maintains for AST nodes information about their origin and uses it to track down the error across all involved stages and outputs, creating a chain of source reference information that supplements the normal compilation error message. The message itself is unchanged as it constitutes an error of normal C++ code (either stage program or the final program) and is not related to the staging process. The additional error chain across all stages and outputs provides the *missing information context* of the staging process required to fully understand the error report.

3.8 Discussion

A critical decision in the design of MetaC++ involved the tradeoff between type safety and expressiveness. In traditional multi-stage languages like MetaML, code generation occurs at runtime, at which point it is too late to report type errors, so such languages sacrifice some expressiveness and perform static type-checking to guarantee well-formedness of all generated code. A similar approach would also be beneficial in a compile-time context, enabling to type-check metaprograms independently of their usages. However, the complexity of C++ would make such an approach impossible or impractical to adopt even for expressions, let alone statements and declarations, without greatly limiting expressiveness. Also, since code is generated during compilation, any type errors will be caught during type-checking of the assembled stage programs or the final program and be reported as compile errors. Thus, we focused on expressiveness and chose to allow generating any language construct at the cost of dropping strong type-safety guaranties. Essentially, we followed the same path C++ takes with its templates that are type-checked late, at instantiation time.

With type-safety not being critical, we further chose to simplify AST usage by adopting a uniform AST type instead of having an AST type hierarchy that reflects and enforces AST usage based on the source location (e.g. ExprAST, StmtAST, TopLevelDeclarationAST, LocalDeclarationAST, ClassMemberDeclarationAST, etc.). This enables quasi-quotes to be less verbose, avoiding any extra syntax that would be

required to disambiguate between different uses of a specific code form. For example, consider referring to a variable declaration such as `int x;` that may appear in global context, local context or within a class body. If we used multiple AST types we would require extra syntax to distinguish between the possible cases, e.g. have the following quasi-quotes to match each case: `.<globalDecl:int x;>.`, `.<localDecl:int x;>.`, `.<classMember:int x;>.`, each producing a different AST type (TopLevelDeclarationAST, LocalDeclarationAST and ClassMemberDeclarationAST respectively). With the uniform AST type approach, we have a single quoted declaration `.<int x;>.` that accommodates all possible declaration contexts. Additionally, enforcing typed AST usage based on the source location would limit expressiveness as some entities can only appear in specific contexts. For example, a DeclarationAST node could involve a function declaration that would be invalid to be used within a block. Instead, the unified AST allows generic code forms that can be deployed in multiple source locations. On the other hand, a single AST type further reduces type-safety as we cannot statically determine if a used AST will generate invalid code, e.g. using a quoted declaration at an expression context. Nevertheless, type information about an AST value is available during metaprogram execution, so the metaprogram logic may consider it to avoid generating ill-formed code. Also, even if erroneous code is generated, as discussed, any errors will still be caught at compile-time when parsing and type-checking the generated code, and eventually be reported as compile errors.

Another design decision relates to macro hygiene. Most metalanguages offer hygienic behavior by default, while enabling explicit name capture through special syntax. We have purposefully chosen an inverse activation policy since we consider it to be a better fit in the context of generative metaprogramming. In particular, many metaprogram scenarios involve generating complete named element definitions such as classes, functions, methods, constants, namespaces and generics as well as code that uses existing definitions (e.g. code that uses an STL algorithm or container). In all these cases, the supplied name has to be directly used for deployment, thus name capture is the only way. Also, when generating non-template code fragments that may be further combined, any name clashes or inadvertent variable captures can be easily avoided in the respective generator by enclosing any statements in blocks and declaring generated variable properly so as to shadow any prior declarations. The only scenario where undesirable name capture may occur involves template code fragments that will be filled-in with other code fragments, where the inserted code fragments may undesirably capture names in the template itself. This is the only case where the template generator should force hygiene for template variables. Overall, we considered that for most scenarios name capture would suffice, so we made the common case less verbose, while also offering extra syntax to enforce hygiene where necessary.

4 Examining Generative Metaprogramming in Standard C++

In C++, the only form of compile-time computation and composition that can be exercised to allow generative metaprogramming is through templates. In this context, before exploring the chances for a multi-stage language extension, we aimed to support generative metaprogramming directly in C++ by practicing compile-time code manipulation through template metaprogramming.

Effectively, we had to enable the writing of compile-time evaluated code that could somehow produce code that is executed as part of the normal runtime control flow. Now, the resulting source code cannot be composed in the form of typical source text

since templates offer no capability to apply in-place source text insertion.

Moreover, we had to enable the typical manipulation of source fragments through ASTs; however, template metaprograms do not offer a built-in notion of code expressed in AST form. Thus we explicitly introduced this notion by developing an AST template metaprogramming library. It is critical to emphasize that a compile-time library is necessary to guarantee that no runtime overhead is introduced, the latter being a fundamental property of compile-time metaprograms.

Because such ASTs should be created and manipulated during compilation, they must be modelled as types that incorporate any required data as nested types or `const` definitions. Additionally, they require other ASTs, i.e. other types, as construction parameters, meaning they must be implemented as templates. Finally, to provide an analogy of the in-place code insertion for composed ASTs we introduced an `eval` function whose code is recursively composed at compile-time via template instantiation, while at runtime evaluates precisely the respective composed AST. The following code outlines some indicative AST node classes (i.e. template declarations) and illustrates how the AST template metaprogramming library can create AST node instances (i.e. template instantiations) and use them for code composition.

```
//AST nodes for a constant integer, if statement and add expression
template<int val> struct const_int
  { static int eval(...) { return val; } };
template<typename Cond, typename Stmt> struct if_stmt {
  template<typename... Args> static void eval(Args... args)
    { if (Cond::eval(args...)) Stmt::eval(args...); }
};
template<typename Left, typename Right> struct add_expr {
  template<typename... Args> static decltype(auto) eval(Args... args)
    { return Left::eval(args...) + Right::eval(args...); }
};
//Code below represents the (contrived) AST if (1) 2 + 3;
using Code=if_stmt<const_int<1>,add_expr<const_int<2>,const_int<3>>>;
void test(){ Code::eval(); } //compile-time code generation of eval
```

Supporting such AST functionality is a challenging endeavor even for a limited set of the language constructs (in our library implementation[2] we focused on a C with Classes subset), but once implemented, its adoption for creating AST values is just a matter of instantiating template classes with appropriate arguments. Using the metaparse [SP12] library, we could further automate the appropriate AST instantiations based on compile-time strings providing a more natural syntax. For instance, the AST of the previous example could be written as `ast<_S("if(1)2+3;")>::type`.

With the AST library available, and template metaprograms being able to express any AST computation (they are Turing-Complete [4]), it is computationally possible to express any generative metaprogram. This, however, has little software engineering value, as the most important criterion is not the feasibility of the approach but the implementation complexity it involves. In this context, it became clear that hiding the advanced template metaprogramming techniques used to implement ASTs within library code was not sufficient, as similar techniques would be required by the client programmer to implement custom AST transformations. This is illustrated below with the supposedly simple example of merging statement ASTs into a block.

[2]Our AST metaprogramming library is available at `https://github.com/meta-cpp/meta-ast`

```
#include <meta_ast.hpp> //defs below are part of the library header
template<typename... Stmts> struct block {/*...*/};
template<typename F, typename...Args> struct call {/*...*/};
template<typename T, T Func> struct cfunc {/*...*/};
#define CFUNC(f) cfunc<add_pointer_t<decltype(f)>, f>
template <char... chars> struct String {/*...*/};
//client code begins here
template<typename... Ts> struct TypeVec { //compile-time vector
 using impl = std::tuple<Ts...>;
 template<int N> using at=typename std::tuple_element<N-1,impl>::type;
 static const int size = sizeof...(Ts);
};
template<template<typename...> class Seq, typename Stmts>
struct MergeStmts { //meta-function implementation
 //struct AppendToSeq used for appending to a compile-time sequence
 template<typename Old, typename New> struct AppendToSeq;
 template<template<typename...>class Seq,typename New,typename... Old>
 struct AppendToSeq<Seq<Old...>, New> { using type=Seq<Old...,New>; };
 //struct loop used for iterating over a sequence at compile-time
 template<template<typename...>class Seq, int N> struct loop {
   using type = typename AppendToSeq<typename loop<Seq, N - 1>::type,
                                     typename Stmts::template at<N> >::type;
 };
 template<template<typename...> class Seq>
 struct loop<Seq, 0> { using type = Seq<>; };
 using type = typename loop<Seq, Stmts::size>::type; //for invocation
};
using Code = MergeStmts<block, TypeVec< //meta-function invocation
 call<CFUNC(printf), String<'F', 'o', 'o', ' '>>,
 call<CFUNC(printf), String<'b', 'a', 'r'>>
>>::type; //Code represents: { printf("Foo "); printf("bar"); }
void test() { Code::eval(); }
```

Even such a simple task requires variadic templates, template template parameters and recursive template specializations to be used in client code. Conversely, the same example is straightforward in MetaC++, using just a loop over a standard container.

```
.@#include <vector> //include the std vector header in meta-code
.@AST* MergeStmts(const std::vector<AST*>& stmts) {
  AST* result=nullptr; //AST for resulting statements, initially empty
  for (AST* stmt : stmts) //iterate over all statements
    result = .<.~result; .~stmt;>.; //merge statements in a list
  return .<{.~result;}>.;//create block with the merged statement list
}
void test(){.!(MergeStmts({.<printf("Foo ");>.,.<printf("bar");>.}));}
//code generation result: void test(){{printf("Foo ");printf("bar");}}
```

Comparing the two approaches exemplifies our original statement about template metaprograms bearing little resemblance to normal programs, involving different programming approaches and disabling reuse. It also justifies our decision to abandon attempts for a pure C++ approach and focus on a multi-stage language extension.

