SOFTWARE: PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2015; **45**:727–763
Published online 25 November 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2241

# An integrated implementation framework for compile-time metaprogramming

Yannis Lilis[1,*,†] and Anthony Savidis[1,2]

[1]*Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Crete, Greece*
[2]*Computer Science Department, University of Crete, Heraklion, Crete, Greece*

SUMMARY

Compile-time metaprograms are programs executed during the compilation of a source file, usually targeting to update its source code. Even though metaprograms are essentially programs, they are typically treated as exceptional cases without sharing common practices and development tools. Toward this direction, we identify a set of primary requirements related to language implementation, metaprogramming features, software engineering support, and programming environments and elaborate on addressing these requirements in the implementation of a metaprogramming language. In particular, we introduce the notion of *integrated compile-time metaprograms*, as coherent programs assembled from specific metacode fragments present in the source code. We show the expressiveness of this programming model and illustrate its advantages through various metaprogram scenarios. Additionally, we present an integrated tool chain, supporting full-scale build features and compile-time metaprogram debugging. Copyright © 2013 John Wiley & Sons, Ltd.

Received 18 April 2013; Revised 18 October 2013; Accepted 21 October 2013

KEY WORDS: compile-time metaprograms; multi-stage languages; staged compilation; metaprogramming model

## 1. INTRODUCTION

Multi-stage programming languages [1–3] take the programming task of code generation and support it as a first-class language feature enabling self-transformations, realizing a sort of reification of the underlying language code generator. When code generation becomes a language construct, one may write generator code, which produces other code, the latter becoming an integral part of the main program by substituting its original generator code. In this sense, the generator plays the role of a *metaprogram*, that is, *program producing another program*, while the overall process is recursive if the generated code is also a generator code, causing staging to be nested. Throughout this paper we use the terms stage and metaprogram interchangeably. Staging definitions involve custom syntax commonly referred to as *staging annotations* [3]. The general notion of staging is depicted under Figure 1.

In earlier multi-stage languages such as *MetaML* [4] and *MetaOCaml* [5], code generation by stages occurs only during program execution (runtime staging or runtime metaprogramming (RTMP)). However, the program staging methods may also be applied during program compilation. In this context, compile-time staging, known also as compile-time metaprogramming (CTMP) [7], supports the evaluation of staging definitions and the actual transformations of the main program during the compilation phase. To support such transformations, the manipulated source code fragments are commonly represented as *abstract syntax trees* (ASTs). Examples of languages supporting such a compilation scheme include *Template Haskell* [6], *Converge* [7][8], and *Metalua* [9].

*Correspondence to: Yannis Lilis, Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Crete, Greece.
†E-mail: lilis@ics.forth.gr

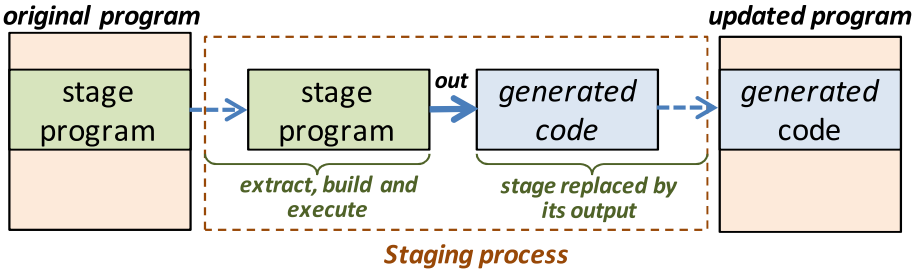



Figure 1. General staging process in multi-stage languages.

Our work falls in the domain of multi-stage languages supporting CTMP and focuses on enabling full-scale metaprogram development through practices and tools commonly available to normal programs. Our primary motivation is that while CTMP is being supported by many languages and is increasingly growing as a development practice over the past few years, it is still offered as a special feature that seems separate from the main language. Metaprograms are usually developed and deployed with no resemblance to normal programs. From a developer perspective, they tend to lack common notions such as files and modules, while from a deployment perspective, they typically adopt a macro invocation policy with no state sharing or the notion of a main control flow.

Moreover, metaprogramming lacks effective support for project management, intelligent editing support and source-level debugging, something restricting larger-scale metaprogram developments. There seems to be no particular intention for such lack of features other than the inherent implementation complexity when trying to accommodate them in languages and tools. As metaprograms are programs, it is irrational to offer diverse development styles amongst the two worlds and to actually provide fewer facilities to metaprograms. In this direction, we emphasize the necessity for a *methodological integration* between metaprogramming and normal programming, featuring common software practices and development tools.

In this context, we first identify a set of prominent requirements related to language features, software engineering, and programming environments to effectively support such integration. Then, we elaborate on the way most identified requirements have been accommodated and implemented in our metalanguage. In this context, we propose the notion of *integrated compile-time metaprograms* as coherent programs assembled from the stage fragments embedded in a source file. Additionally, we introduce extensions to programming environments toward an *integrated tool chain* accommodating metaprograms together with normal programs. In particular, metaprograms are integrated in the: (i) workspace manager, enabling typical code review and source browsing; (ii) full-scale build system (with multiple build threads) supporting all related properties and dependencies; and (iii) source-level debugger offering the full range of features.

The paper is structured as follows. Section 2 establishes the staging terminology used throughout the paper and provides background information related to staging syntax. Section 3 identifies the key requirements for integrated metaprogramming support, while Section 4 elaborates on our proposition, discussing the implementation methods and the proposed metaprogramming model while comparing its expressiveness against the currently prevalent model. Section 5 discusses selected case studies that evaluate and demonstrate the value of the proposed model. Section 6 elaborates on the notion of an integrated tool chain, showing how metaprograms become first-class citizens of the project management, build system, and source-level debugger of integrated development environments (IDEs). Section 7 provides an account of related work reviewing existing metalanguages against the requirements for integrated support. Finally, Section 8 summarizes and draws key conclusions.

## 2. BACKGROUND

Before elaborating on the requirements for integrated metaprogramming and our proposition, we discuss a few common aspects of multi-stage languages. We use the term *stage nesting* to represent the way stages are embedded on each other. In particular, higher stage nesting concerns inner metacode that will be evaluated earlier in the staging process, while lower stage nesting concerns outer metacode



DOI: 10.1002/spe

that will be evaluated later in the staging process. In this sense, in a program with *N* stages, the innermost stage has nesting *N* and will be executed first, while the outermost stage has nesting one and will be executed last. We continue with common features of CMTP languages related to staging syntax and semantics, while discussing the manipulation of source fragments as ASTs and the key notion of quasi-quoting.

### 2.1. *Manipulating source fragments as ASTs*

Because CTMP involves generating, combining, and transforming source code, 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 very low level to be deployed. Currently, ASTs are widely adopted for representing source code, because of their ease of use and because they retain the original code structure.

### 2.2. *Notion of quasi-quoting*

Although ASTs provide an effective method for manipulating source code fragments, manually creating ASTs for source fragments usually requires a large amount of statements, making it hard to identify the actually represented source code [10]. 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 a feature known as *quasi-quoting* [11]. Also, quasi-quotes allow defining templates as source code with placeholders, while they can serve as staging annotations that distinguish the metaprogram from the object program [12], that is, the code it operates on. To better illustrate the notion of quasi-quoting, we briefly discuss its support in various staged languages with an example.

Consider the following *Lisp* [13] macro that generates the multiplication of the argument X by itself. Definitions after the *backquote* operator (‘) are not directly evaluated but are interpreted as a code fragment value (i.e., an AST). The reverse of *backquote* is the *unquote* operator (,) that escapes the syntactic form and inserts its argument directly in the expression being created. With the macro being invoked with argument 5, the result is the expression (* 5 5) that yields 25.

```
(defmacro square (X)
  `(* ,X ,X))
 (square 5) ; 25
```

The same example in MetaML follows, where surrounding *brackets* <…> are used to turn code fragments to ASTs (called delayed computations in MetaML), and *escape* ~ enables combination of such ASTs within bracket expressions. This means that *square* contains the AST of 5*5. Finally, *run* is used to evaluate an AST (called execute of delayed computation in MetaML), which in our example evaluates to 25.

```
val code = <5>;
val square = <~code * ~code>;
val result = run square; (* 25 *)
```

In Converge, the example looks quite similar, with a few syntactic changes regarding the staging annotations: code within quasi-quotes *[|…|]* is converted to AST, while *insertion ${…}* and *splice* $<…> operators relate to *escape* and *run* of MetaML.

```
code := [| 5 |]
square := [| ${code} * ${code} |]
result := $<square> // 25
```

The same example follows as Metalua code: quasi-quotes are denoted with *+{…}* while *–{…}* implies splicing if inside quasi-quotes or execution otherwise.

```
result = -{ block: code = +{ 5 }
                  return +{ -{code} * -{code} }
} -- 25
```



DOI: 10.1002/spe

Finally, in our multi-stage implementation within the *Delta* language [14, 15], the same example is as follows. Quasi-quotes are denoted with << … >> tags, escapes are denoted with ~, while the operator for code generation is *!(…)*. The *&* operator shown is used to denote arbitrary stage code.

```
// 'code' is a stage var assigned the AST of const 5
&code = << 5 >>;
// 'square' is stage var assigned the result of AST composition
&square = << ~code * ~code >>;
// staged generation (inlining) of the code carried by 'square'
result = !(square); // 25
```

Apparently, the discussed languages adopt quasi-quoting in a similar way. There is however a subtle yet important difference between them. Most of the aforementioned languages evaluate staging during compilation while MetaML in particular applies staging during runtime. For the specific examples, this makes no actual difference. However, in general, there may be differences because CTMP and RTMP implementations have access to different information.

## 3. REQUIREMENTS

The practicing of metaprogramming is strongly affected by the ability of related language features and tools to support software engineering. In this context, we continue with the prominent software engineering requirements that are essential for the integrated practicing of metaprograms and normal programs.

It should be noted that such requirements are derived from the weaknesses of existing metalanguages that compromise the software engineering of metaprograms. That is, they are criteria directly affecting the practicing of metaprogramming and constituted the focal point of our work. Overall, we consider metaprogramming to be fundamentally harder to normal programming. However, the restricted software engineering support by existing languages makes it even harder, sometimes rendering metaprogramming to a dark art for average programmers. As it becomes evident, it is the bar regarding metaprogramming facilities that is actually raised by such requirements to a level similar to normal programming.

### *3.1. Exploiting normal language features and tools*

One of the most important requirements toward integrating normal programs and metaprograms is the full exploitation of all normal language features and tools in the context of the metalanguage. It is essential that metaprogrammers experience the metalanguage as an extension on top of the normal language, rather than as a restriction of it. For instance, if the normal language supports classes, threads, and modules, the metalanguage should also provide them in the same manner. This may seem apparent at a first glance, but in fact, it is not currently met by most languages. We briefly explain the implications of this requirement in the practicing of CTMP.

There are two reasons justifying why the exploitation of all normal language constructs in the metalanguage is critical. First, in implementing a metaprogram, one should not be given fewer features than what is offered in implementing a normal program. Second, even when alternative equivalent facilities are offered, it is difficult and painful for programmers familiar with the normal language to learn and deploy a different set of constructs for similar programming tasks. Overall, the normal language should be fully reused in expressing and organizing the metaprogramming logic, with additional syntax and semantics introduced only where necessary.

To outline the importance of this requirement, consider C++ templates [16], which can support some level of CTMP [17, 18]. Templates are essentially a special-purpose functional language, interpreted during compilation, being fundamentally different from the class-based imperative nature of the normal language. The latter requires so radically diverse approaches to handle similar problems that reuse of design or code is disabled. For instance, consider the following C++ code for the Fibonacci sequence:



DOI: 10.1002/spe

```
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
```

Performing the same computation during compile-time with templates requires a feature known as recursive template specialization:

```
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
```

Clearly, the two versions of the *Fibonacci* function are fundamentally different. In particular, the second one is far less readable and obvious as it widely deviates from the common style of the language and involves custom coding practices.

In general, we argue that there should be no particular language-oriented distinction between normal functions and metafunctions. They may differ in terms of their operational role, with the latter typically implementing some AST manipulation logic, but other than that, there should be no fundamental difference between them. For example, the following Delta code implements the Fibonacci function (almost identical to the C++ version) and can be used for computations both at compile-time and runtime.

```
function fibonacci(n) {
      if (n == 0 || n == 1) return 1;
      else return fibonacci(n - 1) + fibonacci(n - 2);
}
// staged invocation prints 8 at compile-time (compiler output)
&std::print(fibonacci(5));
// runtime invocation prints 8 at execution (program output)
std::print(fibonacci(5));
```

Apart from the thorough exploitation of all language features, the entire set of language tools should be reusable as well, including the compiler, runtime library, virtual machine, and debugger. In other words, metalanguage development should avoid reinventing the wheel and emphasize tool reuse, while appropriately extending or refining where needed according to the extra metaprogramming requirements. In particular, the original compiler and virtual machine may be extended to translate and execute respectively both normal programs and metaprograms. Similarly, by extending the original debugging system of the language, source-level debugging of metaprograms should be facilitated as with normal programs.

### *3.2. Supporting context-free and context-sensitive generation*

Metaprogramming is commonly used like a macro system to transform a source fragment by inserting extra source code through code generation directives. In this framework, there are two possible options (Figure 2) in controlling the insertion context: (i) *context-free*: the code is inserted at the source point of the generation directive and (ii) *context-sensitive*: information on the current AST context is provided to the generation directive enabling code insertion at any AST locations reachable by the supplied context.

The context-free case is commonly called *splicing* or *inlining* and covers all cases where the generator logic does not require awareness of the code insertion context. In this case, the generator operates only on its arguments, if any, and produces a source fragment that is inserted by replacing the original generator directive in the AST. Context-free generation is very common in metalanguages, while frequently, the only available option as in Lisp, Scheme [19], MetaML, Metalua, and Converge.