5 Detailed Case Studies

We present various application scenarios highlighting the importance of extending C++ with generative metaprogramming features. In particular, we discuss adopting compile-time reflection to generate desired code structures, enriching client code with exception handling based on custom exception policies, and generating concrete design pattern implementations. These scenarios are very important as achieving similar functionality in standard C++, if all possible, would involve a combination of preprocessor and template metaprogramming tricks, resulting in code that is difficult to write, understand and maintain. Some scenarios involve directly invoking meta-functions for code generation, while others rely on the integrated metaprogramming model and utilize basic object-oriented features like encapsulation, abstraction and separation of concerns. Scenarios in the second category have been earlier introduced in [LS15] as key benchmarks to assess the expressive power of metaprogramming systems, emphasizing the importance of engineering stage programs like normal programs. We briefly revisit these scenarios and elaborate on their implementation in MetaC++.

Notice that the presented scenarios mostly focus on code generation involving a single stage. Since in the integrated metaprogramming model metaprograms are essentially normal programs, code generation naturally generalizes to multiple stages if the stage code is itself subject to code generation. A representative example falling in this category was already presented in Figure 4 with a stage generator class that was turned into a singleton through further metaprogramming.

5.1 Compile-Time Reflection

Compile-time reflection is a significant feature considered for an upcoming C++ standard. The reflection study group of the C++ standards committee has issued a call for compile-time reflection proposals [SC13] identifying four broad areas where reflection would be useful in C++ and choosing a representative use-case for each area. We show how these use-cases can be implemented in MetaC++. In the code, functions within the meta namespace are offered by the meta-compiler to export its internal data structures (e.g. `meta::getClassDecl`) and support creating ASTs from strings (e.g. `meta::id("x")` creates the AST `.<x>.`).

5.1.1 Generating Equality Operators

Equality operators typically perform member-wise equality checks between two objects. We use the context-aware `meta::getDeclContext` function to retrieve the compiler data structure for the current class definition. This can then be used to iterate over the class members and generate the desired equality checks, as show in the code below. Notice that variable `expr`, initially holding the AST of constant boolean expression `true`, at each iteration combines its previous value with the check for the current member, effectively accumulating equality checks for all members.

```
.@AST* genEqualityOp(const ClassDecl& Class) {
 AST* expr=.<true>.;//empty check to be combined with the && operator
 for (auto& field : Class.fields()) { //iterate over all class fields
  AST* id = meta::id(field.getName());
  expr = .<.~expr && .~id==rhs..~id>.; //merge with current == check
 }
 AST* classId = meta::id(Class.getName());
```

```
 return .< bool operator==(const typename .~classId& rhs) const
           { return .~expr; } >.;
}
class Point {
  int x, y;
  .!(genEqualityOp(*meta::getDeclContext())); //generates the code:
  // bool operator==(const Point& rhs) const
  //  { return true && x==rhs.x && y==rhs.y; }
}
```

5.1.2 Struct-of-Arrays Vector

Having a collection of ordered instances of a structure can be represented either as an *array of structs* or as a *struct of arrays*. The latter data layout is preferable in some applications for performance reasons. Using compile-time reflection, we can generate a *struct of arrays* structure for a given struct, potentially adding extra functionality to offer an array like interface. As shown in the following code, we use `meta::getClassDecl` to obtain the compiler data structure for the target struct and then iterate over its members to generate the desired code pattern.

```
.@AST* genSOAVector(const ClassDecl& Class, AST* name) {
 AST* members = nullptr; //AST for generated struct, initially empty
 AST* initList = nullptr;//AST for the initializer list in operator[]
 for (auto& field : Class.fields()) { //iterate over all class fields
  AST* type = meta::type(field.getType());
  AST* id = meta::id(field.getName() + "s");
  AST* member = .< std::template vector<.~type> .~id; >.;
  members=.<.~members;.~member;>.; //merge members with current
  initList=.<.~initList,.~id[i]>.; //accumulate the initializater list
 }
 AST* id = meta::id(Class.getName());
 AST* indexOperator =  //AST with the operator[] implementation
  .<typename .~id operator[](size_t i) const {return {.~initList};}>.;
 return .< struct .~name {.~members; .~indexOperator; }; >.;
}
struct S { int a, b; };
.!(genSOAVector(*meta::getClassDecl("S"), .<typename SoA_S_vector>.));
// struct SoA_S_vector {
//   std::vector<int> as; std::vector<int> bs;
//   S operator[](size_t i) const { return { as[i], bs[i] }; }
// };
```

5.1.3 Replacing Assert

Providing access to compile-time context information (e.g. file name and line number) without using preprocessor macros is possible (but more syntactically verbose) if we wrap the code with quasi-quotes and a generation tag. This is shown in the following code where the meta-function `generate_assert` will extract the compile-time context information of its AST argument to generate a function call that incorporates all necessary runtime information. The generated function call will typically refer to a

library implementation (e.g. a standard library `assertion` function), or it could refer to a custom user-defined assert function, that is passed as an argument to `genAssert`.

```
.@AST* genAssert(AST* expr, AST* assertFunc = .<std::assertion>.){
 if(!expr||!expr->isa<Expr>) //raise error for non-expression ASTs
   { meta::error("expected expression AST"); return nullptr; }
 else //generate the assert call with compile-time context information
   return .<.~assertFunc(.~expr, .~(to_string(expr)),
    .~(expr->getFile()), .~(expr->getLine()))>.;
}
namespace std {//normal function assumed to be part of the std library
  void assertion(bool expr, string str, string file, unsigned line) {
    if (!expr) { cerr << "Assertion failed: " << str << ", file " <<
                      file << ", line " << line << endl;
                 abort();
    }
  }
}
.!(genAssert(.<sizeof(int)==4>.));//instead of assert(sizeof(int)==4);
// std::assertion(sizeof(int)==4,"sizeof(int)==4","examples.cpp",80);
```

5.1.4 Enumerating Enums

Having access to the compiler data structures for a target enum through the `meta::getEnumDecl` function, it is straightforward to implement features like enum-to-string conversions, string-to-enum conversions or checked int-to-enum conversion without involving intrusive changes to the enum declaration or requiring duplicating information. For example, the following code shows how to generate an enum-to-string function.

```
.@AST* generateEnumToString(const EnumDecl& Enum, AST* name) {
  AST* cases = nullptr; //AST for case entries, initially empty
  for (auto& field : Enum.fields()) { //iterate over all enum fields
    AST* id = meta::id(field.getName());
    cases = .<.~cases; case .~id: return .~(field.getName());>.;
  } //create a case for each field and merge with previous cases
  AST* enumId = meta::id(Enum.getName());
  return .< std::string .~name (typename .~enumId v)
              { switch(v) { .~cases; default: return ""; } } >.;
}
enum Difficulty { Easy, Hard };
.!(generateEnumToString(*meta::getEnumDecl("Difficulty"),.<to_str>.));
// std::string to_str(Difficulty v) {
//   switch(v) { case Easy:   return "Easy";
//               case Hard:   return "Hard";
//               default:     return "";   }
// }
```

5.2 Exception Handling

As discussed in [LS12], compile-time metaprogramming can be used to implement exception handling patterns. This is achieved by adopting meta-functions capable of

generating the appropriate exception handling layout, and invoking them at compile-time with the desirable parameters to generate a concrete instantiation of the exception handling pattern. With our language, it is further possible to maintain a collection of the available exception handling patterns and select the appropriate one based on configuration parameters or the metaprogram control flow, while requiring no changes at the call sites within client code. This is illustrated in the following example.

```
.@using ExceptionPolicy = std::function<AST*(AST*)>;//policy prototype
.@AST* Logging(AST* code) //a logging policy: just log any exceptions
  { return .< try {.~code;} catch(std::exception& e) {log(e);} >.; }
.@struct RetryData {    //data for a retry meta-function policy:
  unsigned attempts;    //retry a number of times
  unsigned delay;       //wait for a given delay time between attempts
  AST* failureCode;     //execute arbitrary code if all attempts fail
};
.@ExceptionPolicy CreateRetry(const RetryData& data) { //retry creator
 return [data](AST* code){  //return a lambda implementing the pattern
  return .<        //the lambda returns an AST with the following code
   unsigned i;
   for (i = 0; i < .~(data.attempts); ++i)
    try { .~code; break; }     //execute the code and break on success
    catch(...){ Sleep(.~(data.delay)); }//catch exceptions, wait&retry
   if (i == .~(data.attempts)) { .~(data.failureCode); }   //after max
  >.;                                      //attempts run failure code
 };
}
.@class ExceptionPolicies { //compile-time class holding the policies
 static std::map<std::string, ExceptionPolicy> policies;
 static std::string policy;
public:
 static void Install(std::string p, ExceptionPolicy f){policies[p]=f;}
 static void SetActive(std::string p) { policy = p; }
 //create AST with the exception handling code for the active policy
 static AST* Apply(AST* code) { return (policies[policy])(code); }
};
.&ExceptionPolicies::Install("LOG", Logging); //install logging policy
.&ExceptionPolicies::Install("RETRY",    //create and install a custom
  CreateRetry({5, 1000, .<std::cerr << "fail";>.}));    //retry policy
.&ExceptionPolicies::SetActive("RETRY");   //set initial active policy
.!(ExceptionPolicies::Apply(.<f()>.)); //generates the code below
// unsigned i;
// for (i = 0; i < 5; ++i)
//   try { f(); break;  } catch(...) { Sleep(1000); }
// if (i == 5) { std::cerr << "fail"; }
.&ExceptionPolicies::SetActive("LOG"); //change active policy
.!(ExceptionPolicies::Apply(.<g()>.)); //generates the code below:
// try { g(); } catch e { log(e); }
```

We utilize a meta-code class `ExceptionPolicies` in an object-oriented fashion to hold and compose exception handling policies. We initially install a number of required policies, such as `LOG` and `RETRY`, and then generate the respective exception

handling code through invocations of the `Apply` function. In the example, `Logging` is directly a policy meta-function, while `CreateRetry` uses its supplied data parameters to offer the policy meta-function through a lambda function. Such parameters are provided once, upon policy installation, and are not repeated per policy deployment, relieving programmers from repeatedly supplying them at call sites. Most importantly, it allows a uniform invocation style, enabling different policies to be activated wherever required, without inherent changes at the generation sites.

5.3 Design Patterns

Design patterns [GHJV95] constitute generic reusable solutions to commonly recurring problems. They are not reusable modules, but recipes for applying solutions to a given problem in different situations. This means that in general, a pattern has to be implemented from scratch each time deployed, thus emphasizing design reuse as opposed to source code reuse. In this context, metaprogramming can support generating pattern implementations. Essentially, the pattern skeleton is turned into composition of ASTs, the pattern instantiation options become composition arguments, the actual client code is supplied in AST form and the pattern instantiation is handled by code generation directives. Effectively accommodating such requirements, requires features beyond stage expressions. With MetaC+,+ we can apply practices like encapsulation, abstraction and separation of concerns, thus greatly improving metaprogram development. For example, we can implement abstract pattern generators, have multiple such objects or even hierarchies of them available, and select the appropriate generator for a target context via a uniform invocation style. This is shown in the excerpt below (full code in Appendix B) with parameterized meta-code for generating adapter patterns.

```
.@class Adapter { //adapter pattern generator interface
protected: const ast::ClassDecl& Class;
public:
 using AdapterMap = std::map<std::string, std::string>;
 virtual AST* adapt(AST* name, const AdapterMap& renames) const = 0;
 Adapter(const ast::ClassDecl& Class) : Class(Class) {}
};
.@class AdapterByDelegation : public Adapter {//pattern implementation
 AST* MakeMethods() const;       //skipped for brevity, see Appendix B
public:
 AST* adapt(AST* newId, const AdapterMap& renames) const override {
   AST* classId = meta::id(Class.getName());
   return .< class .~newId {  //new class based on adapted methods AST
              typename .~classId* instance;         //adapted instance
             public:   .~(MakeMethods());     //insert adapted methods
              typename .~newId (typename .~classId* o) : instance(o){}
             };>.; //constructor with the adapted instance as argument
 }
 AdapterByDelegation(const ast::ClassDecl& Class) : Adapter(Class) {}
};
.@class AdapterBySubclassing:public Adapter {/*skipped for brevity*/};
class Window { public: int  Draw (DC& dc) {/*...*/}
                       void SetWholeScreen (void) {/*...*/}
}; //this is the runtime class to be adapted
```

```
.&Adapter::AdapterMap renames{ {"SetWholeScreen", "Maximize"} };
.&const ast::ClassDecl& C = *meta::getClassDecl("Window");
.&Adapter* adapter = new AdapterByDelegation(C);//create a generator
.!(adapter->adapt(.<typename WindowAdapter>., renames));//generates:
// class WindowAdapter {
//          Window* instance;
// public:  int  Draw (DC& dc)   { return this->instance->Draw(dc);  }
//          void Maximize (void) { this->instance->SetWholeScreen(); }
//          WindowAdapter(Window* o) : instance(o) {}
// };
.&delete adapter;//memory management in stages as with normal programs
```

6 Implementation Overview

MetaC++ is implemented as an extension layer on top of the Clang [cla] compiler, (Figure 5). Extensions include the staging-aware preprocessing, the added staging annotation tokens in the lexical analyzer, the handling of staging constructs in the syntax analyzer, the extensions in the semantic analyzer and AST library to take into account the staging infrastructure as well as the introduction of the staging loop and the staging runtime library (i.e. meta-compiler library functions). We continue with a brief, high-level overview of these extensions. A detailed discussion covering implementation aspects and relevant source code excerpts is available on Appendix A.

As discussed, code within quasi-quotes as well as code following a generate tag may contain unknown identifiers, so it is parsed as dependent code, i.e. as though it appeared within the definition of some template. As such, it is expected that quasi-quotes involving unbound identifiers will include AST nodes typically encountered only within template definitions, such as `UnresolvedLookupExpr`, `DependentScopeDeclRefExpr`, `CXXUnresolvedConstructExpr`, `CXXDependentScopeMemberExpr`, etc.

Changes in the semantic analyzer involved introducing new AST nodes for the staging elements and extending all AST visitors to handle them. It also required extending the declaration context, scoping and lookup infrastructure to become staging-aware, so that any symbol would only be visible in the stage nesting it was declared in. Supporting this required having multiple declaration contexts hierarchies (one per stage) instead of a single one, and shifting among them to match the stage nesting.

The staging loop is implemented as previously described, with the assembled stage program code being compiled by a separate Clang compiler instance and executed using the LLVM MCJIT execution engine [Docb]. Using this execution engine has the added benefit of supporting debugging of the dynamically generated code using GDB [Doca], effectively supporting stage program debugging.

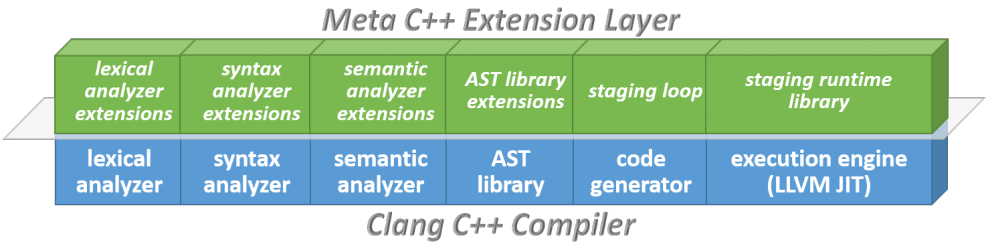

Figure 5 – MetaC++ extension layer on top of the Clang C++ compiler

Regarding compiler performance, the staging process of MetaC++ introduces minimal overhead by itself. Assembling a stage program involves just a couple of straightforward traversals over the program AST, while translating it is proportional to the size of the metaprogram, which typically would be orders of magnitude smaller than the normal program. The time spent on this is not significant compared to the time required to parse, analyze and translate any included standard library header. Any introduced overhead comes directly from the metaprogramming logic that dictates the number of stages and the time required for their execution. Thus, a trivial metaprogram may execute instantly involving no overhead, but an elaborate metaprogram may take a lot of time to execute, affecting compilation performance. Nevertheless, the same situation applies for template or constexpr-based metaprogramming in standard C++. Actually, metaprogram execution in MetaCPP can be significantly faster than executing equivalent `constexpr` functions at compile-time, as in our case the code is compiled through JIT, while `constexpr` functions are interpreted.

There is one implementation-related aspect that can potentially impact stage execution performance, particularly for metaprograms involving intensive AST composition. In Clang, ASTs are designed to be immutable, so the proposed method of transformation is source code rewriting through the Rewriter API that allows inserting and removing source text based on AST node locations. Thus, a generate tag does not replace itself with the evaluated AST value in the main program AST, but instead the evaluated AST value is converted to source text that replaces the generate tag code in the original source text. Likewise, escapes used to combine existing ASTs in new ASTs being created by the enclosing quasi-quotes, require converting the existing ASTs to source text, combining it with the source text of the quasi-quotes and finally parsing the resulting source text to AST. To overcome this, we are investigating alternatives for source code transformation at the AST level, including the TreeTransform functionality, used for template instantiations, or some custom AST processing library (e.g. [Krz]).

To allow our meta-language to be used on other platforms and compilers, we also offer the option to run only the staging loop (similar to the preprocess-only option), effectively operating as a standalone source-to-source transformation tool. The resulting source file is the final outcome of the staging loop that consists of pure C++ code and can then be compiled with any typical C++ compiler.

7 Related Work

We focus on supporting multi-stage generative metaprogramming for C++. As such, we consider work on multi-stage languages, generative programming and metaprogramming systems targeted for C++ to be related to ours. We already compared C++ template metaprograms to our language, so we don't repeat the discussion here.

7.1 Multi-Stage Languages

Early multi-stage languages, like *MetaML* [She99] and *MetaOCaml* [CLT+01] were statically typed functional languages that generate code at runtime and offer strong type-safety guaranties. Later research also covered staging during compilation, e.g. *Template Haskell* [SJ02], and its adoption in the context of imperative languages, including *Converge* [Tra08], *Metalua* [Fle07] and *Delta* [LS15]. Languages with compile-time staging typically offer less type-safety guaranties, as they are allowed to report type errors during compilation, and focus on expressiveness. MetaC++ shares these

elements and as it focuses on generative metaprogramming, it drops type-safety guaranties in favor of expressiveness. In particular, it allows expressing any AST code fragment and supports generating all language elements. It also distinguishes itself from most multi-stage languages by treating code with the same stage nesting as a unified coherent program, instead of isolated stage expressions. In this direction, it is closely related to Delta from which it adopts the integrated metaprogramming model. However, Delta is dynamically typed and has simpler grammar, AST representation, lookup and scoping rules, so offering similar staging features in C++ is far more challenging. For instance, Delta variables are declared by use, so an unknown identifier simply becomes a variable AST node. In MetaC++, parsing the unknown identifier may require disambiguation, while the created AST node will refer to a dependent variable and require extra handling when the declaration it refer to becomes available.