The context-sensitive case is more general, accounting to all scenarios where the generator logic needs to decide the actual code insertion context. Additionally, the generator may insert code



DOI: 10.1002/spe

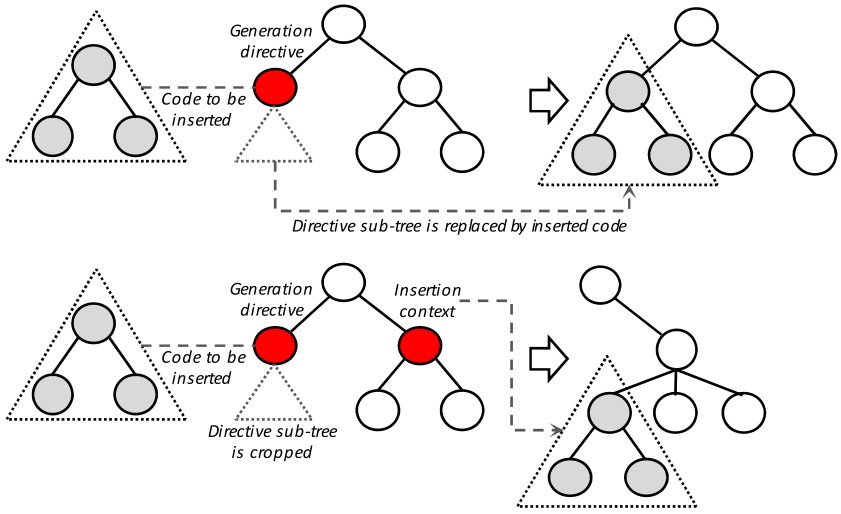



Figure 2. Context-free (top) versus context-sensitive (bottom) code generation.

fragments at multiple different locations, not merely in a single context. Typically, the latter involves an AST search by the generator logic in order to locate the appropriate target contexts. Examples of context-sensitive transformations relate to meta-attribute definitions of *Nemerle* [20] and the built-in context variable in *Backstage Java* [21] providing access to the surrounding AST.

As we elaborate later, Delta offers both options for staged code generation. Context-free generation is directly supported through the already discussed inline tag *!(…)*. Additionally, context-aware generation is facilitated by providing access to the main program AST through a special compile-time library function named *std::context*(). The latter allows obtaining the parent AST node of its invocation point from the AST of the outer stage or main if no enclosing stage exists. Because of this behavior, its result always carries the current context and can be freely used to iterate and manipulate as needed by stage code. The latter case is discussed in more detail under Section 4.2.5.

### *3.3. Composing and generating all language constructs*

While metaprograms eventually generate code, they always reflect some kind of source fragment composition logic according to particular design demands. Usually, reuse is the primary motivation leading to composition and is frequently practiced by designing code skeletons or templates. In such a typical scenario, reused code clearly concerns the entire range of language constructs, while recurring code patterns become code skeletons with composition and insertion applied by metaprogramming. Now, once this type of reuse is anticipated for normal programs, there is no particular reason to be excluded for metaprograms.

In other words, metaprograms are programs too, thus deserving all features available to normal programs, including the ability to reuse any repeating metacode. For the latter, it is essential that the metalanguage enables expressing and composing metacode as with normal code. In conclusion, we need to enable all kinds of metatags, including generator directives, to be freely quasi-quoted, manipulated, and composed in the form of ASTs. To illustrate this requirement, we provide a simple example from a text-based macro system, in particular, the C preprocessor [22] where the feature is missing. As a result, macros generating further macro definitions are disabled. For instance, consider the following C macro and its use:

```
#define SINGLE_ARG_MACRO_GENERATOR(name, arg, replacement) \
    #define name(arg) replacement
SINGLE_ARG_MACRO_GENERATOR(SQR, x, (x)*(x))
```

After the preprocessing stage, the resulting source text is as follows:



DOI: 10.1002/spe

```
#define SQR(x)  (x)*(x)
```

The latter is invalid as pure C code and results to a compile error. A solution for this problem would be to apply multiple preprocessing stages instead of a single one.

Another requirement relates to the practical limitations of quasi-quoted code to handle more comprehensive scenarios of code composition. In particular, quasi-quotes express code fragments with a constant structure, known at compile-time. They cannot express structures being the outcome of computation, such as *if* statements with a variable number of *else if* clauses. To allow generating such dynamic patterns, the metalanguage should provide extra facilities for manipulating AST values including methods to traverse ASTs. This may be achieved either through extra custom constructs such as algebraic data types for trees in Metalua or via special library functions such as the *ITree* functions of the *Compiler.CEI* interface in Converge.

Finally, a known issue related to generating code that introduces names is *variable capture* [23]. It concerns the potential of name conflicts between the inserted code and the code already available at the insertion site. The earliest approach to eliminate such conflicts was introduced in Lisp with the mandatory use of *gensym* in all macros involving temporary variables. Later, the problem was better addressed in Scheme through hygienic macros showing the importance of code generation without having to explicitly address potential name collisions.

However, there are still cases where variable capture without automatic renaming is indeed the desired effect, for instance, when separately generating code for definitions and code for their actual use. In such scenarios, we should enable programmers conditionally disable the hygienic behavior and preserve the original names in the generated code. In conclusion, it is important to support both cases by enabling the selection of either hygienic generation or name capture.

As we discuss later, Delta covers all these code generation requirements. Using quasi-quotes, a programmer may express any language construct including metatags as well as generate both hygienic and capturing names. Additionally, Delta provides a complete AST library facilitating the creation, traversal, and manipulation of ASTs. Regarding the aforementioned example of a macro-generating macro, the following Delta code shows how a metafunction (i.e., prefixed with *&* meaning defined in first stage) can actually generate another metafunction.

```
// metafunction generating another metafunction: it returns
// a function definition prefixed with the execute & tag
&function SINGLE_ARG_MACRO_GENERATOR(name, arg, replacement)
  { return << &function ~name(~arg) { return ~replacement; } >>; }
// actual invocation of the metafunction as part of an inline ! tag
!(SINGLE_ARG_MACRO_GENERATOR(<<SQR>>, <<x>>, <<x * x>>))
// this is the result of the previous inline directive
&function SQR(x) { return x * x; }
```

### 3.4. Sharing and separation of concerns among stages and main

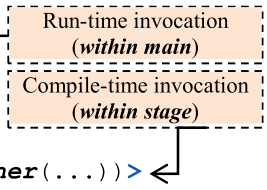
In general, all stages share the common role of transforming the source code of their embedding program. In particular, outermost stages transform the source code of the main program, while inner stages transform the source code of their enclosing stage. To realize the required transformations, stages will typically implement functionality for creating, editing, and inspecting the necessary source fragments. Because similar types of generated source patterns may well reappear, reuse and sharing of functionality across stages is prominent.

Besides source code manipulation, stages as programs will require all sorts of utility functions commonly needed in normal programming. When such functionality is also required in the main program, sharing between stages and main program is inevitable. In CTMP, the latter means such functionality is available both at runtime, by main, as well as during compile-time evaluation, by stages. For example, consider the following Converge example, where a function for some custom data container is shared between runtime and compile-time evaluation.



DOI: 10.1002/spe

```
func CreateAndPopulateContainer(...):
   ...
func CompileTimeCalculation(container):
   ...
func main():
   container := CreateAndPopulateContainer(...)
   $<CompileTimeCalculation(CreateAndPopulateContainer(...))>
   ...
```




Alternatively, the shared container functionality can be better organized as a library deployed both by main and stages. But again, it is not always a best practice to produce a library just to reuse some common code between stages and main. Thus, the language should offer both options, by enabling code sharing and library deployment, while letting programmers choose the one better fitting a situation.

Besides any possibly shared functionality between stages and main, each should remain a distinct program with its separate hidden definitions and execution state. In this sense, encapsulation should also be supported to enable separation of concerns and thus facilitate modular staging. An example of encapsulation is shown in the following Metalua code, with function *CreateAST* being available only during compilation (stage) and function *Print* being available only during runtime (main).

```
-{ block: function CreateAST() return +{1} end }
function Print(x) print(x) end
Print( -{CreateAST()} )
```

Delta supports both sharing and separating functionality across the compilation stages and the main program. In particular, main program functions with no access to runtime state are directly visible and available to all stages. Finally, when we have multiple stages, *&* tags can be nested to specify that a function is available only in a specific stage. This functionality is illustrated in the following Delta example.

```
function    f(){...no access to main state...}  // available across stages and main
&function   f1(){...}                           // available only in stage 1
&&function  f2(){...}                           // available only in stage 2
function    g(){...has access to main state...} // available only in main
```

### 3.5. *Programming model for stages equal to normal programs*

Normal programs can be decomposed into separate modules, enabling sharing of functionality and state, while realizing a common global control flow. The present situation with stages is very different from this notion because of a custom and arguably impractical programming model commonly offered. While theoretically, the existing model renders stages as expressive as normal programs, this remark refers to computability only and has little value in the software engineering quality of the model itself. More specifically, as depicted under Figure 3, stages are evaluated as independent transformations that operate on their input source fragments and eventually affect their enclosing program. In this sense, they resemble macro invocations of traditional macro systems, running sequentially and within independent execution sessions, effectively operating as batches.

As shown, stages of the same nesting level always input from and output to their enclosing source text, meaning they practically operate on the same data. Thus, conceptually, their concatenation may comprise a single larger stage program affecting the enclosing program. Now, following the current practice, the evaluation of stages at the same nesting level can be actually interleaved with the evaluation of inner stages. Thus, the conceptual model of joining stages into a single program is not actually mapped into a corresponding sequential, lexically scoped, control flow.

Additionally, stages are evaluated as independent programs. Then, to have some kind of state sharing across stages, one should rely on custom implementation features. In particular, under interpreted language implementations, one may deploy shared global environments or dynamic scoping to feature persistent variables across the multiple interpreter invocations for stages. Not only are the latter merely implementation workarounds and not a standard property of the stage



DOI: 10.1002/spe

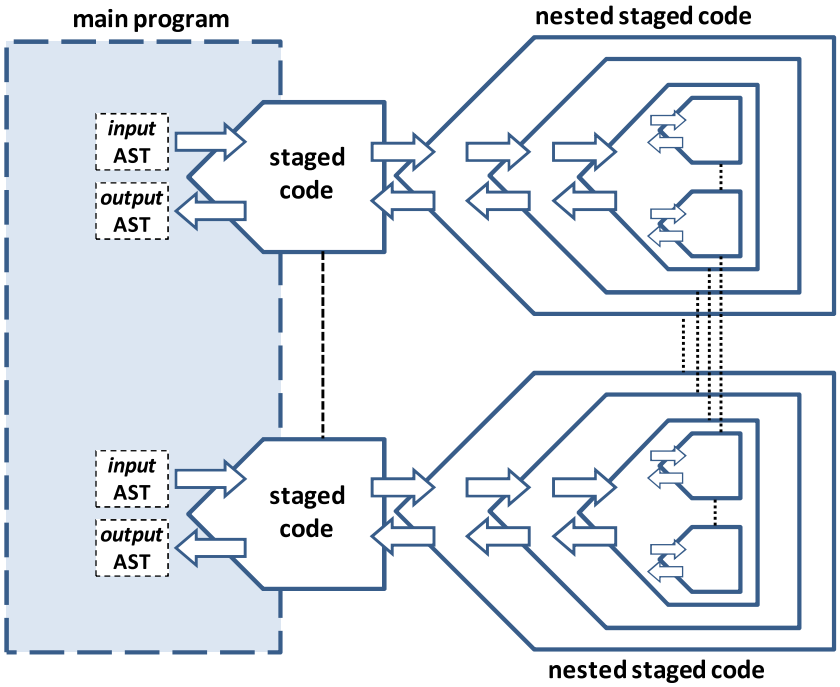



Figure 3. Common evaluation of stages in popular multi-stage languages (e.g., MetaML, MetaOCaml, Converge, Metalua, etc.) and macro systems (e.g., Lisp and Scheme): inside-out for nested stages and top-down for top-level stages, all as independent execution sessions. Dotted lines connect stage fragments of the same nesting level whose concatenation could comprise a single stage program.

programming model, but it turns all stages, inner or outer, to a single program with a common shared state.

In general, the notion of a single program comprising only stages of the same nesting enabling state sharing and common control flow is not supported. In fact, most languages with compiled stages have no state-sharing workaround similar to interpreted languages. The latter disables even very simple tasks, such as implementing conditional source code insertions relying on information produced by the evaluation of preceding stages in the source text. To our knowledge, the only language that partly supports the aforementioned notion is Metalua. In fact, Metalua allows sharing state among stage code at the same nesting, as shown by the following example.

```
function Car() return {…} end

--No extension so BasicCar is same as Car
-{block: extra = +{block:}}
function BasicCar() local car = Car() -{extra} return car end

--Add ABS, so ABSCar is Car + ABS
-{block: extra = +{block: -{extra} car.abs = function () end}}
function ABSCar() local car = Car() -{extra} return car end

--Incrementally add Turbo, so ABSTurboCar is Car + ABS + Turbo
-{block: extra = +{block: -{extra} car.turbo = function () end}}
function ABSTurboCar() local car = Car() -{extra} return car end

--Incrementally add 4WD, so ABSTurboWD4 is Car + ABS + Turbo + 4WD
-{block: extra = +{block: -{extra} car.WD4 = function () end}}
function ABSTurboWD4() local car = Car() -{extra} return car end
```

In this example, there is a stage variable named *extra* carrying a code block as an AST that is added to implementations of *Car* constructor functions. The code block is initially empty,



DOI: 10.1002/spe

that is, *{block:}* in Metalua, and is then incrementally extended with additional statements by the successive stage blocks; for the example to work, the *extra* variable should be shared across such blocks.

The latter is true in Metalua because stages are not evaluated by the original language virtual machine but by a custom interpreter supporting dynamic scoping and a common shared state across all stage executions, including nested ones. In other words, although Lua is compiled, in Metalua, stages are actually interpreted. As a result, any inner stage can access and overwrite *extra* thus breaking state encapsulation on individual stages. Moreover, Metalua adopts the common multi-stage language evaluation order where stage execution is interleaved. Thus, there is no notion of sequential control flow for stage code at the same nesting.

Stages in existing languages are commonly evaluated in a depth-first fashion with either recursive interpreter invocations or successive compilation and execution rounds. Effectively, the current prevalent models for stages are two: (i) if interpreted while offering state sharing among evaluations, then the stage code collectively behaves as one big program or (ii) if compiled or interpreted without state sharing, then stage code is totally fragmented and disjointed, executed as independent sessions. We consider these two options to be special and limit cases, severely restricting the chances for deploying common software engineering practices on stages. In this context, we argue that a programming model for stages is needed joining stage code of the same nesting into a separate coherent program, with lexically scoped control flow, enabling software engineering practices on stages as with normal programs.