Relevant to our work are also multi-stage extensions of other mainstream languages. *Metaphor* [NR04] is a C# extension that supports runtime staging and features a reflection system that can interact with the staging constructs, thus allowing the generation of types. We too offer support for reflection and allow generating any code fragment, including types. *Mint* [WRI+10], a Java extension that supports runtime staging, tries to overcome the issue of scope extrusion in order to maintain type-safety. In our language, unbound variable within quasi-quotes are resolved in the context where the quoted code will actually be inserted, offering no name binding guaranties and involving no scope extrusion issue. *Backstage Java* [PS11] is a Java extension for compile-time staging that supports non-local changes and ensures that changes introduced by individual stage fragments are consistent. Our model treats such stage fragments as a unified program with a well-defined control flow, so there are no generation inconsistencies. Also, we support non-local changes as well by retrieving the internal compiler data structures and modifying them directly.

7.2 Generative Programming

There are various languages and systems supporting some form of generative programming. We present those we consider to be relevant to ours.

The *Jakarta Tool Set* [BLS98] supports creating domain specific languages using *Jak*, a Java extension with AST construction and manipulation features, and *Bali*, a parser generator for creating syntactic extensions. A domain specific program is parsed into an AST using the parser generated created by Bali, the AST is then modified through a Jak transformation program and the result is unparsed into a pure host-language program. Our language can also express algorithmic AST transformations, but it doesn't support syntactic extensions based on grammar specification.

SafeGen [HZS05] supports writing generators for Java programs. It features *cursors* that are variables matching program elements against first-order logic predicates, and *generators*, written as quasi-quotes, that use cursors to output code fragments. It can generate any legal Java code and ensures type-safety of generated code, but it is not as expressive as our language since it does not support algorithmic code generation logic.

Genoupe [DLW05], *CTR* [FCL06], *Meta-trait Java* [RT07] and *MorphJ* [HS11] are all C# and Java extensions that provide compile-time reflective facilities allowing to statically iterate over fields or members derived from some pattern and generate code for each match in a type-safe way. They offer limited or no support for code manipulation and they cannot generate arbitrary code. *MetaFJig* [SZ10] is a Java-like language that treats class definitions as first class values allowing them to be composed with some operators. Our language can support similar functionality with these

languages using its reflective features and expressing the pattern matching logic in an algorithmic fashion. In fact, concepts such as class composition or class morphing can be directly adopted and offered as metaprogramming libraries.

7.3 Metaprogramming Systems for C/C++

MS^2 [WC93] was the first language to offer Lisp-like macros for syntactically rich languages, like C. MS^2 has similarities with our language as its macros manipulate ASTs in an algorithmic way and generate code, operating similarly to our multi-stage computations. However, it is less expressive than our language, as it cannot generate types, and it also targets a syntactically and semantically simpler language than C++.

'C [PHEK99], is a two-level language that extends C with quasi-quote operators and supports metaprogramming through dynamic code generation. It introduces type constructors that allow dynamic code to be statically typed, however it does not offer strong type safety guarantees in the sense of MetaML. MetaC++ is more expressive as it can express any AST code fragment as opposed to just expressions and compound statements, and apart from functions can also generate types, structs and declarations.

OpenC++ [Chi95] is a C++ extension that offers a compile-time Meta-Object Protocol and focuses on enabling syntax extensions. Meta-objects are available during compilation and provide a compile-time reflection mechanism used to manipulate source code, eventually generating a pure C++ source file. However, meta-objects are restricted only for top-level class and member function definitions and the protocol adopted for their translation focuses only on objects, limiting the potential source locations for code generation to only class definitions, object declarations and instantiations, member read and write operations, and method invocations. Our language focuses on generative metaprogramming and naturally supports code generation in a far wider range of source locations, while it also offers compile-time reflection facilities.

C++ proposals for compile-time code generation and injection [VD17] and metaclasses [Sut18] are also closely related to our work, as we share the goal for supporting generative metaprogramming using normal C++ source code. Their approach is based on `constexpr` blocks, being closer to standard C++ `constexpr` functions but also inheriting their limitations. Additionally, our approach is more expressive as it supports multi-stage code generation and allows arbitrary code generation contexts.

8 Conclusions

We presented a generative multi-stage extension of C++ in which metaprograms share both common syntax and development practices with normal programs, fully reusing C++ as the language for implementing metaprograms. As shown with a case study for generative metaprogramming in standard C++, our approach helps overcoming the issues of template metaprograms that involve a different syntax and programming model compared to the normal language, thus disabling design or source code reuse.

We covered issues of both language design and implementation. We presented the adopted programming model and provided an overview of the staging annotations introduced for the meta-language as well as the extensions required for the normal language. We also detailed the staging assembly and evaluation process and illustrated the integration of our staging infrastructure with other compile-time evaluation features of the language. Then, we focused on practical application, presenting details for the extensions and modifications involved in our clang-based implementation.

Finally, we presented a series of application scenarios, focusing on compile-time reflection and the use of staging to generate exception handling patterns and design patterns, illustrating and validating the software engineering value of our approach.

Our work follows a different path compared to the currently prevalent approach of template metaprogramming. Nevertheless, it promotes a model in which metaprograms are no different than normal programs and can thus be developed and deployed in a coherent and uniform manner, without requiring elaborate template tricks. Essentially, with templates, metaprogramming is targeted mostly for experts and library authors, while with our proposition it becomes accessible to every C++ programmer.

Overall, we consider our work to be a significant step for metaprogramming in C++ and believe that integrating the two worlds will push forward current practices, allowing more advanced and comprehensive metaprograms to appear in the future.

References

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Baw99] Alan Bawden. Quasiquotation in Lisp. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, 1999. Available from: `http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf`.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 143–, Washington, DC, USA, 1998. IEEE Computer Society.

[Chi95] Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 285–299, New York, NY, USA, 1995. ACM. `doi:10.1145/217838.217868`.

[cla] clang: a c language family frontend for llvm. Available from: `http://clang.llvm.org/`.

[CLT+01] Cristiano Calcagno, Queen Mary London, Walid Taha, Liwen Huang, and Xavier Leroy. A bytecode-compiled, type-safe, multi-stage language. Technical report, 2001. Available from: `http://www.cs.rice.edu/~taha/publications/preprints/pldi02-pre.pdf`.

[Cpp] Cppreference.com. Dependent names. Available from: `http://en.cppreference.com/w/cpp/language/dependent_name`.

[Dio] Louis Dionne. Boost.Hana. Available from: `http://boostorg.github.io/hana/`.

[DLW05] Dirk Draheim, Christof Lutteroth, and Gerald Weber. A type system for reflective program generators. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*,

GPCE'05, pages 327–341, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11561347_22.

[Doca] LLVM Documentation. Debugging jit-ed code with gdb. Available from: http://llvm.org/docs/DebuggingJITedCode.html.

[Docb] LLVM Documentation. Mcjit design and implementation. Available from: http://llvm.org/docs/MCJITDesignAndImplementation.html.

[FCL06] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 275–284, New York, NY, USA, 2006. ACM. doi:10.1145/1173706.1173748.

[Fle07] Fabien Fleutot. Metalua manual, 2007. Available from: http://metalua.luaforge.net/metalua-manual.html.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In *Proceedings of the 7th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILPS '95)*, volume 982 of *LNCS*, pages 259–278. Springer, 1995. doi:10.1007/BFb0026825.

[HS11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–44, February 2011. doi:10.1145/1890028.1890029.

[HZS05] Shan Shan Huang, David Zook, and Yannis Smaragdakis. *Statically Safe Program Generation with SafeGen*, pages 309–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/11561347_21.

[KFFD86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM. doi:10.1145/319838.319859.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[Krz] Olaf Krzikalla. Performing source-to-source transformations with clang. Available from: http://llvm.org/devmtg/2013-04/krzikalla-slides.pdf.

[LS12] Yannis Lilis and Anthony Savidis. *Implementing Reusable Exception Handling Patterns with Compile-Time Metaprogramming*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-33176-3_1.

[LS13] Yannis Lilis and Anthony Savidis. An integrated approach to source level debugging and compile error reporting in metaprograms. *Journal of Object Technology*, 12(3):1:1–26, August 2013. URL: http://www.jot.fm/contents/issue_2013_08/article2.html, doi:10.5381/jot.2013.12.3.a2.

[LS15] Yannis Lilis and Anthony Savidis. An integrated implementation framework for compile-time metaprogramming. *Software: Practice and Experience*, 45(6):727–763, 2015.

[NR04] Gregory Neverov and Paul Roe. *Metaphor: A Multi-stage, Object-Oriented Programming Language*, pages 168–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-30175-2_9.

[PHEK99] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. C and tcc: A language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, March 1999. doi:10.1145/316686.316697.

[PS11] Zachary Palmer and Scott F. Smith. 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, pages 939–958, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048137.

[RT07] John Reppy and Aaron Turon. *Metaprogramming with Traits*, pages 373–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-73589-2_18.

[SC13] Jeff Snyder and Chandler Carruth. Call for compile-time reflection proposals (n3814), 2013. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3814.html.

[She99] Tim Sheard. *Using MetaML: A Staged Programming Language*, pages 207–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi:10.1007/10704973_5.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the 2Nd International Conference on Semantics, Applications, and Implementation of Program Generation*, SAIG'01, pages 2–44, Berlin, Heidelberg, 2001. Springer-Verlag.

[SJ02] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002. doi:10.1145/636517.636528.

[SP12] Ábel Sinkovics and Zoltán Porkoláb. Metaparse - compile-time parsing with template metaprogramming. C++Now, Aspen, USA, 2012. Available from: https://pdfs.semanticscholar.org/1997/ae1852b0ff66299323dfb6e5f045a27db041.pdf.

[Ste90] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[Sut18] Herb Sutter. Metaclasses: Generative c++, 2018. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf.

[SZ10] Marco Servetto and Elena Zucca. 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, pages 464–483, New York, NY, USA, 2010. ACM. doi:10.1145/1869459.1869498.

[Tah04] Walid Taha. *A Gentle Introduction to Multi-stage Programming*, pages 30–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-25935-0_3.

[Tra08] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6):1–40, 2008. doi:10.1145/1391956.1391958.

[TS00] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211 – 242, 2000. PEPM'97. doi:https://doi.org/10.1016/S0304-3975(00)00053-0.

[VD17] Daveed Vandevoorde and Louis Dionne. Exploring the design space of metaprogramming and reflection, 2017. Available from: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0707r3.pdf.

[Vel96] Todd Veldhuizen. Using c++ template metaprograms. In *C++ gems*, pages 459–473. SIGS Publications, Inc., 1996.

[Vel03] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University Computer Science, 2003. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.3670&rep=rep1&type=pdf.

[WC93] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 156–165, New York, NY, USA, 1993. ACM. doi:10.1145/155090.155105.

[WRI+10] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 400–411, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1806596.1806642, doi:10.1145/1806596.1806642.

About the authors

Yannis Lilis owns an PhD from the Department of Computer Science, University of Crete, and is a Research Associate at the Institute of Computer Science - FORTH. His e-mail address is lilis@ics.forth.gr.

Anthony Savidis is a Professor of 'Programming Languages and Software Engineering' at the Department of Computer Science, University of Crete, and an Affiliated Researcher at the Institute of Computer Science, FORTH. His e-mail address is as@ics.forth.gr.

A Implementation Details

A.1 Preprocessing Extensions

The staging-aware preprocessing is currently partially implemented with normal preprocessor directives expanded directly for normal program code and stage preprocessor directives being treated as special declarations that do not directly affect parsing, but instead become part of the main program AST and follow the standard rules for stage assembly, eventually taking effect later when the staging loop reaches the matching nesting. For example, a stage include directive is not actually directly preprocessed in the context of the original source file, but instead during the assembly of the corresponding stage and within its source code context. The target was to support stage preprocessor directives with minimal changes by adopting a delayed preprocessing scheme that utilizes the original preprocessor. An excerpt of the code extensions for the described functionality is provided below.

```
//AST class for a stage preprocessor directive, e.g. .@#directive
class PreprocessorDirectiveDecl : public Decl {
  StringRef Directive; //directive is verbatimely copied in stage code
  //additional code here...
};
PreprocessorDirectiveDecl* Sema::ActOnPreprocessorDirectiveDecl(
       StringRef D, SourceLocation Start, SourceLocation End){
 PreprocessorDirectiveDecl* PD =//AST for the parsed stage directive
   PreprocessorDirectiveDecl::Create(Context,CurContext,D,Start,End);
 CurContext->addHiddenDecl(PD); //add directive in current context
 return PD;
}
Decl* Parser::ParsePreprocessorDirectiveDecl() {
  SourceLocation Start = Tok.getLocation(), End;
  getPreprocessor().DiscardUntilEndOfDirective(&End);
  const char* begin = PP.getSourceManager().getCharacterData(Start);
  const char* end = PP.getSourceManager().getCharacterData(End);
  StringRef Directive(begin, end - begin);
  return Actions.ActOnPreprocessorDirectiveDecl(Directive,Start,End);
}
Parser::DeclGroupPtrTy Parser::ParseExternalDeclaration(/*...*/) {
   //...original code here...
 switch (Tok.getKind()) {   //...original code here...
   case tok::hash: SingleDecl=ParsePreprocessorDirectiveDecl();break;
 }
}
```

A potential problem of this approach is that stages require further preprocessing and if they include files with meta-code (e.g. as in Figure 4) the preprocessed stage will end-up containing meta-code, thus breaking the working model of stage code being normal program code that can be compiled with the original language compiler. Nevertheless, such code can be compiled recursively through the meta-compiler, at the cost of slightly increasing the implementation complexity of the staging loop.

A real issue relates to stage macro definitions and the fact that delayed expansion of their invocations may cause parsing discrepancies. For example, consider the following

code that attempts to declare a stage function with an empty body.

```
.@#define EMPTY_BLOCK {}
.@void f() EMPTY_BLOCK
```

Since the macro definition does not affect the original parsing context, the function definition sees an identifier instead of an opening and closing brace, thus failing to parse correctly. Considering that the problem arises only in macros, and that our language offers far greater metaprogramming support compared to preprocessor macros, we could argue for dropping the problematic *define* directives and replacing them with stage programs. However, such an approach is not pragmatic as C++ is a mainstream programming language with huge code bases potentially utilizing such macros, that we simply cannot reject.

Overall, we plan to revisit our implementation to deliver a fully working solution for the staging-aware preprocessing step that we described earlier.

A.2 Syntactic Extensions

The lexical analyzer extensions are minimal and straightforward, requiring only the introduction and recognition of staging annotations, as shown with the code below.

```
PUNCTUATOR(periodless,   ".<") //staging annotation token definitions
PUNCTUATOR(greaterperiod,">.")
PUNCTUATOR(periodtilde,  ".~")
PUNCTUATOR(periodexclaim,".!")
PUNCTUATOR(periodamp,    ".&")
PUNCTUATOR(periodat,     ".@")
//lexer extensions to recognize staging annotations
bool Lexer::LexTokenInternal(/*...*/){
  //...original code here...
  switch(CurChar) {  //...original code here...
    case '.': //...original code for other cases that begin with '.'...
      std::map<char, tok::TokenKind> mappings{ { '@',tok::periodat },
        { '<', tok::periodless    }, { '~', tok::periodtilde },
        { '!', tok::periodexclaim }, { '&', tok::periodamp } };
      auto iter = mappings.find(NextChar);
      Kind = iter == mappings.end() ? tok::period : iter->second;
      break;
    case '>': //...original code for other cases that begin with '>'...
      Kind = NextChar == '.' ? tok::greaterperiod : tok::greater;
      break;
  }
}
```

The syntax analyzer extensions require more effort in order to match each staging annotation with the desired parse form and the contexts it may appear in. For instance, define and execute tags typically appear at global scope, while generate tags, escape tags and quasi-quotes typically appear in an expression context. As such, the code for parsing external declarations and expressions is extended as shown below.

```
Parser::DeclGroupPtrTy Parser::ParseExternalDeclaration(/*...*/) {
 //...original code here...
```

```
 switch (Tok.getKind()) {   //...original code here...
   case tok::periodat: {
    ParseScope Scope(this, Scope::DeclScope); //new declaration scope
    Define* D=Actions.ActOnStartDefine(ConsumeToken());//Sema actions
    ParseExternalDeclaration(/*...*/);//parse decl after the .@ token
    SingleDecl = Actions.ActOnEndDefine(D, PrevTokLocation);
    break;
   }
   case tok::periodamp: {
    ParseScope Scope(this, Scope::DeclScope); //new declaration scope
    Execute* E=Actions.ActOnStartExecute(ConsumeToken())//Sema actions
    StmtResult Res = ParseStatement(); //parse stmt after the .& token
    SingleDecl=Actions.ActOnEndExecute(E,Res.get(),PrevTokLocation);
    break;
   }
 }
 //...original code here...
}
ExprResult Parser::ParseCastExpression(/*...*/) {
  //...original code here...
 switch (Tok.getKind()) {   //...original code here...
  case tok::periodless: {
   unsigned ScopeFlags = Scope::BreakScope | Scope::ContinueScope   |
    Scope::ControlScope| Scope::DeclScope  | Scope::FnScope         |
    Scope::BlockScope  | Scope::ClassScope | Scope::QuasiQuotesScope;
   ParseScope QuasiQuotesScope(this, ScopeFlags);//special quote scope
   SourceLocation StartLoc = ConsumeToken(), EndLoc;
   Actions.ActOnStartQuasiQuoteExpr(StartLoc);//make quote DeclContext
   AST* ast = ParseQuasiQuotes(); //parse the code inside quasi-quotes
   if (ast && TryConsumeToken(tok::greaterperiod, EndLoc))
     return Actions.ActOnEndQuasiQuoteExpr(StartLoc, EndLoc, ast);
   else
     return Actions.ActOnQuasiQuoteError();
  }
  case tok::periodexclaim: //fallback      //parse .!() or .~() exprs
  case tok::periodtilde:   return ParseMetaGeneratedExpr();
 }
}
```