Our multi-stage implementation of Delta offers such a programming model by collecting the stage code fragments of the same nesting, following their order of appearance in main, and executing them as a single program. There is no interleaving with any other stage execution. For instance, consider the Delta code for the previously discussed example regarding *Car* constructor functions.

```
function Car() { return […]; }

&extra = nil; // empty AST
function BasicCar() { local car = Car(); !(extra); return car; }

&extra = << ~extra; car.abs = (function (){…}); >>;   // extend AST
function ABSCar() { local car = Car(); !(extra); return car; }

&extra = << ~extra; car.turbo = (function (){…}); >>; // extend AST
function ABSTurboCar() { local car = Car(); !(extra); return car; }

&extra = << ~extra; car.WD4 = (function (){…}); >>;   // extend AST
function ABSTurboWD4() { local car = Car(); !(extra); return car; }
```

The aforementioned stage code is collected and executed as a single program as follows. Notice that the *std::inline(extra)* calls are automatically generated from the respective !*(extra)* metacode segments of the original code to handle code generation. A more detailed discussion on stage assembly and evaluation is provided in Section 4.2.3.

```
extra = nil;
std::inline(extra);
extra = << ~extra; car.abs = (function (){…}); >>;
std::inline(extra);
extra = << ~extra; car.turbo = (function (){…}); >>;
std::inline(extra);
extra = << ~extra; car.WD4 = (function (){…}); >>;
std::inline(extra);
```

We should note that even for the other languages discussed later, an advanced user or the language developer may find a way to emulate the semantics of a lexically scoped control flow and state sharing for stage code. In this context, our emphasis is not put on expressiveness, meaning we do not argue that the model cannot be implemented in any of these languages. Instead, our focus is on the optimal delivery of the model to programmers in the most straightforward manner, as easy as with normal programs.



DOI: 10.1002/spe

### *3.6. Treat stages as first-class citizens of the programming environment*

Currently, compiled stages are evaluated as part of the build process in a way that is transparent to programming environments. For example, there is no support for build dependencies and flags on stages as with all other programs. In this context, stages should become first-class citizens of the programming environment supporting: (i) reviewing and browsing the code of stages and the actual program transformations they introduce; (ii) improved source editing of stages through staging-aware editing facilities; and (iii) a stage-aware build process. We continue by detailing the necessity for including such features in metaprogramming environments.

*3.6.1. Source browsing and editing.* When a program that involves metaprogramming does not behave as expected, it is usually difficult to directly determine the cause. The reason could be a faulty implementation of the metaprogram, wrong deployment, or even some error in the logic of the final program. Because programmers only view the original source code, they cannot observe the transformations performed by the metaprogram. Moreover, the metaprogram implementation may itself be generated by another metaprogram that is never part of the main source. Converge offers a solution to this problem by relating errors to all parts of the transformation path [8], thus allowing users to debug relatively easy. We consider an alternative solution, targeting to provide programmers with a view of the source code of their metaprograms as well as the transformations they perform on the main program. Apart from debugging, such a view also allows browsing through the various source code structures, providing easy access and navigation across modules, classes, functions, variables, and so on. This is especially important in cases where such code is not available in the original source but generated via metaprogramming and allows programmers to better understand the structures and functionality available in the generated code.

Apart from source browsing, another IDE facility that greatly improves the software development process relates to source editing. Assistance in source editing, also referred to as *IntelliSense* [24], is generally provided in the form of tooltips that display information regarding various language expressions and symbols (e.g., argument names, symbol types, external documentation, etc.) or in the form of auto-completion. When it comes to metaprogramming, such features are invaluable as they can provide information that is not directly available from the editing context. A metafunction invocation may introduce multiple declarations to be used in the generated code that never appear in the original source text. Nevertheless, editing in a subsequent context should provide auto-completion support for them as if they were part of the original source. An example of this functionality is depicted under Figure 4, where both the metaprogram (i.e., the *GENERATE_CLASS* macro) and the result of its evaluation (i.e., the generated class *X*) support IntelliSense information.

In the context of multi-staging, such functionality is even more important as small changes in the editing context (e.g., inside or outside a staging annotation) may result in a different stage and thus different set of visible symbols. In this direction, a programming environment should produce symbolic information for staged definitions being utilized to offer IntelliSense features as for non-staged definitions.

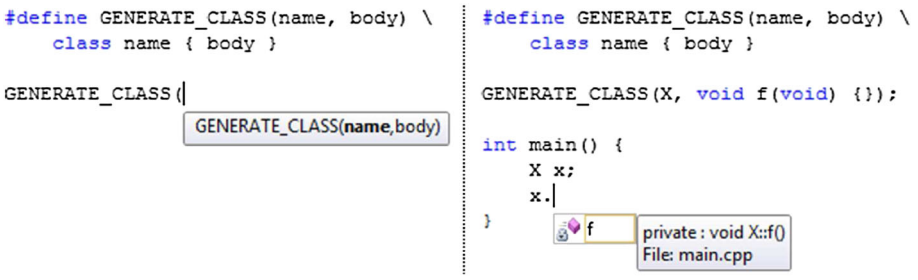



Figure 4. Supporting IntelliSense information for both metaprograms (left) and their outcomes (right).



DOI: 10.1002/spe

*3.6.2. Build tools.* While metaprograms are encapsulated in a main program, they may require external libraries or compile flags that vary from those required by other metaprograms or the main program. For example, consider a metaprogram generating code for a GUI application. The main program will typically require a graphical library. However, such a library is likely not needed in the generator metaprogram. Similarly, while most metaprograms will need to utilize an AST library, the main program will usually not. Consequently, programmers should be allowed to specify custom build options on metaprograms as they can on normal programs.

Besides build flags, typical build dependencies may emerge on stages too, which should be handled similarly to normal program, that is, building any dependencies prior to stage compilation. For this to work on stages, we need to build the deployed modules prior to stage compilation, all during the compilation of the main program. But the actual build process is not handled solely by the compiler because it requires information present in the build system of the programming environment. Practically, this implies interplay between the compiler and the build system to build stage dependencies prior to actual stage compilation. To avoid rebuilding if stages are up-to-date, checking of respective stage binaries and their dependencies is also required.

## 4. INTEGRATED METAPROGRAMS

We propose a new programming model for stages that we call *integrated* because it allows software engineering of metaprograms in a way similar to normal programs. Overall, the generative nature of metaprograms is treated as any other functional characteristic that programs may have, meaning no methodological separation between the two worlds is necessary. In this model, independent snippets of stage code at the same nesting are treated as a unified program, with a lexically scoped control flow, shared program state, and scoping rules of the main language. Additionally, all normal language features are available in implementing stages.

We continue by first elaborating on the programming model. Then, we brief our metalanguage constructs to support the model and discuss the semantics regarding the assembly of stage snippets in order to form integrated metaprograms. Finally, we show the expressiveness of our model in comparison with the prevalent existing model and discuss the trade-offs involved in choosing one model over the other.

### *4.1. Programming model*

As already mentioned, an integrated metaprogram is composed by the concatenation of stage code at the same stage nesting with their order of appearance in the main source. Because of this assembly, their evaluation is essentially the sequential execution of their constituent source fragments thus denoting a lexically scoped control flow sequence within the integrated metaprogram. Because the concatenated stage fragments may encompass generation directives, an integrated metaprogram behaves as having multiple input and output locations within its *enclosing program*. We use here the term enclosing program and not just main program because for nesting levels above one, the resulting integrated metaprograms are hosted within other integrated metaprograms. In Figure 5, an illustration of the integrated metaprogramming model is provided, depicting source transformations, stage assembly, evaluation order, and lexically scoped (sequential) control flow.

In stage code, any feature available in normal programs can be used, like performing typical file I/O, launching GUIs to possibly interact with programmers in tuning code generation behavior, handling network connections and communication, loading dynamically linked libraries, and so forth.

The integrated metaprogramming model compared with fragmented stage code reflects a fundamental methodological shift concerning transformations. In particular, we treat transformations as any other program function. Effectively, because 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. Overall, segregating the stage fragments of the same enclosing program serves no particular goal and only complicates the engineering of metaprograms. In summary, the following rounds are repeated for stage evaluation until no stages exist:



DOI: 10.1002/spe

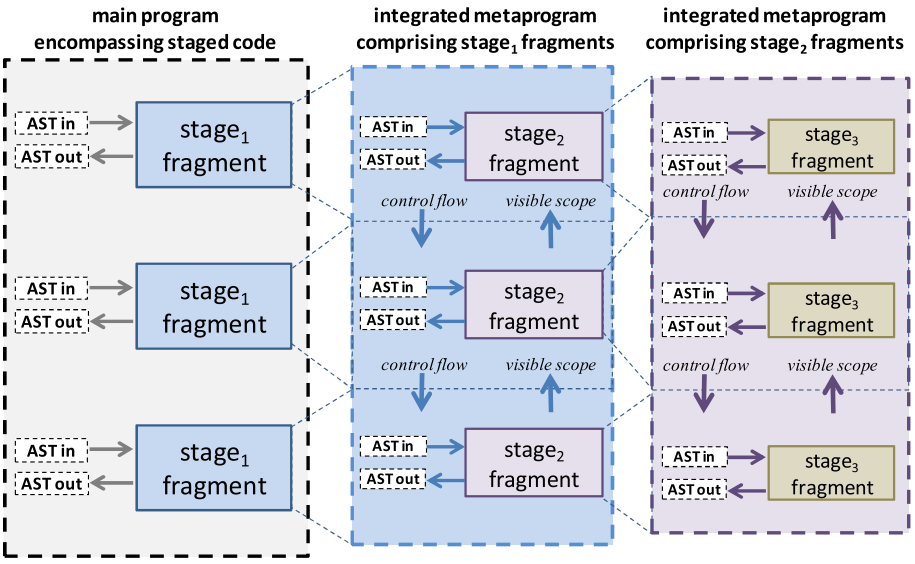



Figure 5. Concept of integrated metaprograms: (i) comprising all stage fragments at the same nesting in their order of appearance; (ii) denoting a sequential control flow among stage fragments; and (iii) providing scope visibility to previous stage fragments.

1. determine innermost stage nesting level
2. assemble integrated metaprogram for this nesting level
3. build and execute

It should be noted that the result from the evaluation of each integrated metaprogram is a transformed version of main called *intermediate main*. Besides the first round using the original main, all the rest collect stage code from the current intermediate main that is then transformed to the next one. Eventually, the intermediate main from the last evaluated metaprogram is not staged and is called *final main*. It is compiled to binary constituting the output of the entire multi-stage compilation process.

From the aforementioned process for stage evaluation, it is evident that our proposed model reflects a different evaluation order compared with the traditional practice of top-down and inside-out evaluations of current multi-stage languages. For example, consider the following nested staged code, where $f_1$, $f_2$, $g_1$, and $g_2$ are staged expressions and *!* denotes metaprogram invocations.

```
!f1(!f2());
!g1(!g2());
```

The traditional evaluation order in current multi-stage languages is

$$!f_2 \rightarrow !f_1 \rightarrow !g_2 \rightarrow !g_1$$

The evaluation order with integrated metaprograms is

$$\{\ !f_2 \rightarrow !g_2\ \} \rightarrow \{\ !f_1 \rightarrow !g_1\ \}$$

The brackets are used to denote that enclosed staged expressions are executed as a single coherent program and not as isolated invocations. Clearly, the two orders are different. A general form of this example showing the evaluation order between the two approaches is illustrated under Figure 6.

### 4.2. Metalanguage

The reported work has been implemented as the multi-stage extension of the *Delta* language [14, 15], which is untyped object-based, compiled to byte code, and run by a virtual machine. In this context, the entire set of main language features are available in stage code, while only the language



DOI: 10.1002/spe

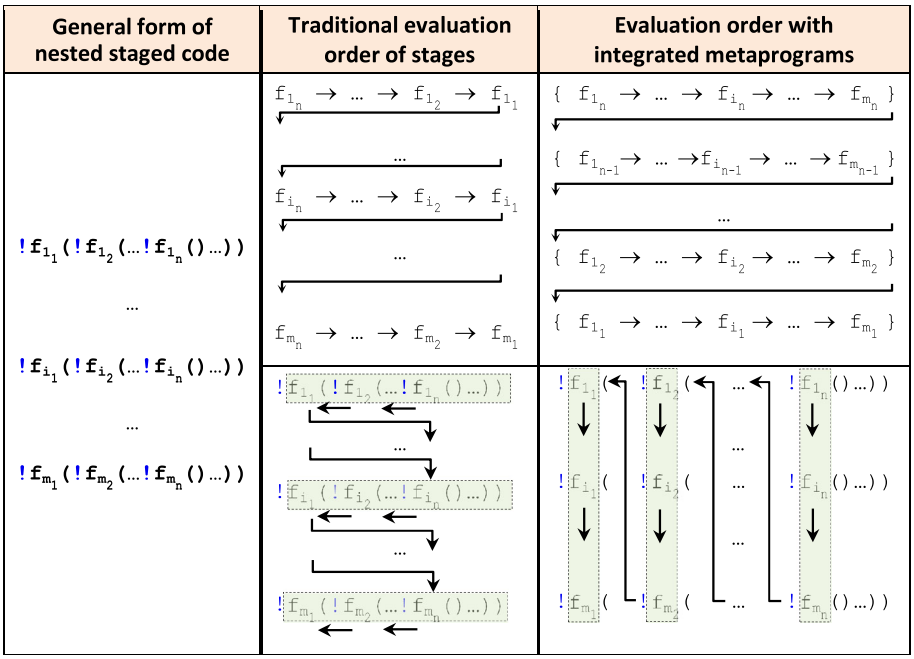



Figure 6. Typical evaluation order in multi-stage languages—$f_{ij}$ are staged expressions with *i* enumerating staged code blocks and j denoting the stage nesting in the respective block.

compiler has been extended to accommodate staging functionality. The original virtual machine has not been modified, while it is directly deployed for stage evaluation. In the following sections, we briefly outline the staging syntax and semantics for our implementation of integrated metaprograms.

*4.2.1. AST tags.* Such tags allow directly converting source text into ASTs, involve no staging at all, and are translated into calls that create ASTs by parsing source text or combining other ASTs together. This is the same approach followed by Template Haskell and Converge and could in fact be adopted to introduce similar tags in languages with no CTMP as extra syntactic support on a reflection library (runtime parsing, translation, and execution). Besides quasi-quotes and escape that appear in many multi-stage languages, we also introduce *delayed escape*, which is essential to metagenerators.

> *Quasi-quotes* (written <<...>>) may be inserted around definitions, such as expressions, statements, functions, and so on, to convey their AST form and are the easiest way (not the only one) to create ASTs directly from source text. For instance, <<*1 + 2*>> is the AST for the source text *1 + 2*, while *ast_make_const (3.14)* produces the AST for the numeric constant *3.14*. Variables within quasi-quotes are resolved with their names in the context where the respective AST is finally inserted, that is, are lexically scoped at the insertion point. For instance, <<*x = 1*>> does not bind to any *x* visible at the quasi-quote location. In the Delta language, variables are lexically scoped and are declared by use the first time a name is met. Thus, if no *x* is defined at an insertion point, a new *x* is introduced by the assignment.

To prevent variable capture and support hygienic macros, we allow in quasi-quotes alpha-renamed variables using special syntax. Those are given automatically contextually unique identifiers. In particular, <<*$x*>> denotes that *x* will be given a fresh unique name at the insertion context. In most metalanguages, such hygienic behavior is active by default with name capture enabled only through special syntax.

We have purposefully chosen such an inverse activation policy because we consider it to more frequently fit the actual use of metaprograms. More specifically, metaprograms may produce: (i) complete named elements such as classes, functions, methods, constants, namespaces, and generics



DOI: 10.1002/spe

that may be directly deployed; (ii) template code fragments to be filled in with other code fragments; and (iii) other non-template code fragments that may be further combined.

In case of named elements, the supplied name will be directly used for deployment; thus, name capture is the only way. When generating non-template code fragments, those may be further composed together or inserted in templates. In this case, the statements of such code fragments may erroneously capture earlier or outer variables. The latter is avoided easily in the respective generator by always declaring generated variables as local and enclosing related statements in blocks. Finally, in case of templates, it is possible that 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, on the basis of the previous remarks, we considered that for most scenarios, name capture would suffice and that for the template cases, hygiene may be deployed where required.

Finally, we allow quasi-quotes to be arbitrarily nested, something useful in implementing *metagenerators*. For example, $<< <<1 + 2>> >>$ is a nested quasi-quoted expression whose generation produces the quasi-quoted expression$<<1 + 2>>$.

*Normal escape* (written *~(expr)*) is used only within quasi-quotes to prevent converting the source text of *expr* into an AST form by evaluating *expr* normally. Practically, escape is used on expressions already carrying AST values that need to be combined into an AST constructed via quasi-quotes. For example, assuming *x* already carries the AST value of $<<1>>$, the expression $<<\sim x + 2>>$ evaluates to $<<1 + 2>>$. The latter also applies in nested quasi-quotes, meaning the expression $<< <<\sim x + 2>> >>$evaluates to $<< <<1 + 2>> >>$. Additionally, we also support the escaped expression to carry scalar values such as number, Boolean, or string (i.e., *ground* values). In this case, the value is automatically converted to its corresponding AST value as if it were a constant. For instance, if *x* is *1*, then *~x* within $<<\sim x + 2>>$ will be converted to the AST of value *1* or $<<1>>$; thus, $<<\sim x + 2>>$ evaluates to $<<1 + 2>>$.

*Delayed escape* (written *~…~(expr)*) is used when escape evaluation should be deferred, something common in metagenerators. The number of tildes is the initial *nesting*, which for normal escapes is one. Then, escape evaluation, being performed when quasi-quotes are evaluated, is applied as follows:

$$eval(escape(n, expr)) = \begin{cases} escape(n-1, expr), & n > 1 \\ expr, & n = 1 \end{cases}$$

Notice that the previous evaluation is not recursive—it returns either the escaped expression or a new escape with decreased nesting. Practically, delayed escapes will be at some point inlined into generated quasi-quotes. The following examples simply outline the behavior of delayed escape (*gen* denotes code generation with an AST parameter). Later, once the details of the staging tags and integrated stage assembly are explained, more elaborate examples with metagenerators are discussed showing the importance of delayed escapes.

- Writing $<< <<\sim\sim x>> >>$ represents the AST of $<<\sim x>>$.
- With $y = << \sim\sim x >>$, the expression $gen<< <<\sim y>> >>$ yields $<<\sim x>>$.

The introduction of delayed escapes allows generating escapes and preserves syntactically their presence within quasi-quotes. The latter is in contrast to the single escape preserving the value it carries. One could also think of a library function such as *esc(ast,n)* to generate a chain of *n* parent escapes on the supplied syntax tree. Now, while the latter would produce tree forms identical to the delayed escape ones, without *~$^n$(expr)* in the language syntax, the resulting trees would be syntactically ill-formed. In fact, in our implementation, all syntax trees composed through library functions pass internal syntactic validity checks, meaning the output of such library function would be directly rejected. We consider delayed escapes to serve a purpose similar to quasi-quotes: one may live without them, but using them makes life easier.

The fact that all the aforementioned tags do not introduce staging as such but are essentially facilities for AST manipulation is depicted under Figure 7. As shown, quasi-quotes are shortcuts



DOI: 10.1002/spe

| *AST tag expressions* | *Respective intermediate code* |
|---|---|
| `<<1 + g()>>` | `ast_create $0 "1 + g()"` |
| `<<~(f(x)) + 2>>` | `param x`<br>`call f`<br>`getretval $0 #carries f(x)`<br>`ast_create $1 "~(f(x)) + 2"`<br>`ast_escape $1 $0 #inserts f(x)` |
| `<< << ~~x >> >>` | `ast_create $0 "<< ~~x >>"` |
| `<< << ~~x + ~y >> >>` | `ast_create $0 "<< ~~x + ~y>>"`<br>`ast_escape $0 y #inserts y` |
| `<<f(~a, ~b)>>` | `ast_create $0 "f(~a, ~b)"`<br>`ast_escape $0 a #inserts a`<br>`ast_escape $0 b #inserts b` |

Figure 7. Code generation examples for quasi-quotes and escapes showing they do not involve staging.

for AST creation (*ast_create*), the latter in our language handled with internal parser invocations. Similarly, escapes (*ast_escape*) involve AST composition operations, again without staging, and are only invoked for normal escapes.

*4.2.2. Staging tags.* Staging tags generally imply compile-time evaluation of associated source code and are essential in supporting CTMP. Syntactically, they define the boundaries between stage code fragments and also introduce stage nesting, also known as metalevel shifting. Besides *inline* (generation) and *execute* (metalevel shifting), appearing in various metalanguages, we also introduce *define*. We will show that the latter is required for languages where *execute* cannot syntactically represent both block-scoped statements and program-scoped definitions. For instance, interface definitions and import directives in Java are only globally defined, while statements can only be locally defined in blocks. Thus, there can be no appropriate common non-terminal that *execute* could adopt to address both. We will further discuss this matter after first detailing the staging tags syntax and semantics.

*Inline* (written *!(expr)*) is staged code evaluating *expr* (whose value must be of an AST type) into the enclosing program by substituting itself. Inline tags within quasi-quotes are allowed, and as all other quasi-quoted expressions, are just AST values that are not directly evaluated. It is allowed for expressions carrying an AST representing an inline directive to be inlined, meaning generation directives may generate further generation directives, thus supporting metagenerators. The latter, besides nested stages statically defined via explicit staging tags, allows any number of stages to be dynamically introduced by metaprograms themselves.

As an extreme example, the following inline directive (the second one) repeatedly reproduces itself (using the first one that is quasi-quoted), causing endless staging.

```
function f() { return <<!(f())>>; }
!(f());
```

With a small change, the same example works in a way that the inline directive reproduces itself a controlled number of times. The *nil* value shown in the example denotes an empty AST value that essentially replaces the generator with no code.

```
function f(n) { return n < 1 ? nil : <<!(f(~(n-1)))>>; }
!(f(10));
```

*Execute* (written *&stmt*) defines a stage *stmt* representing any single statement, local definition, or block in the language. Any definitions introduced are visible only within staged code. Execute tags can also be nested (e.g., *&&stmt*), with their nesting depth specifying the stage nesting of the *stmt* that follows. Essentially, *execute* is similar to Metalua's metalevel shifting construct *–{…}*. For example, the Delta code *&std::print(1)* is equivalent to Metalua's *–{ print "1" }* while *&&std::print(2);* is equivalent to *–{–{ print "2" }}*. Additionally, execute tags can be quasi-quoted and be converted to AST form, meaning their inlining will introduce further staging.



DOI: 10.1002/spe

The following is a simple example (adopted from [25]) combining the use of *inline* and *execute* tags. The function *ExpandPower* creates the AST of its *x* argument being multiplied by itself *n* times, while *MakePower* creates the AST of a specialized power function. It should be noted that in Delta, function definitions are syntactically statements, thus allowed within *execute* tags. As shown, the code resulting from this program encompasses only an assignment of the generated anonymous function.

```
&function ExpandPower (n, x) {
  if (n == 0)
    return <<1>>;
  else
    return <<~x * ~(ExpandPower(n - 1, x))>>;
}
&function MakePower (n) {
  return << ( function(x) { return ~(ExpandPower(n, <<x>>)); } ) >>;
}
power3 = !(MakePower(3));
power3 = (function(x) { return x * x * x * 1; });
```

*Define* (written @*defs*) allows introducing stage *defs*, the later syntactically representing any *global* program unit in the language (e.g., global definitions). Define tags may be nested (e.g., @@*def*) with the nesting depth specifying the stage the *defs* will appear in, being analogous to nested *execute* tags.

This tag is only needed for languages where there is a syntactic distinction between global and local definitions. Thus, in languages such as Lua, JavaScript, or Delta, the latter is not needed because global and local elements are not syntactically separated. There, *execute* can do what *define* is supposed to offer. But *define* is required in languages such as C++, Java, or C# because there are differentiations as to what can be defined locally or globally.

Now, why *define* becomes necessary in such language case and what actual metaprogramming need it serves become clear with an example. Consider the staged code of Figure 8 (top left part), having stage nesting one, in a hypothetical meta-C++ language adopting our staging tags. When no *define* is available, stage code adopts *execute* and *inline* tags, but our example is using just *execute*. The integrated metaprogram is assembled from all *execute* snippets into the *stage_1()* function of Figure 8 (right part). Notice the two function definitions in *execute* tags, namely, *f* and *Load_B* (shaded areas, top left part of Figure 8), whose functionality is required in stage code. The problem is that their concatenation into *stage_1()* constitutes illegal C++ syntax as no local function definitions are allowed (shaded areas, right part of Figure 8).

Offering *define* allows the distinction between global definitions and statements, enabling metalanguages to assemble integrated metaprograms by separating *global stage definitions* from the main execution block. Using *define*s, we rework our example in Figure 9, turning the resulting C++ metaprogram to syntactically correct.

*meta-C++ staged code:*

```
& int f(…) {…}
& int x = f(…);
class C {…};
& static B* Load_B(C* c){…}
& C* c = new C(…);
& B* b = Load_B(c);
& class A {…};
& A* a = new A(…);
```

*main() function of the integrated metaprogram:*

```
int main (int argc, char** argv){
      … stage_1(…); …
}
```

*assembled main stage block:*

```
class C {…};

void stage_1(…) {
   int f(…) {…}
   int x = f(…);

   static B* Load_B(C* c){…}
   C* c = new C(…);
   B* b = Load_B(c);
   class A {…};
   A* a = new A(…);
}
```

Figure 8. Without supporting *define* all stage snippets are by default collected inside the main stage block, which could cause ill-formed C++ syntax as C++ forbids local function definitions (shaded code).



DOI: 10.1002/spe

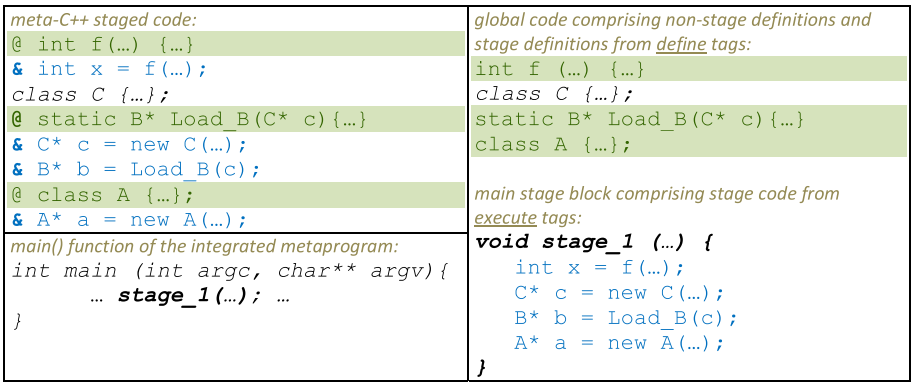



Figure 9. When supporting *define*, only stage code from *execute* directives is collected inside the main stage block; code from *define* is assembled with rest non-stage global definitions, following their order of appearance, resulting in syntactically correct C++ code.

As shown, *define* allows programmers to explicitly instruct the stage assembler to separate specific definitions from the main stage block and place them in the global definitions section of the integrated program.

We may try dropping the extra *define* tag by extending the semantics of *execute* to automatically treat global definitions as implicit *define* directives. Now, this can be problematic in languages such as C++ where elements allowed in a global context, such as classes and type definitions, may also appear locally in a block. If the goal is to locally define a class hidden outside, such as for template instantiations (being a very common practice), the class will be placed in the global section, and encapsulation is broken. Consider the example of Figure 10 where local type definitions are provided. This example fails as staged code when implicit *define* directives are implemented because of name conflicts when locally defined types are transferred in the global section.

*4.2.3. Stage assembly and evaluation.* The current integrated stage program is composed by considering stage code only at the innermost level, thus consisting of non-staged code. It consists of two sections, one after the other: *global area* and *main block*. The main block collects code from *execute* and *inline* directives concatenated together in the order they appear in the source text. The global area encompasses all main program definitions used by the current stage code and also all codes from *define* directives (if applicable), while preserving the relative order of appearance of concatenated fragments in the main source text. An example is provided under Figure 11 illustrating this process.

The stage assembly process begins by computing the maximum stage nesting with a traversal on the entire AST. This computation should be repeated at the beginning of every stage evaluation because the maximum stage nesting may be increased if the evaluation of the last stage has generated further staged code. Then, we need to perform a depth-first traversal and collect source code from all staging tags in this nesting. For *execute* and *define* tags, the associated code is actually all that is needed, and it can be used as it is, while cutting respective nodes from the main program AST.