Beyond the typical expression context, we further support code generation to occur in a variety of source locations including statements, top level declarations, declaration contexts (e.g. body of a class, struct or union), parameter declarations, types and names. The statement context is supported directly through the expression context as `expr;` is a valid C++ statement. To support top level declarations and declaration contexts, we extend the parser to accept the forms `.!(...);` and `.~(...);` as declarations that are subject to code generation, as illustrated below.

```
Parser::DeclGroupPtrTy Parser::ParseExternalDeclaration(/*...*/) {
  if (Decl* D = TryParseMetaGeneratedDecl())
    return Actions.ConvertDeclToDeclGroup(D);
  //...original code here...
```

```
}
Parser::DeclGroupPtrTy Parser::ParseCXXClassMemberDeclaration(/*..*/){
  if (Decl* D = TryParseMetaGeneratedDecl())
    return Actions.ConvertDeclToDeclGroup(D);
  //...original code here...
}
Decl* Parser::TryParseMetaGeneratedDecl() {
 if (Tok.isOneOf(tok::periodexclaim, tok::periodtilde)) {
 TentativeParsingAction TPA(*this);//tentative parse-it may be an expr
 SourceLocation StartLoc = Tok.getLocation(), EndLod;
 ExprResult Res=ParseMetaGeneratedExpr();//parse the .!() or .~() expr
 if (!Res.isInvalid() && TryConsumeToken(tok::semi, EndLoc)) {
   TPA.Commit();//if generated expr parsed successfully commit actions
   return Actions.ActOnMetaGeneratedDecl(StartLoc,EndLoc,Res.get()));
 }
 TPA.Revert();//if parsing failed revert actions and continue normally
 }
 return nullptr;
}
```

The remaining cases are supported by further accepting the forms `.!(...)` and `.~(...)` as valid identifiers that are again subject to code generation. In this sense, the source code to get the name of an identifier token (present in various parsing functions), is replaced with calls to the following introduced `ParseIdentifier` function.

```
IdentifierInfo *Parser::ParseIdentifier() {
 IdentifierInfo *II = nullptr;
 if (Tok.is(tok::identifier))//original code was just the if statement
  II=Tok.getIdentifierInfo(); //extension to checks for generated code
 else if (Tok.isOneOf(tok::periodexclaim, tok::periodtilde)) {
  ExprResult Res=ParseMetaGeneratedExpr(); //parse a .!() or .~() expr
  if (!Res.isInvalid()) {
   II=PP.newIdentifierInfo(makeUniqueName()); //make unique identifier
   II->setFETokenInfo(Res.get());   //set AST as extra identifier data
  }
 }
 return II;
}
```

In all cases, the parsed entity (identifier or declaration) contains the AST of the expression enclosed within the *generate* or *escape* tag. The code generated during stage execution may consist of a single identifier or declaration, substituting itself in the originally parsed form, or it may involve multiple or more complex language elements, potentially extending the originally parsed form upon code generation. For example, consider the following code that generates a function definition.

```
.@AST* retType = .<int>.;
.@AST* name = .<add>.;
.@AST* formals = .<int x, int y>.;
.@AST* body = .<return x + y;>.;
typename .!(retType) .!(name) (typename .!(formals)) { .!(body); }
//code generation result: int add (int x, int y) { return x + y; }
```

The function to be generated is parsed as *typename* ID_1 ID_2 *(typename* ID_3*) {...}*, where typename ID_1 and typename ID_3 are dependent types and ID_2 is a name. Thus, the parsed entity denotes a function named ID_2 that takes a single unnamed argument of dependent type ID_3 and returns a value of dependent type ID_1. The types ID_1 and ID_3 as well as the name ID_2 are associated with the corresponding expression ASTs of `.!(retType)`, `.!(formals)` and `.!(name)`. Then, upon stage execution and based on the actual values of these expressions the return type ID_1 is substituted by `int`, the name ID_2 is substituted by `add` and the formal argument list is transformed by substituting the unnamed argument of type ID_3 with two `int` arguments named `x` and `y`. Further examples with the supported source locations for code generation and their corresponding parse forms are presented in Table 1.

Source Location	*Parse form*	*Sample code segment involving generate tags*	*Code generation examples*
Expression	`expr`	`int x = .!(ast);`	`int x = 1 + 2;` `int x = f();`
Actual argument	`expr`	`f(!(ast));`	`f(x);` `f(1, 2, 3);`

Statement or block declaration	`expr;`	`void f(){ .!(ast); }`	`void f(){ int x; }`
			`void f(){ printf("ok"); }`
			`void f(){ return; }`

Top-level declaration	`decl;`	`.!(ast);`	`int x;`
			`void f(){}`
			`class C {…};`

Class member declaration	`decl;`	`class X { .!(ast); };`	`class X { int x; };`
			`class X { void f(){} };`
			`class X{virtual int f();};`

Type	`id`	`typename .!(ast) f();`	`void f();` `char* f();`
Name	`id`	`void .!(ast)();`	`void generated_name();`
Namespace specifier	`id`	`.!(ast)::var;`	`std::cout;` `X::static_var;`
Parameter declaration	`id`	`void f(typename.!(ast));`	`void f();` `void f(int x, double y);`
Base class specifier	`id`	`class X :` `typename .!(ast) {};`	`class X : Y {};` `class X : Y, Z {};`
Catch clause variable declaration	`id`	`try {} catch(.!(ast)){}`	`try {} catch(X& ex)) {}` `try {} catch(...)) {}`

Enum entry	`id`	`enum Options {.!(ast)};`	`enum Options {SLOW, FAST};`
			`enum Options`
			`{ EASY=0, DIFFICULT=1 };`

Lambda function capture list	`id`	`[.!(ast)](){};`	`[a](){};` `[a, b, c](){};`

Table 1 – Adopting specific parse forms to support code generation for various source locations

Code within quasi-quotes may consist of various language elements such as expres-

sions, statement or declaration lists, top level declarations, parameter declarations, declaration contexts (e.g. body of a class, struct or union), and types. Thus, parsing such code requires extending the tentative parsing infrastructure to support matching any of the alternative code forms, taking into account a significant portion or even the entire quasi-quote contents instead of few tokens past the opening quote. For example, the fact that the quasi-quote `.<int x; void f(); friend void g();>.` refers to declarations within a class body instead of top level or statement level declarations only becomes clear mid-parse after encountering the friend declaration, meaning multiple different parses may be required to match the given form. The code bellow offers a high level overview of the quasi-quote parsing implementation.

```
AST* Parser::ParseQuasiQuotes() {
  using ParseFunc = std::function<AST* (Parser&)>;
  std::vector<ParseFunc> parseFuncs { //parsing function wrappers that
    ParseType, ParseExpression,       //simulate each parsing context
    ParseParameterDeclaration, ParseTopLevelDeclarations,
    ParseStatementOrDeclarationList, ParseClassBody
  };
  DiagnosticErrorTrap ErrorTrap(Diags);//trap errors in parse attempts
  Diags.setSuppressAllDiagnostics(true);//but do not issue diagnostics
  AST* ast = nullptr;
  for (const ParseFunc& f : parseFuncs) {
    TentativeParsingAction TPA(*this); //mark any parsing as tentative
    if ((ast = f(*this)) && !ErrorTrap.hasErrorOccurred())
     { TPA.Commit();break; }//if parsing succeeded commit actions&exit
    else {     //otherwise revert parsing, clear any errors and remove
      TPA.Revert(); ErrorTrap.reset();    //any tentative declarations
      RemoveTentativeDeclarations(Actions.CurContext, getCurScope());
    }
  }
  Diags.setSuppressAllDiagnostics(false);        //restore diagnostics
  return ast;
}
```

Multiple alternative parses may match but only the first match is kept at this point. Any alternatives are explored in the context where the quasi-quoted code will be inserted, effectively adapting the original parse form, as shown below.

```
.@AST* ast = .<int x;>.; //parsed as a top level declaration
.!(ast);                 //ok, used as top level declaration
void f(){ .!(ast); }  //ok, valid use as a statement-level declaration
class X { .!(ast); };  //ok, valid use as a class member declaration
int y=.!(ast); //stage execution error:use as an expression is invalid
```

Another viable approach would be to keep all potential parse forms and then disambiguate based on the insertion context, but this would involve a significant overhead in both parsing time (always parse all forms) and memory consumption.

A.3 Semantic Extensions

Various extensions are required for the semantic analysis (i.e. the `Sema` class) to handle the staging infrastructure, with the most important being the treatment of

unknown identifiers. Code involving unknown identifiers (within quasi-quotes or after a *generate* tag) is essentially treated as dependent code, i.e. as though it appeared within the definition of a template. This means that the AST representation of such code fragments may include nodes that are typically encountered only within template definitions, such as `DependentScopeDeclRefExpr`, `CXXDependentScopeMemberExpr`, `UnresolvedLookupExpr`, `CXXUnresolvedConstructExpr`, etc., and requires extensions to generate and accept dependent types and variables outside of template definitions. In particular, the semantic name classification method (`Sema::Classify`) consulted by the parser to resolve identifiers and direct parsing is extended to treat unknown names as dependent identifiers. Moreover, unknown identifiers specifying types (based on the context or through the extended `typename` keyword) are transformed into artificial `DependentNameType` objects by adopting the same technique as in `Sema::ActOnDelayedDefaultTemplateArg` but with an empty `NestedNameSpecifier` instead of a synthesized one, as shown below.

```
ParsedType Sema::ActOnUnknownType(const IdentifierInfo &II,
                                  SourceLocation NameLoc) {
 NestedNameSpecifier *NNS = nullptr; //build a fake DependentNameType
 QualType T = Context.getDependentNameType(ETK_None, NNS, &II);
 return CreateParsedType(T, BuildTypeSourceInfo(Context, NameLoc, T));
}
```