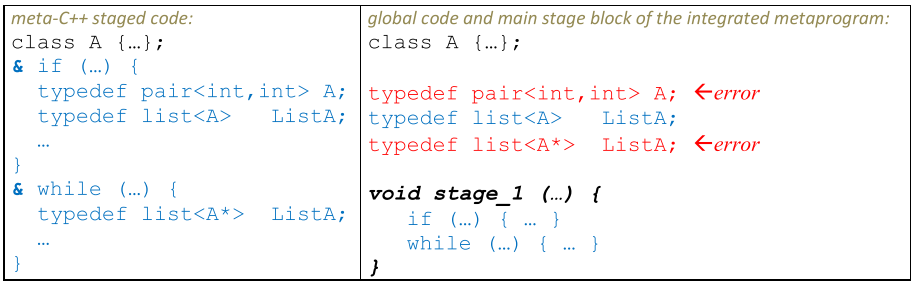



Figure 10. The automatic treatment of *execute* directives involving global definitions as *define* directives disables encapsulation and information hiding for element categories allowed both at global and local scope; this may cause semantic errors, such as replicate definitions (name conflicts).



DOI: 10.1002/spe

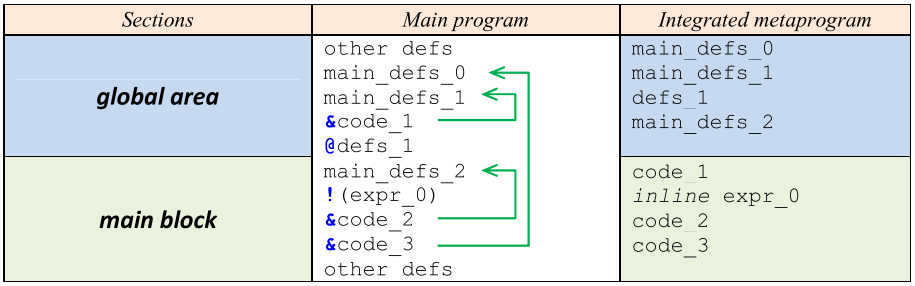



Figure 11. Illustrating the assembly of integrated metaprograms with a general example with arrows outlining dependencies—only the required definitions from the main program are included.

However, *inline* directives require a different treatment, because merely copying the associated *expr* in the main block will not realize its expected generative behavior. The latter, as mentioned earlier, involves substitution of the *inline* node by its *expr* value. Clearly, some special invocations need to be included, which will internally handle the required AST modifications. In this context, an effective approach is to adopt a library function offered by the metacompiler that is linked only with integrated metaprograms. This ensures that integrated metaprograms are syntactically just normal programs and can be compiled using the original language compiler. When running, the metacompiler is just part of their execution environment.

In our implementation, this function is called *std::inline*, with no result and a single argument being the *expr* of AST type. Then, while assembling the stage, the *expr* node is cut, leaving a leaf *inline* alone, and an *std::inline(expr)* invocation is introduced in the main block. The role of the *inline* leaf is to be a bookmark for the insertion point of its respective *std::inline* call. For this reason, an *inline* leaf is pushed on a stack, exactly after its respective *std::inline* call is placed in the main block, thus ensuring their relative orders match. Then, the *std::inline* function simply pops an *inline* leaf from the stack and substitutes it by its *expr* argument.

Definitions from the main program deployed across stages need not be placed in external modules but are directly included in the stage assembly (reflected in the *main_defs_i* parts of Figure 11). In case of deployed functions, they should not access global variables of the main program. In other words, they should not depend on the state of the main program; thus, they may be safely copied and become an integral part of another program. Also, further dependencies of copied definitions should be copied as well, the process being recursive until no required dependencies are missing.

In terms of performance, the integrated staging approach involves a single compilation and evaluation round per stage nesting. In comparison, existing multi-stage languages involve one round per stage code fragment. Effectively, the evaluation of integrated metaprograms is overall more efficient.

An example is provided under Figure 12 with a multi-stage main program: its integrated metaprograms with resulting intermediate and final main versions.

Initially, the maximum stage nesting is 2, tied between the declaration and usages of function *create_macro* (prefixed by the tags *&&* and *&!* respectively). The first stage thus contains the code present within these tags (Figure 12, top left, highlighted with a dotted rectangle), along with the appropriate invocations to *std::inline* (generated by the inline directives) for performing the code generation. Because *create_macro* is defined at stage nesting two, it is removed from the intermediate main, while all *inline* tags at nesting two are substituted by the generated code of their *std::inline* calls.

Thus, the resulting intermediate main contains the stage definitions of functions *identity* and *double* along with the remaining code (i.e., the two assignments) that was part of the original program (Figure 12, middle left, highlighted with a dotted shape). The same process continues for the next stage. Now, the stage nesting is 1, tied between the definitions and usages of functions *identity* and *double*, so the extracted metaprogram contains their code along with the extra *std::inline* invocations. After stage execution, the function definitions of the stage itself are removed, and the *inline* directives are again replaced by the generated code of their *std::inline* calls. This results into the final main having no further stages (Figure 12, bottom left).



DOI: 10.1002/spe

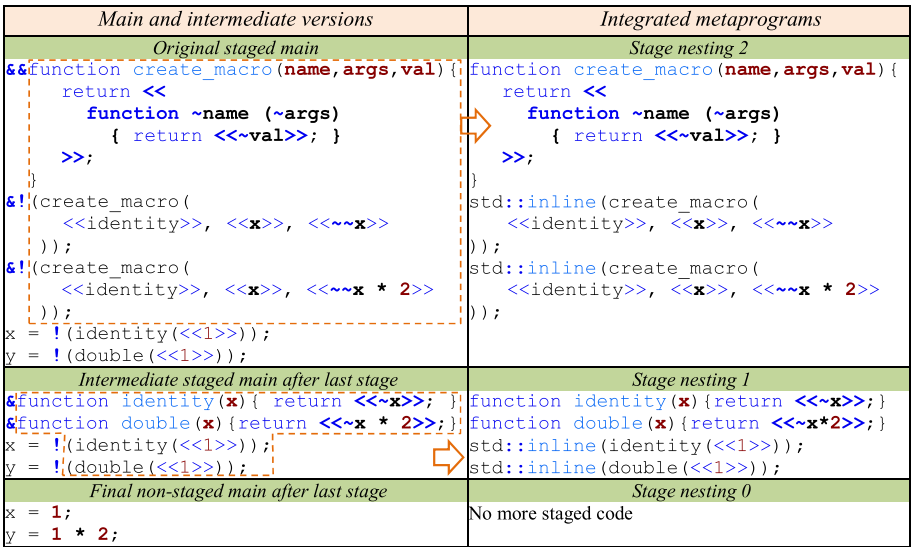



Figure 12. Left: main with its intermediate and final versions; right: integrated metaprograms from the original and intermediate main versions.

From the previous discussion, it should be clear that metacode fragments are assembled and evaluated within the context of the source file they reside in and cannot interact with or manipulate metacode fragments of another source file. The latter is true even in case the metacode of a source file deploys additional modules; any such modules are used in binary form, so any metacode that was part of the original source is no longer available for any interaction to occur between metacode of different sources.

*4.2.4. Metagenerators.* As previously mentioned, to support metagenerators, the language should enable creating ASTs with nodes representing staging tags. Simply inlining such ASTs will introduce staging. The latter can be done by supporting staging tags directly in quasi-quotes or through AST composition using a library. In general, stages may even generate code with more deep staging than themselves. For instance, consider the example shown in Figure 13, where *meta_gen* is a metafunction that is capable of generating code with arbitrary stage nesting, even though it is defined in the first stage with nesting one. In our example, it is invoked twice to generate two *print* calls, at stage nesting one and two respectively. Notice that the loop assignment *code* = *<<&~code;>>;* essentially prepends the syntax tree carried by *code* with an extra *execute* parent tag, thus increases its stage

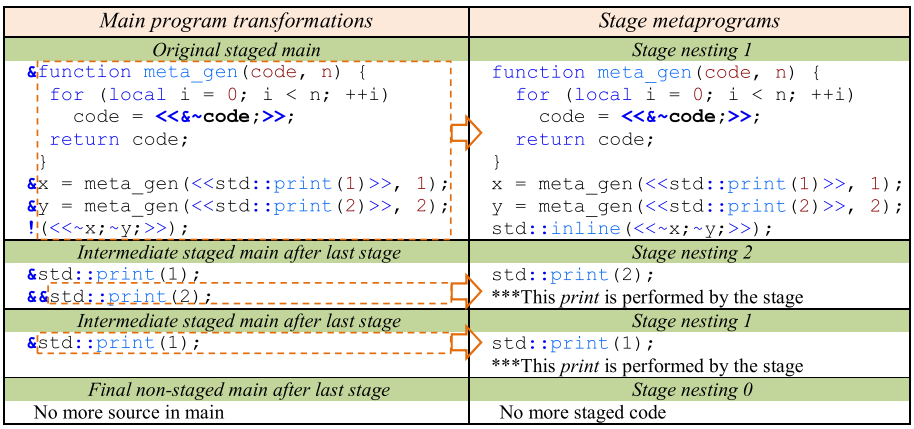



Figure 13. An example where the first evaluated stage is a metagenerator. Left: main with intermediate and final versions; right: integrated metaprograms from original and intermediate main versions.



DOI: 10.1002/spe

nesting in every iteration. As a result, the two invocations return the trees *<<&std::print(1);>>* and *<<&&std::print(2);>>* assigned to stage variables *x* and *y* respectively. These trees are combined with quasi-quotes and are then *inlined*, thus introducing additional staging.

*4.2.5. Context sensitivity* As discussed earlier, context sensitivity is supported when the actual source code insertion point may become the outcome of a computation. Currently, the *inline* directives of multi-stage languages denote a statically defined context, which is the particular location of the directive itself. To support code generation at an arbitrary context, that is, location within the source code, a possible solution is to offer an entry point for obtaining and directly manipulating the source program AST. In this direction and following the approach of offering the functionality as a special library function so as to allow stages to be syntactically normal programs, we propose the provision of another compile-time library function that simply returns the AST node of itself, representing its actual location in the source program. In our implementation, the latter is named *std::context()*. For convenience, we also offer an overloaded version that receives an AST tag and instead returns the closest matching ancestor node. This way, an invocation *std::context(tag)* operates almost as it reads; it returns the AST node that matches the given tag within the invocation context.

We provide an example of context-sensitive generation for our implementation in the Delta language, where objects are created *ex nihilo* via respective constructor expressions, also called object expressions. Consider an object expression in which we wish to insert *set / get* methods for its members. Instead of repeated *inline* directives per data member, we use *std::context* to get the object expression AST, traverse it to find the data members, and generate and insert the desired methods.

```
&function InsertAccessors(obj) {      ← input is a class AST node
  foreach (local attr, obj.getAttributes()) {      ←iterate over class attributes
    local name = attr.getName();
    local set  = <<method ~("set_"+name)(val){ self.~name=val; }>>;
    local get  = <<method ~("get_"+name)(){ return self.~name; }>>;
    obj.addMethods(set, get);   ← insert set / get methods directly in object expression
  }
  return nil;                   ← no code explicitly returned
}
const OBJ_TAG = std::AST_OBJECT_CONSTRUCTOR_TAG;
function Point(x, y) {
  return [
    @x : x, @y : y,
    !(InsertAccessors (
       std::context(OBJ_TAG)
    ));
  ];
}
```

⇨

```
function Point(x, y) {
 return [
  @x : x, @y : y,
  method set_x(val){ self.x = val;  },
  method get_x()   { return self.x; },
  method set_y(val){ self.y = val;  },
  method get_y()   { return self.y; }
 ];
}
```

### 4.3. Expressiveness

We prove that the integrated metaprogramming model is at least as expressive as the current multi-stage programming model. Effectively, using the staging tags, evaluation order, and stage assembly semantics of our language, we emulate the top-down and inside-out evaluation orders of existing multi-stage languages.

In the introduction of our model, we explained the different evaluation orders between current multi-stage languages and integrated metaprograms. We recall the example we used and its traditional evaluation order:

```
!f1(!f2());
!g1(!g2());

!f2 → !f1 → !g2 → !g1
```



DOI: 10.1002/spe

We can emulate the traditional stage evaluation sequence under the integrated metaprogramming model by modifying the example as follows.

```
!f1(!f2());
!<<!g1(!g2())>>;
```

The second staged expression !<<*!g1(!g2())*>> denotes a metaprogram inlining the quoted AST <<*!g1(!g2())*>> and now has stage nesting one. The key point here is that once we quasi-quote a definition, we turn it to a constant non-staged AST expression. We recall our earlier discussion on tags where we mentioned that none of the AST tags are staging tags. Essentially, we *hide* any staging embedded in the quasi-quoted definitions, until *revealed* latter at some point through inlining. The maximum stage nesting in the refined program is two and concerns $f_2$ in *!f1(!f2())*; thus, the first stage consists only of the execution of $f_2$. Then, the resulting program consists of *!f1* and !<<*!g1(!g2())*>>, both with stage nesting one, thus composing the next stage and evaluated together. Up to this point, the evaluation sequence is the following (blocks indicate distinct stage programs; arrows indicate order).

```
{ !f2 } → { !f1 → !<<!g1(!g2())>> }
```

The expression !<<*!g1(!g2())*>> is only used to generate the code whose original staging was hidden with quasi-quotes and thus revealing again its staging:

```
!g1(!g2());
```

In the same way as before, this code will further evaluate as follows, with two different stage programs:

```
{ g2 } → { g1 }
```

As shown, the resulting evaluations follow a sequence that is identical to the traditional order. This method can be generalized for any multi-stage program and may be automated in the compiler, if traditional evaluation is needed, through AST manipulation. More specifically, we can locate all top-level staged fragments and surround them with an increasing number of nested !<< … >> tags, starting with zero. Thus, the first fragment remains as it is (zero tags), the second is put inside ! << … >> (one tag), the third inside ! << *!* << … >> >> (two tags), and the *n*-th inside *n-1* tags. This process is illustrated in Figure 14 and shows that that the integrated model can emulate the traditional model.

### 4.4. Discussion

There are certain trade-offs involved in adopting the integrated metaprogramming model. On the one hand, it offers the notion of a coherent metaprogram with its lexical scoping, shared state, and sequential control flow. On the other hand, supporting these features introduces an explicit dependency across stage fragments that could restrict the potential for evaluating in a different way,

| General form of nested staged code | Emulating the traditional evaluation order in the integrated model |
|---|---|
| $!f_{1_1}(!f_{1_2}(!f_{1_3}(\ldots!f_{1_{n_1}}()\ldots)))$ | $!f_{1_1}(!f_{1_2}(!f_{1_3}(\ldots!f_{1_{n_1}}()\ldots)))$ |
| $!f_{2_1}(!f_{2_2}(!f_{2_3}(\ldots!f_{2_{n_2}}()\ldots)))$ | $!<<\ !f_{2_1}(!f_{2_2}(!f_{2_3}(\ldots!f_{2_{n_2}}()\ldots)))\ >>$ |
| $!f_{3_1}(!f_{3_2}(!f_{3_3}(\ldots!f_{3_{n_3}}()\ldots)))$ | $!<<\ !<<\ !f_{3_1}(!f_{3_2}(!f_{3_3}(\ldots!f_{3_{n_2}}()\ldots)))\ >>\ >>$ |
| … $!f_{i_1}(!f_{i_2}(!f_{i_2}(\ldots!f_{i_{n_i}}()\ldots)))$ … | *i-1* repetitions … *i-1* repetitions $!<<\ \ldots\ !f_{i_1}(!f_{i_2}(!f_{i_2}(\ldots!f_{i_{n_i}}()\ldots)))\ \ldots\ >>$ |
| $!f_{m_1}(!f_{m_2}(!f_{m_3}(\ldots!f_{m_{n_m}}()\ldots)))$ | $!<<\ \ldots\ !<<\ !f_{m_1}(!f_{m_2}(!f_{m_3}(\ldots!f_{m_{n_m}}()\ldots)))\ >>\ \ldots\ >>$ |

Figure 14. Emulation of the traditional top-down and inside-out stage evaluation order in the integrated model; delayed stage evaluation is forced with quasi-quotes and inlining.



DOI: 10.1002/spe

such as arbitrary reordering or parallel evaluation. However, we consider the latter to be a general language issue rather than a metaprogramming model concern. In particular, as with normal programs, reordering or parallelism may be automated by optimizers and runtimes to improve performance, however, without mandating a paradigm shift from the original programming model of languages. Because we emphasize the common treatment of metaprograms and normal programs, we consider all these issues on metaprograms to be uniformly addressed following the practices of normal programs.

Another trade-off relates to the inherent programming complexity in managing and orchestrating separate stage fragments in order to behave meaningfully under their sequential control flow. Because our control flow links code segments together that are not close to each other in file scope, programmers may have to non-locally shift focus of attention to assimilate such behavior. Apparently, this is not an easy task and is something beyond the requirements of handling normal programs. However, its complexity is not the same as splitting an algorithm into disparate sections and keeping track of its behavior. In fact, we do not suggest or foresee that algorithms within stages are to be split for some reason into separate stage fragments. We consider that most dependencies between fragments will concern scope access to earlier state and behavior, with the stage control executing linearly across fragments from top to bottom of the main source file. Clearly, once stage fragments can be independent to each other, they are essentially executed atomically and are far easier to understand and control their behavior. While the latter is implemented with no extra complexity in our model, we believe such scenarios represent only a very small picture on what we expect metaprograms would do in the future.

Once a metaprogram grows at a point where it becomes difficult to manage its source code, runtime state, and control flow, we can deploy refactoring, separate modules, abstraction techniques, or whatsoever, as with any normal program suffering from similar issues. For instance, it is possible to make libraries with code composition functionality (AST manipulation and non-staged) that can be deployed by the various metaprogram stages. However, it is not possible to entirely decouple the functionality of a metaprogram from the original source file it actually affects. The actual source locations where the code generation occurs are determined by the inline tags placed directly within the affected source file. Thus, even if the entire metaprogram logic is placed in a separate module, the original source still needs staged code to load the module and call generator functions inside the appropriately placed inline directives.

Finally, a note related to type checking, as it enables an entire class of metaprogramming bugs to be caught early by the type checker. Because Delta is an untyped object-based language and the metaprogramming extension fully reuses the host language, it involves no type checking. However, the integrated metaprogramming model as such is orthogonal to the presence of a type system. Our proposition targets the composition and execution of stages and does not involve the type system, even if the hosting language would be typed.

## 5. EXAMPLE CASE STUDIES

We discuss metaprogram scenarios utilizing basic object-oriented features such as encapsulation and state sharing. Such features may differ from what is typically met in the discussion of a metalanguage, but they are chosen on purpose to: (i) emphasize our point that metaprograms are more than atomic macro expressions and (ii) highlight the importance of engineering stages like normal programs exploiting shared state and control flow among stage fragments. It should be noted that we do not argue that the computation involved in such examples cannot somehow be expressed in existing multi-stage languages. Instead, our focus is purely on the software engineering advantages of our model enabling typical programming patterns and techniques that are applied in normal programs. In general, most examples involve grouping of common functionality under generative objects that are shared across different stage fragments. Such objects are only setup initially and are freely deployed within different stage fragments to generate code. The latter avoids the tedious repetition of the setup sequence as required with atomic macros that lack state sharing.

### *5.1. Exception handlers*

*Exception handling* [28] is known to be a global design issue that affects multiple system modules, mostly in an application-specific way. In this sense, it should be possible to select a specific



DOI: 10.1002/spe

exception handling policy for the entire system or apply different policies for different components of the system. By using typical object-oriented techniques, the only solution would be to abstract the desired exception handling policy within a function (or object method) and place a corresponding invocation to every applicable catch block. However, it does not avoid the boilerplate code required for declaring the handler and performing the function call nor does it support arbitrary exception handling structures or context-dependent information.

In this context, we can use metafunctions to generate code for exception handling patterns. However, without shared state, metafunction invocations are separated and require explicit and tedious repetition of the pattern details. Moreover, if multiple exception handling patterns are available, it is not possible to parameterize their application or even use binders, to form custom exception handling policies. By using our model, it is possible to maintain a collection of the available exception handling patterns and select the appropriate policy based on configuration parameters or normal control flow while requiring no changes at the call sites inside client code. This is illustrated in the following example.

```
&function Logging (stmts)
  { return << try { ~stmts; } catch e { log(e); } >>; }

&function CreateRetry (data) {      ←constructor for a custom retry policy
  return function (stmts) {         ←return a function implementing the code pattern
    return <<                       ←the returned function returns an AST
      for (local i = 0; i < ~(data.attempts); ++i)
        try { ~stmts; break; }                  ←try & break loop when successful
        catch e { Sleep(~(data.delay)); }       ←catch & wait before retrying
      if (i == ~(data.attempts))                ←maximum attempts were tried?
        { ~(data.failure_stmts); }              ←then give-up & invoke failure code
    >>;
  }
}

&ex = [                             ←compile-time structure for holding exception handling policies
      @policies : [], @active : "",
      method InstallPolicy (key, func) { @policies[key] = func; },
      method SetActivePolicy (policy) { @active = policy; },
      method Apply (code) { return @policies[@active](code); }
];
&ex.InstallPolicy("LOG", Logging);              ←install the logging policy
&ex.InstallPolicy("RETRY",CreateRetry([         ←create and install a retry policy
  @attempts : 5, @delay : 1000, @fail : <<post("FAIL")>>
]));
```

```
&ex.SetActivePolicy("RETRY");
!(ex.Apply(<<f()>>));
```

⇨

```
for (i = 0; i < 5; ++i)
  try     { f(); break;  }
  catch e { Sleep(1000); }
if (i == 5) {post("FAIL");}
```

```
&ex.SetActivePolicy("LOG");
!(ex.Apply(<<g()>>));
!(ex.Apply(<<h()>>));
```

⇨

```
try { g(); } catch e { log(e); }
try { h(); } catch e { log(e); }
```

As shown, we utilize the stage object *ex* to accommodate and compose typical exception handling policies. It is used in an object-oriented fashion to initially install a number of required policies, such as *LOG* and *RETRY* and to generate the respective exception handling code by the invocation of the *Apply()* method. In this example, *Logging* is directly a policy metafunction, while *Retry* accepts parameters to produce the required policy metafunction (e.g., number of retries, delay between attempts, and fallback code when all attempts fail). Such parameters are provided once, upon policy installation, and are not repeated per policy deployment. This relieves programmers from repeatedly supplying all required parameters and constructing all needed objects. Additionally and most importantly, it allows a uniform invocation style, enabling different policies to be



DOI: 10.1002/spe

activated as required at an initial point, without inherent changes at the generation sites involving *!(ex.Apply(…));* directives. An extended version of this example, as well as additional exception handling patterns based on our model, is provided in [29].

### 5.2. Design patterns

*Design patterns* [30] constitute generic reusable solutions to commonly recurring problems. They are not offered as reusable modules but are recipes to apply a solution 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.

We have examined the possibility of utilizing metaprogramming to support generating concrete pattern implementations, where applicable. In this context, the pattern skeleton is turned into composition of ASTs, the pattern instantiation options become composition arguments, the actual client code is supplied as AST arguments, and the pattern instantiation is handled by generative directives. To effectively accommodate such requirements, metaprograms require features beyond staged expressions.

With integrated metaprograms, programmers may apply practices like encapsulation, abstraction, and separation of concerns, thus significantly improving the metaprogram development process. For example, it is possible to implement abstract pattern generators, have multiple of such objects or even hierarchies of them available, and select the appropriate generator for a target context while preserving a uniform invocation style. This functionality is demonstrated in the following example that implements the *adapter* pattern. The pattern is implemented in two ways, by using delegation and sub-classing, while its application may be parameterized with staging.

```
function Window(args) {                          ← runtime class that will be adapted
      return [
            method Draw() {…},
            method SetWholeScreen() {…},
            method Iconify() {…}
      ];
}
&function GetClassDef (target) {…}     ← uses compiler state to find the target class
&function AdapterByDelegation() {    ← creates an adapter object that uses delegation
   return [
      method adapt (spec) {
         local methods = nil;       ← AST of adapted class methods, initially empty
         local class = GetClassDef(spec.original);
         foreach(local m, class.getMethods()) {   ← iterate over class methods
            local name = m.GetName();
            local newName = spec.renames[name];
            if (not newName) newName = name;← if no renaming use original name
            methods = <<       ← merge existing adapted methods with the current one
               ~methods,
               method ~newName (...) { @instance.~name(...); }
            >>;
         }
         return <<   ← create and return the adapted class using the adapted methods AST
            function ~(spec.adapted) (...) {
               return [
                  @instance : ~(spec.original)(...),
                  ~methods
               ];
            }
         >>;
      }
   ];
}
```



DOI: 10.1002/spe

```
&function AdapterBySubclassing() { ← creates an adapter object that uses subclassing
   return [
      method adapt (spec) {
         local adaptedMethods = nil;← AST of methods to be adapted, initially empty
         local class = GetClassDef(spec.original);
         foreach(local m, class.getMethods()) {   ← iterate over class methods
            local name = m.GetName();
            local newName = spec.renames[name];
            if (newName)←only check renamed methods, other are inherited by base class
               adaptedMethods = <<   ← merge adapted methods with the current one
                  ~adaptedMethods,
                  method ~newName (...) { self.~name(...); }
               >>;
         }
         return <<   ← the adapted class as a subclass that introduces the adapted methods
            function ~(spec.adapted)(...) {
               local base = ~(spec.original)(...);       ← base class object
               local derived = [~adaptedMethods];     ← derived class object
               std::inherit(derived, base); ← derived object inherits from base
               return derived;
            }
         >>;
      }
   ];
}
&AdapterFactory = [     ←Creating and populating a factory with adapter implementations
      @adapters : [],
      method Install (type, func) { @adapters[type] = func; },
      method New (type) { return @adapters[type](); }
];
&AdapterFactory.Install("delegation", AdapterByDelegation);
&AdapterFactory.Install("subclassing", AdapterBySubclassing);
&adapterType = "delegation";          ← can also be read or computed dynamically
&adapter = AdapterFactory.New(adapterType);          ←create an adaptor object
&windowAdapterData = [                ← compile-time data for the window adapter
  @original: <<Window>>, @adapted : <<WindowAdapter>>,
  @renames : [{"SetWholeScreen":"Maximize"}, {"Iconify":"Minimize"}]
];
```

```
!(adapter.adapt(
  windowAdapterData
));
```

⇨

```
function WindowAdapter(...) {
 return [
  @instance : Window(...),
  method Draw(...) { @instance.Draw(...); },
  method Maximize(...)
    { @instance.SetWholeScreen (...); },
  method Minimize(...){@instance.Iconify(...);}
  ];
}
```

```
&adapter = AdapterFactory.New("subclassing");     ←create new adapter object
&windowAdapterData.adapted = <<WindowAdapter2>>;     ←change adaptation data
```

```
!(adapter.adapt(
  windowAdapterData
));
```

⇨

```
function WindowAdapter2(...) {
 local base = Window(...);
 local derived = [
   method Maximize(...)
     { self.SetWholeScreen(...); },
   method Minimize(...){ self.Iconify(...); }
 ];
 std::inherit(derived, base);
 return derived;
}
```



DOI: 10.1002/spe

Such generator objects can also abstract implementation details of the classes they produce, with such details specified only upon creation. For instance, consider a *singleton* class that may adopt different invocation styles (e.g., static functions or static instance and methods), which may even be declared within a namespace, thus requiring extra syntax in its usage. Implementing such a code generation scheme in a typical multi-stage language requires repeating the generated class details at every location, something painful (consider that such details are syntactically verbose because of quasi-quotes) and error-prone. Similarly, updating or replacing the implementation would require manually locating all affected sites and applying individually the required changes. We show an example for the definition and usages of a *MemoryManager* singleton class. In this example, the singleton class is modeled either through a prototype function or as global data.

```
&memoryManagerClass = << [          ← basic MemoryManager class implementation
                            method Initialize ()     {...},
                            method Cleanup    ()     {...},
                            method Allocate   (n)    {...},
                            method Deallocate (var)  {...}
                      ] >>;
&function GenerateMemoryManagerAsFunction() {
  return [
    @defs : << function MemoryManager()    ← definition as a prototype function
              { static mgr = ~memoryManagerClass; return mgr; } >>,
    @init : <<MemoryManager().Initialize()>>, ←uses will generate a proper call
    @cleanup : <<MemoryManager().Cleanup()>>, ←to obtain the static object
    method alloc(n) { return << MemoryManager().Allocate(~n) >>; },
    method dealloc(var){return <<MemoryManager().Deallocate(~var)>>;}
  ];
}
&function GenerateMemoryManagerAsGlobalData(){
  return [
    @defs : <<mgr = ~memoryManagerClass>>, ← definition as a global data object
    @init : <<mgr.Initialize()>>,            ←uses generate code that directly
    @cleanup : <<mgr.Cleanup()>>,            ← accesses the  global data object
    method alloc (n) { return <<mgr.Allocate(~n)>>; },
    method dealloc (var) { return <<mgr.Deallocate(~var)>>; }
  ];
}
&memoryManagerImplementations = [
      @func  : GenerateMemoryManagerAsFunction,
      @global: GenerateMemoryManagerAsGlobalData
];
&mm = memoryManagerImplementations["global"](); ←create a generator object
```

```
!(mm.defs);                                  mgr = [
...other normal program definitions...  ⇨          method Initialize ()    {...},
                                                   method Cleanup    ()    {...},
                                                   method Allocate   (n)   {...},
                                                   method Deallocate (var){...}
                                             ];
                                             ...other normal program definitions ...
!(mm.init);                             ⇨    mgr.Initialize();
...other normal program initializations...   ...other normal program initializations ...
x = !(mm.alloc(<<10>>));                     x = mgr.Allocate(10);
!(mm.dealloc(<<x>>));                   ⇨    mgr.Deallocate(x);
...other normal program code...              ...other normal program code...
!(mm.cleanup);                          ⇨    mgr.Cleanup();
...other normal program cleanups...          ...other normal program cleanups...
```



DOI: 10.1002/spe

As shown, the invocation details are specified only once for each case and are abstracted through the *mm* code generator object, allowing the definition and deployment code to be automatically produced without requiring any extra information. The latter allows updating the generation parameters, possibly affecting names or calling styles, without having to change all client uses of the generated class. Also, in this example, the ordering of the inline directives is important. In particular, the definitions regarding the memory manager object should be generated before its actual deployment; thus, !*(mm.defs)* is put first. In the same sense, the initialization statements of a memory manager should be generated before any memory allocation calls; thus, !*(mm.init)* is put next.