Similarly, we handle the usage of the scope resolution operator involving unresolved identifiers through artificially dependent nested name specifiers. The same applies for the `::template` form in which the global scope is treated as an artificially dependent context. The following code sketches the implementation for these extensions.

```
//handle the a::b form
bool Sema::BuildCXXNestedNameSpecifier(IdentifierInfo &Id,
  SourceLocation IdLoc,SourceLocation CCLoc,CXXScopeSpec &SS,/*...*/){
 LookupResult Found(/*...*/); //...original code here...
 //if identifier not resolved, build a dependent nested-name-specifier
 if (Found.empty() && CurContext->allowUnresolvedIds())
  { SS.Extend(Context, &Id, IdLoc, CCLoc); return false; }
 //...original code here...
}
//handle the ::template a form
TemplateNameKind Sema::isTemplateName(CXXScopeSpec &SS, bool
  hasTemplateKeyword, UnqualifiedId &Name,TemplateTy &Result,/*...*/){
 LookupResult R(/*...*/); //...original code here...
 //if id is not resolved return a dependent template name
 if(R.empty()&&hasTemplateKeyword&&CurContext->allowUnresolvedIds()){
   Result = TemplateTy::make(Context.getDependentTemplateName(
     SS.getScopeRep(), Name.Identifier));
   return TNK_Dependent_template_name;
 }
 //...original code here...
}
//handle the ::template a::b form
bool Sema::ActOnCXXNestedNameSpecifier(CXXScopeSpec &SS,
     TemplateTy Template, SourceLocation CCLoc, /*...*/) {
```

```
  //...original code here...
  DependentTemplateName *DTN =
    Template.get().getAsDependentTemplateName();
  if (DTN && DTN->isIdentifier() && DTN->getQualifier() &&
      DTN->getQualifier()->getKind() == NestedNameSpecifier::Global){
    QualType T = Context.getDependentTemplateSpecializationType(
      ETK_None, nullptr, DTN->getIdentifier(), TemplateArgs);
    NestedNameSpecifier *NNS = NestedNameSpecifier::Create(Context,
      DTN->getQualifier(), true, T.getTypePtr());
    SS.MakeTrivial(Context, NNS, SourceRange(SS.getBeginLoc(),CCLoc));
    return false;
  }
  //...original code here...
}
```

All above cases also require extending the use of `NestedNameSpecifier` objects as well as loosening some assertions regarding their usage.

The presence of unknown identifiers involves further extensions related to lambda functions. Unknown identifiers found within a capture list (presumably capturing a generated variable in an outer scope) require introducing artificial dependent-typed variables so as to keep the capture list valid. Similarly, unknown identifiers within the body of a lambda function, referring either to a variable generated within its body or a generated identifier within its capture list, require introducing an artificial capture list entry to keep the lambda body valid.

Various small scale extensions are also needed to suppress certain typing checks that would otherwise fail for generated code or code in quasi-quotes. For example, we cannot statically determine the result of a code generation expression, so we treat it as a dependent type to skip any further type-checking. The same applies for the `this` keyword appearing in quasi-quotes without an enclosing class definition. For instance, in the quasi-quoted code `.<this->f()>.` there is no information about the type of this so we should skip any type-checking regarding the presence of a member function f. Similarly, a quasi-quoted `return` statement without an enclosing function, e.g. `.<return 0;>.`, should not type check its return value against an inexistent return type. Examples of such extensions are presented in the code excerpt below.

```
QualType Sema::getCurrentThisType() {
  //...original code here...
  if (ThisTy.isNull() && getCurQuasiQuotesDecl())   //treat unresolved
   ThisTy=Context.DependentTy; //quasi-quoted 'this' as type dependent
  //...original code here...
}
ExprResult Sema::CreateBuiltinUnaryOp(UnaryOperatorKind Opc, /*...*/){
 //...original code here...
 switch (Opc) {    //...original code here...
  case UO_Generate://treat unary operators .! and .~ as type dependent
  case UO_Escape:  resultType = Context.DependentTy; break;
 }
 //...original code here...
}
StmtResult Sema::BuildReturnStmt(/*...*/) {
 //...original code here...
```

```
 //treat quoted returns outside functions as type dependent
 if(!getCurFunctionOrMethodDecl() && getCurQuasiQuotesDecl())
  FnRetType = Context.DependentTy;
 //...original code here...
}
```

Additional extensions are required to allow specific code forms to appear in source locations that would typically be considered erroneous. For example, quasi-quotes denoting a function or template definition may appear within the body of a function, even though C++ does not allow function or template definitions within the scope of a function. Similarly, quasi-quotes may include `break`, `continue` and `return` statements without necessarily appearing within a switch, loop or function body. Also, quasi-quotes may include virtual or friend function declarations without necessarily appearing in a class scope. To properly handle such cases, quasi-quotes introduce a scope capable of hosting various items by combining multiple bit-fields of the `ScopeFlags` enum (shown earlier as part of the `Parser::ParseCastExpression` extensions). They also introduce a new declaration context in order to collect any declarations appearing inside them and disallow them to interfere with enclosing declaration contexts of normal program code. For quasi-quotes denoting class members in particular, a fake enclosing class is artificially introduced supporting entities that appear only inside classes and keeping any class-related semantic checks valid. In this context, constructor definitions require additional handling as the name of the class is unknown and the member initializer list may include unknown member fields. In particular, within the artificially introduced class, any declaration beginning with the form *typename ID (parameter declarations)* is considered to be a constructor, presumably for a class named `ID`, while any missing member fields within the initializer list are handled by inserting fake dependent type member entries in the artificial class body. As such, the quoted code `.<typename X(): x(){}>.` is effectively parsed and semantically analyzed as though it occurred in the following context.

```
  template<typename T> class X {  //artificially generated class scope
  T x;                  //artificially inserted dependent class members
  X() : x() {}          //equivalent quoted code to be parsed
};
```

Apart from the `Sema` class extensions, semantic analysis also requires extending the declaration context and scoping infrastructure to become staging-aware, so that any symbol would only be visible in the stage nesting it was declared in. Supporting this requires having multiple hierarchies of declaration contexts and scopes (one for each stage) instead of a single one, and shifting among them to match the respective stage nesting. The following code illustrates this extended infrastructure and presents an example of its usage to implement the semantic actions for the *define* staging tag.

```
class Sema {
 //...original code here...
 std::map<unsigned, DeclContext*> Contexts; //decl context hierarchies
 std::map<unsigned, Scope*> Scopes;         //scope hierarchies
 unsigned CurStage;                         //stage nesting
 void ShiftStage(bool enter) {
  Contexs[CurStage]=CurContext;//update stored declaration context and
  Scopes[CurStage] = CurScope; //scope entries with latest information
  CurStage = enter ? CurStage + 1 : CurStage - 1; //shift stage
```

```
  if (enter && Contexts.find(CurStage) == Contexts.end()) {
   //on first stage nesting enter create a TranslationUnitDecl & Scope
   TranslationUnitDecl *TUDecl = TranslationUnitDecl::Create(Context);
   TUScope = new Scope(nullptr, Scope::DeclScope, Diags);
   TUScope->setEntity(TUDecl);
   Context.setTranslationUnitDecl(TUDecl); //update context and link
   CurContext->addHiddenDecl(TUDecl);   //with the stage DeclContext
   Contexts[CurStage] = TUDecl;
   Scopes[CurStage] = TUScope;         //store current stage info
  }
  else //otherwise restore the existing stage TranslationUnitDecl
    Context.setTranslationUnitDecl(
      Cast<Decl>(Contexts[CurStage])->getTranslationUnitDecl());
  CurContext = Contexts[CurStage];   //use the declaration context and
  CurScope = Scopes[CurStage];  //scope entries from the updated stage
 }
 Define* ActOnStartDefine(SourceLocation StartLoc) {
  ShiftStage(true); //enter a nested stage
  Define *D = Define::Create(Context, CurContext, StartLoc);
  CurContext->addHiddenDecl(D);//add new DeclContext for following def
  PushDeclContext(CurScope, D);
  return D;
 }
 Define* ActOnEndDefine(Define *D, SourceLocation EndLoc) {
  D->setEndLoc(EndLoc); PopDeclContext();
  ShiftStage(false); //leave from a nested stage
  return D;
 }
}
```

Finally, all aspects of the AST library need to be extended to support the staging infrastructure and other related extensions such as the extra syntax disambiguation. This involves introducing new AST nodes for the staging elements and extending the various AST visitors and serialization routines, as well as extending the AST-to-text transformations (AST printing and pretty printing) to generate appropriate source text for both new and extended AST nodes.

A.4 Compiler Staging Loop

The staging loop operates as described in section 3.4, with the computation of the maximum stage nesting and the assembly of stage code being implemented as custom AST visitors (`RecursiveASTVisitor` subclasses). The code assembled for each stage program is compiled by a separate Clang compiler instance, while any internal meta-compiler functions used in stage code (i.e. `meta::codegen`, `meta::quasiquotes`, `meta::getClassDecl`, etc.) are just part of the meta-compiler code base with appropriate linkage so as to be exported to the execution environment. An excerpt of the extended Clang driver, illustrating the implementation of the staging loop and the custom AST visitors it involves, is presented below.

```
template<typename Derived>   //AST visitor to count the stage nesting
class StageNestingVisitor : public RecursiveASTVisitor<Derived> {
```

```
public:
  StageNestingVisitor() : stageNesting(0) {}
#define DEF_TRAVERSE_STAGING_TAG(NAME, TYPE)              \
  bool NAME(TYPE *Val) {                                  \
    ++stageNesting;                                       \
    bool result = RecursiveASTVisitor<Derived>::NAME(Val);\
    --stageNesting;                                       \
    return result;                                        \
  } //macro to generate Traverse funcs for the 3 staging tag AST nodes
  DEF_TRAVERSE_STAGING_TAG(TraverseGenerate, Generate)
  DEF_TRAVERSE_STAGING_TAG(TraverseExecute,  Execute)
  DEF_TRAVERSE_STAGING_TAG(TraverseDefine,   Define)
protected:
  unsigned stageNesting;
}
//AST Visitor to assemble stage progam code
class StageAssembler : public ASTConsumer,
                       public StageNestingVisitor<StageAssembler> {
public:
 StageAssembler() : maxStageNesting(0) {}
 void HandleTranslationUnit(ASTContext&Context) override //ASTConsumer
  { TraverseDecl(Context.getTranslationUnitDecl()); } //API refinement
 bool VisitGenerate(Generate *G){//Visit func for 'generate' tag node
  if(CheckAndUpdateMaxStageNesting())//write max nesting code to stmts
   stmts << "meta::codegen(" << to_string(G) << ");\n";
  return true;
 }
 bool VisitExecute(Execute*E){//Visit func for execute writes to stmts
  if (CheckAndUpdateMaxStageNesting()) stmts << to_string(E) << "\n";
  return true;
 }
 bool VisitDefine(Define *D) { //Visit func for define writes to defs
  if (CheckAndUpdateMaxStageNesting()) defs << to_string(D) << "\n";
  return true;
 }
 std::string getStageCode() const {   //assemble collected stage code
  std::ostringstream ss; //add meta lib header & main stage function
  ss << "#include <meta>\n" << defs.str() << "\n"
     << "int main() {\n" << stmts.str() << "return 0;\n}\n";
  return ss.str();
 }
 unsigned getMaxStageNesting() const { return maxStageNesting; }
private:
 bool CheckAndUpdateMaxStageNesting() {
  if(stageNesting>maxStageNesting){ //if we encounter a higher nesting
   maxStageNesting = stageNesting;  //update the maximum stage nesting
   defs.str(std::string());        //clear the code streams to discard
   stmts.str(std::string());       //code from the previous nesting
  }
```

```
  return stageNesting == maxStageNesting; //return if at max nesting
 }
 unsigned maxStageNesting;
 std::ostringstream defs, stmts;
};
//AST visitor to rewrite the program code
class MainRewriter : public ASTConsumer,
                     public StageNestingVisitor<MainRewriter> {
public:
 MainRewriter(CompilerInstance *C, unsigned nesting) :
  maxStageNesting(nesting),
  Rewrite(C->getSourceManager(),C->getLangOpts()) {}
 void HandleTranslationUnit(ASTContext&Context) override {
  iter = generated.begin();//iterator for the generated code fragments
  TraverseDecl(Context.getTranslationUnitDecl());  //perform traversal
  generated.clear();  //clear the code generated for the current stage
 }
 bool VisitGenerate(Generate *G){ //Visit func for 'generate' tag node
  if (stageNesting == maxStageNesting)    //focus on max stage nesting
   Rewrite.ReplaceText(G->getSourceRange(),to_string(*iter++));
  return true; //replace tag code with the matching meta::codegen call
 }
 bool VisitExecute(Execute *E){ return Prune(E->getSourceRange()); }
 bool VisitDefine (Define *D) { return Prune(D->getSourceRange()); }
 bool Prune(SourceRange SR){     //prune .& and .@ tags at max nesting
  if (stageNesting == maxStageNesting) Rewrite.ReplaceText(SR, "");
  return true;
 }
 std::string getTransformedCode() const {      //get final source text
  FileID FID = Rewrite.getSourceMgr().getMainFileID();
  const RewriteBuffer *Buffer = Rewrite.getRewriteBufferFor(FID);
  return std::string(Buffer->begin(), Buffer->end());
 }
 //func used by meta::codegen to provide the generated code fragments
 static void addGenCode(AST *ast) { generated.push_back(ast); }
private:
 static std::vector<AST*> generated;
 std::vector<AST*>::const_iterator iter;
 unsigned maxStageNesting;
 Rewriter Rewrite;//Rewriter used for source-to-source transformations
};                //as clang ASTs are immutable by design
//exported meta-compiler lib function invoked during stage execution
META_LIBRARY void meta::codegen(AST*a){ MainRewriter::addGenCode(a); }
int cc1_main(ArrayRef<const char*>Argv){ //meta-compiler main function
 std::unique_ptr<CompilerInstance> Compiler(new CompilerInstance());
 InitializeCompilerFromCommandLineArgs(Compiler, Argv);//main Compiler
 while(true) {                           //staging loop implementation
  StageAssembler assembler; //visitor to track and assemble stage code
  Parse(Compiler, &assembler);  //parse input to AST and apply visitor
```

```
  if(unsigned nesting=assembler.getMaxStageNesting()){//if staged code
   std::string stageCode=assembler.getStageCode();//stage source text
   std::unique_ptr<CompilerInstance>//use a separate (normal) compiler
     StageCompiler(new CompilerInstance());  //for the assembled stage
   InitializeCompilerFromText(StageCompiler, stageCode);
   Parse(StageCompiler);                //parse the stage source text
   std::unique_ptr<CodeGenAction> Act(new EmitLLVMOnlyAction());
   if (!StageCompiler->ExecuteAction(*Act) ||  //generate stage binary
       !Execute(std::move(Act->takeModule()))) //execute stage binary
    break;     //on stage generation or execution errors exit the loop
   //AST visitor that will generate the transformed program code
   MainRewriter rewriter(Compiler.get(), nesting);
   std::string code = rewriter.getTransformedCode(); //reinit compiler
   InitializeCompilerFromText(Compiler,code);  //with the updated code
  }//if there is no more meta-code we have pure C++ code so we perform
   //a normal compiler invocation and exit the staging loop
  else { ExecuteCompilerInvocation(Compiler.get()); break; }
 }
}
```

B Full Code for Adapter Pattern Generator Case Study

```
.@class Adapter { //adapter pattern generator interface
protected: const ast::ClassDecl& Class;
public:
 using AdapterMap = std::map<std::string, std::string>;
 AST* adapt(AST* newId, const AdapterMap& renames) const {
  AST* methods=nullptr;//AST of adapted class methods, initially empty
  for (auto& method : Class.methods()) {  //iterate over class methods
   if (method.getAccess()!=AS_public) continue;//handle public methods
   const std::string name = method.getName();
   auto iter = renames.find(name); //if no renaming, use original name
   std::string newName = iter == renames.end() ? name:iter->second;
   AST* actuals=nullptr;//AST of adapted call actuals, starts empty
   const ast::Formals* formals = method.formals();
   for (auto& formal:formals) //iterate formals to create actuals list
    actuals = .<.~actuals, .~(meta::id(formal.getName()))>.
   AST* call = MakeCall(meta::id(name),actuals);//make proper call AST
   const ast::Type* retType = method.getReturnType();
   //return the adapted call result for non-void functions
   AST* body = retType->isVoidType() ? call : .<return .~call;>.;
   AST* newMethod = //full AST for the adapted method
    .<typename .~retType .~(meta::id(newName)) (.~formals){.~body;}>.;
   methods = .<.~methods, .~newMethod>.; //merge with previous methods
  }
  return MakeClass(newId, methods); //make the proper class AST
 }
 virtual AST* MakeCall(AST* name, AST* actuals) const = 0;
```

```
 virtual AST* MakeClass(AST* newId, AST* methods) const = 0;
 Adapter(const ast::ClassDecl& Class) : Class(Class) {}
};
.@class AdapterByDelegation : public Adapter {//pattern implementation
public:                                        //using delegation
 AST* MakeCall(AST* name, AST* actuals) const override
   { return .<this->instance..~name)(.~actuals)>.; }
 AST* MakeClass(AST* newId, AST* methods) const override {
   AST* classId = meta::id(Class.getName());
   return .< class .~newId {  //new class based on adapted methods AST
              typename .~classId* instance;         //adapted instance
             public:   .~methods;          //insert all adapted methods
              typename .~newId (typename .~classId* o) : instance(o){}
            }; >.; //constructor with the adapted instance as argument
 }
 AdapterByDelegation(const ast::ClassDecl& Class) : Adapter(Class) {}
};
.@class AdapterBySubclassing : public Adapter{//pattern implementation
public:                                       //using subclassing
 AST* MakeCall(AST* name, AST* actuals) const override
   { return .<.~(meta::id(Class.getName()))::.~name)(.~actuals)>.; }
 AST* MakeClass(AST* newId, AST* methods) const override {
   AST* classId = meta::id(Class.getName());
   return .<class .~newId : .~classId { public: .~methods; }; >.;
 }
 AdapterBySubclassing(const ast::ClassDecl& Class) : Adapter(Class){}
};
class Window { //runtime class to be adapted
  public: int  Draw (DC& dc) {/*...*/}
          void SetWholeScreen (void) {/*...*/}
};
.&Adapter::AdapterMap renames{ {"SetWholeScreen", "Maximize"} };
.&const ast::ClassDecl& C = *meta::getClassDecl("Window");
.&Adapter* adapter1 = new AdapterByDelegation(C);//create a generator
.!(adapter1->adapt(.<typename WindowAdapter1>., renames));//generates:
// class WindowAdapter1 {
//          Window* instance;
// public:  int  Draw (DC& dc)   { return this->instance->Draw(dc);  }
//          void Maximize (void) { this->instance->SetWholeScreen(); }
//          WindowAdapter(Window* o) : instance(o) {}
// };
.&Adapter* adapter2 = new AdapterBySubclassing(C);//create a generator
.!(adapter2->adapt(.<typename WindowAdapter2>., renames));//generates:
// class WindowAdapter2 : Window {
// public:  int  Draw (DC& dc)   { return Window::Draw(dc);  }
//          void Maximize (void) { Window::SetWholeScreen(); }
// };
.&delete adapter1; //memory management in stages as in normal programs
.&delete adapter2; //memory management in stages as in normal programs
```