## 6. INTEGRATED TOOL CHAIN

Treating stages as full-scale programs means supporting them with tools and facilities similar to normal programs. In this context, we elaborate on the integration of stages in the IDE, in particular, the workspace manager, the build system, and the source-level debugger. We also discuss the necessary interactions between the compiler and the various IDE subsystems. All such features were implemented in the Sparrow IDE [26] of the Delta language, thus providing an integrated tool chain for metaprogramming.

### *6.1. Integration in the workspace manager*

First, the integrated metaprograms are extracted and are stored as separate read-only files attached to their respective main program. Clearly, this allows for better reviewing and understanding, compared with studying a metaprogram across separate text fragments as embedded in the main program. Second, the modified versions of the main program, after every stage evaluation, are also stored as separate read-only files showing clearly the staged transformations introduced over the original program. All such source files are produced during the compilation process and are propagated to the workspace manager for inclusion in its respective folders and structures (for more details, see [27]). Once incorporated in the workspace, besides reviewing and browsing, they are also used for reporting compile errors, setting breakpoints, and performing source-level debugging.

### *6.2. Integration in the build system*

With metaprograms already hosted in the workspace, it is possible to support custom compilation options for stages by extending the original compilation options of normal sources to accommodate information about stages. We support specifying different options per stage or have default options applicable for all stages. The default options are actually very useful when the number of stages cannot be statically determined, thus disabling the association of compilation options with specific stage numbers. As mentioned earlier, once a stage source is assembled during compilation, it is directly inserted in the workspace and is associated with its respective main source. From the latter, it also receives its actual compilation options by matching its stage number. Then, it is actually built using the standard build system. This means that all dependencies defined within stages are resolved as expected, and if necessary built, before actual stage compilation.

Because the extraction and the translation of stages occur within the compilation loop of the main program, to build stages, the compiler should directly interoperate with the build system. This process, outlined under Figure 15, involves the following steps:

1. When a source is to be compiled, the build system resolves and builds dependencies, finally invoking the compiler on the actual source, while also waiting possible further requests from it.
2. If the source contains stages, the process continues from step 4.
3. Otherwise, the source is directly compiled, the build system is notified that the binary is ready, and the compilation process ends here.
4. The stage source, that is, an integrated metaprogram, is composed.



DOI: 10.1002/spe

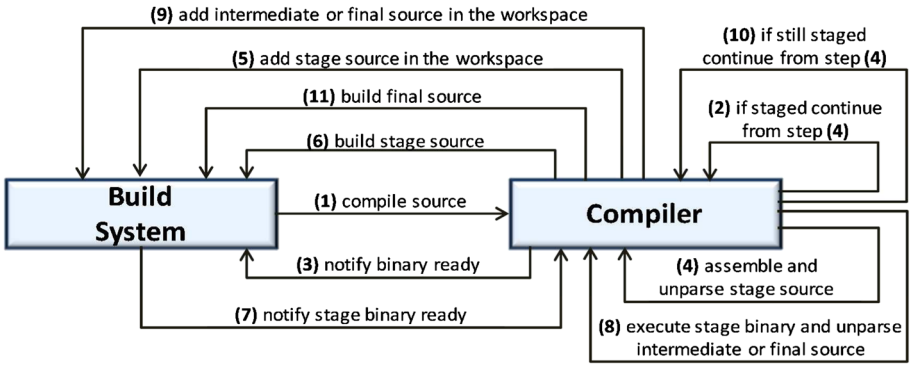



Figure 15. Build system and compiler interaction sequence diagram regarding metaprograms.

5. It is submitted to the build system for propagation to the workspace manager.
6. The build system is asked to build the submitted stage source.
7. Once ready, the build system responds that the stage binary is ready.
8. The metaprogram is run, modifying the current AST, and the intermediate main source is produced by unparsing the AST.
9. It is submitted to the build system for propagation to the workspace manager.
10. If it is still staged, then we continue from step 4.
11. Otherwise, it is the final source, and the build system is asked to build it.

Effectively, the compiler becomes a client of the build system, capable of recursively involving the build system back in the loop with additional build requests for stages. When no changes are met until the last build, the respective binaries should be cached, and their source files should be treated as up-to-date. The latter must apply to stage sources as well. For this purpose, the build process has been extended to become stage aware, as illustrated in the flowchart of Figure 16.

It is worth noting that the first compiler invocation on a source file with stages performs no actual translation. Instead, its purpose is to produce the stage sources, supply them to the build system, and when done, execute their binary to apply their transformations on the main program. Once done with stages, the compiler will eventually submit the final source of the transformed main to the build system.

### 6.3. *Integration in the debugger*

To allow source-level debugging of integrated metaprograms, we discuss a series of minimal interventions on the IDE infrastructure. Generally, a debugging infrastructure is split in the backend, attached to the executing program (i.e., the debuggee) and providing an appropriate query protocol, and the frontend, offering user interaction and internally communicating with the

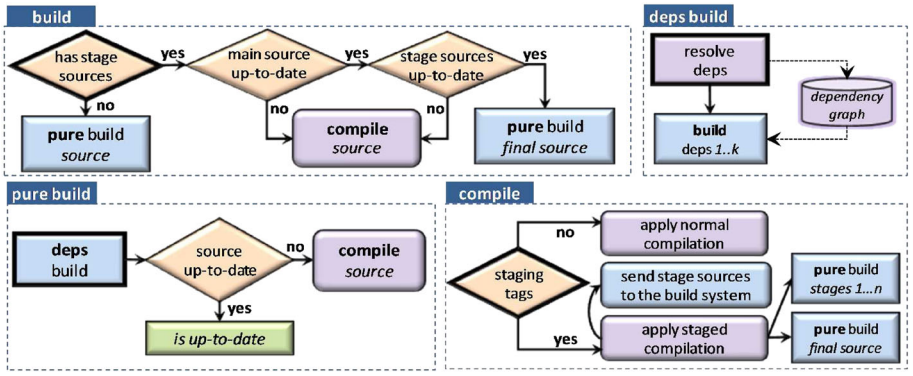



Figure 16. Control flow for the staging-aware build process; starting process is 'build' (top-left).



DOI: 10.1002/spe

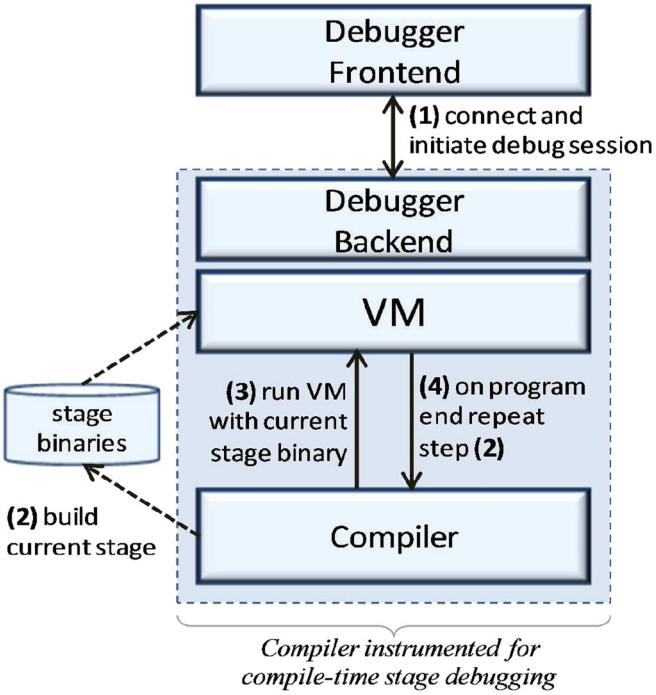



Figure 17. Architecture to support compile-time stage debugging.

backend. The backend is typically incorporated into the language runtime, and the frontend is usually part of the IDE. In our case, the debuggee is the compiler executable. Because the compiler is responsible to run stages, besides the language runtime, it must be linked with the backend as well.

In the context of CTMP, as is the case with our language, tracing stages requires initiating a compile-time debug session through some special user option. Now, consider that stage debugging is enabled in building a source file with stages. This will launch the compiler, which in effect activates the debugger backend and connects to the debugger frontend (Figure 17). Then, the debugging session operates as with any other program. From the point of view of the debugger backend, the only relevant step is the one executing the current stage binary (Figure 17, step 3), which is performed by the virtual machine. Internally, the stage build process and the transition to the next stage when the current terminates are totally transparent to the backend.

The latter actually allows for the entire staging process to be debugged in a seamless way; stepping from the last instruction of one stage will cause the execution to pause at the first instruction of the next stage requiring no separate debug session. In stage debugging, there are also a couple of issues that require extra support such as breakpoint management on stage sources and inspection of runtime values containing ASTs. Our implementation fully addresses these issues as discussed in detail in [27].

## 7. RELATED WORK

We review representative and popular multi-stage languages regarding the identified integrated metaprogramming requirements and compare them with our approach. While we primarily focus on compile-time staging, we also discuss runtime staging within either compiled or interpreted implementations. The comparative summary is provided under Figure 18 and shows that the requirements are only partially met by existing multi-stage languages.



DOI: 10.1002/spe

| | | *Compiled* | | | | | | | *Interpreted* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Integrated Metaprogramming Requirements** | **Sub-properties** | **C++ Templates** | **Template Haskell** | **Nemerle** | **Converge** | **Metalua** | **Groovy** | **Delta** | **Common Lisp** | **Scheme** | **MetaML** |
| **Exploiting Normal Language Features & Tools (§3.1)** | Language | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Compiler | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Execution System | × | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Debugger | × | × | ↓ | × | × | ✓ | ✓ | ↓ | ↓ | × |
| **Context-Free & Context-Sensitive Generation (§3.2)** | Context extraction and editing | × | × | ✓ | × | × | ✓ | ✓ | × | × | × |
| | Inlining | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ |
| **Composing & Generating All Language Constructs (§3.3)** | Generation of staged code | × | × | × | × | × | × | ✓ | ✓ | ✓ | ✓ |
| | Quasi quotes | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | AST manipulation | × | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| | Hygienic names | × | ✓ | ✓ | ✓ | ↓ | × | ✓ | ↓ | ✓ | ✓ |
| | Capturing names | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| **Sharing & Separation of Concerns among Stages & Main (§3.4)** | Functionality and state for stages only | ✓ | × | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ | × |
| | Functionality shared across stages and main program | × | ✓ | ↓ | ✓ | ↓ | ↓ | ✓ | ✓ | ✓ | ✓ |
| **Programming Model for Stages Equal to Normal Programs (§3.5)** | Common state across distinct meta-code fragments | × | × | × | × | ↓ | × | ✓ | ✓ | ✓ | × |
| | Typical control flow across distinct fragments | × | × | × | × | × | × | ✓ | ✓ | ✓ | × |
| **Stages as First-Class Citizens of the Programming Environment (§3.6)** | Source browsing for stages | ✓ | ↓ | ✓ | × | × | ↓ | ✓ | × | × | × |
| | Source editing for stages | ✓ | ↓ | × | × | × | ↓ | × | × | × | × |
| | Build system for stages | × | × | ✓ | × | × | ✓ | ✓ | × | × | × |

Figure 18. Comparison of languages regarding the requirements for integrated metaprogramming. The symbol ↓ means that the feature is offered by the language with certain limitations (more details in the language-specific discussion sections).

### *7.1. Languages with compile-time metaprogramming*

*7.1.1. C++.* C++ support for metaprogramming is based on its template system that is essentially a functional language interpreted at compile time [18]. While C++ templates are Turing Complete [31], they neither offer the main C++ language features nor share its runtime and debugger. Some low-level techniques such as such as those in [32] have been proposed to help in the debugging of template metaprograms. Essentially, C++ can be seen as a two-stage language where the first stage consists of the interpretation of the templates (denoted by the $< >$ tags used in both declarations and instantiations) and the second stage the compilation of the non-template code. Thus, there is no notion of code expressed in AST form allowing for code traversal or manipulation, and there is no way for a template to compose another template. Additionally, templates cannot store or share any state and the programmer has no control on the flow of their evaluation.

Finally, regarding IDE support, programming environments such as Microsoft Visual Studio provide browsing and editing features for code resulting from template instantiations. For example, in viewing classes, functions, and type definitions, everything coming from template instantiations is shown, while linked to its actual template arguments. Moreover, there is auto-completion on functions, classes, and members instantiated from templates, with quick information via tooltips indicating the template parameters. However, C++ templates do not allow freely generating source code but allow only instantiating skeleton classes and functions



DOI: 10.1002/spe

by filling the gaps using the supplied type arguments. Thus, providing source browsing and intelligent editing features is easier compared with general stage evaluation.

*7.1.2. Template Haskell.* Template Haskell [6] is a two-stage language that provides metaprogramming facilities through quasi-quotes and splicing. It reuses most aspects of the normal language without however providing debugging support for stages. It supports custom AST manipulation and allows generating names with either hygienic or capturing style. It also reifies the compiler's symbol table, thus enabling programmers querying the current state of compilation. Additionally, it allows querying the context through *reifyLocn* thus enabling context-dependent generation. However, it does not support splices that generate additional splices, because the splice itself is not an *Expr*.

All declared functions are available for both runtime and compile-time computations, but it is not possible to define functions used only during compile-time stage evaluation. Additionally, no function defined in a module can be used by splices in the same module; for functions to be used in a splice, they must be placed within an imported module. Moreover, splices are evaluated separately, meaning there is no notion of state sharing or lexical control-flow sequence linking them. Finally, *Leksah* [33], a Haskell IDE offering some support for Template Haskell, does provide some browsing and editing support for stages. In particular, any declarations introduced by top-level splices are visible in the source browser and are utilized by the auto-completion system during editing. However, this information is not updated during editing but requires the source file to be compiled first, meaning the symbolic information is supplied to the editor as a product of the compilation process.

*7.1.3. Nemerle.* *Nemerle* [20] is a statically typed class-based language compiled to Common Intermediate Language (.NET binary) supporting metaprogramming through a macro system. Nemerle macros are implemented as compiler plugins and have to be implemented separately, built as typical dynamically imported libraries. They are loaded on demand during compilation of any source filer that invokes them. Macros can be either invoked like functions to generate code during compile-time, or they can be linked (called meta-attribute definitions) to a variety of constructs such as classes and methods to enable context-sensitive source code generation.

Because Nemerle macros are dynamically linked libraries, it is possible to debug them using the original .NET common language runtime debugger. However, the debugging process is rather awkward because it requires setting the programming environment to run the Nemerle compiler in debug mode, as opposed to the user program. Then, tracing macro code is possible during the compile session. Additionally, Nemerle macros, like any other dynamic libraries, can share functionality and exchange state. However, such state sharing requires macro library dependencies, thus restricting modular scenarios where the shared state needs to be propagated across independently defined macros. Also, there is no explicit notion of a lexical control flow among macro invocations of the same source file, because no other stage definitions besides direct macro invocations are supported. Finally, macros cannot generate macros; thus, attempting to declare the following simple macro yields a compile error.

```
macro Generator() { <[decl: macro Identity(x){x}]> } // Compile error
```

Because in Nemerle, macros are not staged, but are separately edited and built as normal modules, the standard source browsing facilities and build system are directly applicable (NemerleStudio or Microsoft Visual Studio with Nemerle plugin). Regarding staging, editing tooltips are supported for macros with point of definition, possible associated keywords, and its actual evaluation result. However, there is no parameter help for macro invocations or syntax hints in case of syntactic extension macros.

*7.1.4 Converge.* *Converge* [7, 8] is a dynamic class-based language that allows CTMP in the spirit of Template Haskell. It reuses the original language features, compiler, and execution system for metaprograms, while it currently offers no source-level debugger to judge if it is reusable on stages as well. However, because the language offers an underlying infrastructure to introduce debugging frontends, it could enable the implementation of a reusable debugger frontend. ASTs can be created and manipulated either via library functions or through



DOI: 10.1002/spe

quasi-quotes. Additionally, generated ASTs may encompass either *alpha-renamed* (i.e., hygienic) variables or dynamically scoped names to support variable capture. Code generation is allowed through splicing at various program locations. However, there is no support for context-sensitive insertions, neither for generation of splices thus disabling metagenerators. By design, Converge makes no scope-related distinction between normal functions and metafunctions, so all user-defined functions are eligible to be included within both stages and the final program (in fact, only the minimal subset of functions are included in each stage). However, it is not possible to explicitly state that a function is only visible for compile-time computations. Finally, concerning stage evaluation, splices are treated as separate temporary modules, being independent of other splices, thus sharing no state or common control flow. For instance, assume the following hypothetical example that is not possible in Converge. The first splice declares a stage variable *x*, and the second tries to access it. The special *pragma* splice *$p<...>* of Converge is here used to ignore the result of the splice expression.

```
import Sys
$p<x := 1>
$p<Sys::println(x)> // Compile error: Unknown variable 'x'.
```

The first splice alone would compile normally, with its evaluation declaring and initializing the stage variable *x*. Actually, if we printed its value within the first splice, the result would be 1. However, once the second splice is put, a compilation error is caused, indicating that no variable *x* exists. As mentioned, this is due to the fact that splices are separate modules, meaning that the second splice will not find the declared variable *x* introduced by the first one.

*7.1.5 MetaLua.* Metalua [9] supports stages with the concept of separated metalevels, allowing shifting between them using special syntax. The original language constructs are fully available in stages. Metacode directly embedded in the main program is referred to as level *zero*, while nested metacode takes the level of its enclosing metacode −1. Thus, levels are numbered as 0, −1, −2, and so on, with innermost levels attributing to the smallest level number. Evaluation of nested metacode is performed inside out, with inner levels always preceding the outer ones. Because metacode is evaluated top-down for level zero and inside-out for negative levels (nested), the execution of stages is interleaved, thus supporting no notion of a lexically scoped sequential control flow.

Sharing of functionality and state across stages is supported. This is due to a custom metacode interpreter that performs the evaluation of all stages in Metalua supporting dynamic scoping. However, such state sharing concerns all metalevels. As a result, different metalevels, although strictly separated and encapsulated, may access and affect the state of other metalevels. The latter breaks encapsulation and seems more of an implementation artifact, inherent in the stage interpreter, rather than design intent. For instance, in the following example, all references to *x*, whether of stage 1 or stage 2, bind to a single stage variable, resulting in the following output: *nil, 2, 3, 1*.

```
x = 1
-{ block:
     print(x) -- uninitialized stage variable x so prints nil
     x = 2  }
-{ block:
     -{ block:
          print(x) -- binds to the earlier stage x so prints 2
          x = 3  }
     print(x) -- x retains the value of 3 above thus prints 3
}
print(x) -- binds to main program x so prints 1 (at runtime)
```

As mentioned, while Metalua exploits all main language features of Lua, it uses a custom interpreter implementation for stages, and it does not support metacode debugging. Finally, while it offers visitors and manipulators for ASTs, it forbids introduction of metalevels programmatically,



DOI: 10.1002/spe

thus disabling metagenerators. Finally, generated variables are dynamically scoped, meaning they are subject to variable capture, while a hygiene library is offered utilizing *gensym* for hygienic macros.

*7.1.6 Groovy.* Groovy [34] supports CTMP through AST transformations. The latter are defined as normal classes implementing the *ASTTransformation* interface and can be applied on other classes through annotations. The class annotations specify the transformations that will be invoked by the compiler while the *ASTTransformation* interface essentially provides an entry point to obtaining and manipulating the AST of the target class. Global transformations are also supported by supplying an *ASTTransformation* subclass to the compiler that will apply it on the entire syntax tree of the code being translated, but their application is separated from the language and involves additional compilation parameters.

Groovy metaprograms reuse the language compiler and runtime system, while it is possible to debug local transformations directly from the IDE (e.g., in IDEA [35]). Additionally, because the transformations are normal Groovy code, the source browsing and editing facilities of the language can be directly deployed. However, there is no such support for the transformation outcomes: symbols resulting from transformations are invisible to the source browser and the auto-completion tool. Finally, all build flags and rules apply on transformation classes as well, thus allowing dependencies on other normal sources or even further transformations.

In Groovy, transformations are evaluated independently to each other and cannot share state or be orchestrated to a lexically scoped control flow. They can share functionality with normal sources only if organized and imported as separate modules. Finally, as with multi-stage languages studied, metagenerators are not supported.

### *7.2. Languages with runtime metaprogramming.*

*7.2.1 Lisp and Scheme.* The two major Lisp dialects, Common Lisp [13] and Scheme [19], support metaprogramming through their powerful macro systems. In Common Lisp, programs can manipulate source code as a data structure; thus, macros may perform any data operation on code as well, offering the entire set of language constructs to express the transformation logic. Simple Scheme macros (created with *syntax-rules*) are essentially transformation procedures accompanied by a simple pattern-matching sublanguage, while R6RS macros (created with *syntax-case* [36]) are procedural syntax transformers that allow destructuring input syntax objects and rebuilding syntax objects as output. At the implementation level, in both languages, normal code and macros share the same interpreter. However, while normal code is traced through the standard language debugger, macros require a dedicated macro stepper to trace code transformations resulting from macro invocations.

Lisp can create ASTs either through normal list processing functions or via the quasi-quoting mechanism, providing full support for traversal and manipulation. Code generation relies on replacing macro invocations with their output expressions, that is, *inlining*, but without offering any context information. The output of macros can be extra macro definitions or invocations, meaning Lisp supports metagenerators. Regarding the generated names, Common Lisp offers name capture, while enabling hygienic macros using *gensym* calls. On the other hand, Scheme offers hygienic macros, while supporting capturing names through the syntax-case macros. By default, Lisp dialects do not offer the notion of a coherent program collecting all macro definitions and invocations by stage nesting, as typical top-down and inside-out evaluations are applied. However, given its runtime system and its powerful reflectivity with self-interpretation, such functionality can be implemented as a custom library for macro management and evaluation. The latter is feasible in Lisp dialects, but it is rather complicated even for advanced users, making it more practical when offered as a built-in feature.

The reason for including Lisp dialects in the RTMP section is practical and relates to the way they are actually implemented. Most implementations are interpreted, that is, executing instructions on an AST, meaning that macros are expanded along with the interpretation of normal code. As CTMP, we consider languages directly compiled to byte code, intermediate code, or machine code.



DOI: 10.1002/spe

In particular, for the former two cases, we assume a comprehensive low-level instruction set with mostly general-purpose instructions. For instance, the *InterLisp* [37] virtual machine specification defines the vast majority of instructions to be Lisp dependent and to directly reflect language constructs. As a result, from an implementation perspective, a Lisp interpreter and a virtual machine are practically identical. Clearly, these are not options to judge, but we only mention to explain why we put Lisp dialects under the RTMP family.

*7.2.2 MetaML, MetaOCaml, and Mint.* MetaML [4] is a multi-stage language that supports RTMP on the basis of staging annotations. It is based on ML, fully exploiting the original language, compiler, and runtime system, however, not providing debugging support for stages. MetaML allows creating ASTs only through a quasi-quote mechanism and while it does not support explicit AST iteration, it offers a pattern-matching search facility on ASTs. Also, while it offers hygienic names, it does not allow selective name capture. Code generation is achieved only through *inlining* (*Run* operator) with no support for context-sensitive generation. In MetaML, all functions are shared across stages and main with no facility for hiding and encapsulation. Finally, stages do not actually share common state, other than possible cross-stage persistent [3] values, while they are evaluated with the typical inside-out and top-down orders, meaning stage code of different nesting levels can be also interleaved in this language.

MetaOCaml [5] is a metaprogramming extension of OCaml and is essentially a compiled dialect of MetaML. *Mint* [38] extends Java with the three standard multi-stage constructs, namely, *brackets*, *escape* and *run*, constituting an application of these concepts in an imperative language with a compiled implementation. As such, they both share the same properties as MetaML with respect to the identified requirements.

## 8. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an integrated implementation framework for CTMP, involving various disciplines ranging from programming model, evaluation semantics, and tool support. We are motivated by the need for a methodological integration between metaprogramming and normal programming, as we consider irrational to have diverse development styles and approaches amongst the two universes.

Because metaprograms are essentially programs, we identified a set of prominent requirements for an integrated framework in which metaprograms may directly adopt the engineering practices, processes, and tools of normal programs. The latter emphasizes an integrated code of practice where metaprograms are not considered to belong to a separate and customized language domain.

Central to our proposition is the notion of integrated compile-time metaprograms, such as coherent programs, collecting the code fragments of the same stage nesting, with their order of appearance in the main source, implying a lexically scoped control flow. Our model is at least as expressive as the traditional stage evaluation in existing stage languages, while providing significant improvements in terms of software engineering. The latter is proved through the example case studies that included demanding scenarios for exception handling and design patterns.

Finally, we discussed an approach for integrating metaprograms in the tool chain, turning metaprograms to first-class citizens of the programming environments, including the workspace manager, the build system, and the debugger. One of the topics that we did not yet address concerns source editing support, such as auto-completion, parameter help, and go-to definition, that we plan to explore in the future.

Another direction for future research regarding the metaprogramming tool chain relates to advanced editing views for integrated metaprograms. Currently, in our implementation, integrated metaprograms are automatically inserted in the IDE to aid program understanding. However, code editing is only possible in the context of the original enclosing source file. Clearly, the latter involves a significant amount of extra source code, most of which may be irrelevant to the current stage itself. In this direction, we consider improved editing features, such as temporarily folding or hiding code not belonging to the focused stage, thus enabling programmers to experience a visually continuous stage program.



DOI: 10.1002/spe

Overall, metaprogramming is a demanding topic that may involve unconventional thinking and require advanced programming skills. As such, offering less support on metaprograms compared with normal programs is essentially a mismatch. Our work emphasized the integration between the two universes so that more challenging, advanced, and comprehensive metaprograms can appear.

## REFERENCES


1. Sheard T, Benaissa Z, Martel M. *Introduction to Multistage Programming Using MetaML*, 2nd edn. Pacific Software Research Center, Oregon Graduate Institute, 2000. Available at: http://web.cecs.pdx.edu/~sheard/papers/manual.ps. [last accessed 18 October 2013].
2. Taha W. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation, Germany*, March 2003, Lengauer C, Batory D, Consel C, Odersky M (eds). Springer, LNCS 3016, 2004; 30–50. Available at: http://dx.doi.org/10.1007/978-3-540-25935-0_3.
3. Taha, W, Sheard T. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM '97)*, ACM: New York, NY, USA, 1997; 203–217, December 1997, Available at: http://doi.acm.org/10.1145/258994.259019.
4. Sheard T. Using MetaML: *a staged programming language*. In: Advanced Functional Programming. Springer LNCS 1608, 1998; 207–239. Available at: http://dx.doi.org/10.1007/10704973_5.
5. Calcagno C, Taha W, Huang L, Leroy X. A bytecode-compiled, type-safe, multi-stage language, 2001. Available at: http://www.cs.rice.edu/~taha/publications/preprints/pldi02-pre.pdf. [last accessed 18 October 2013]
6. Sheard T, Jones SP. Template metaprogramming for Haskell. SIGPLAN Not. 37, 12, 2002; 60–75, Available at: http://dx.doi.org/10.1145/636517.636528.
7. Tratt L. Compile-time meta-programming in a dynamically typed OO language. In *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*, Roel Wuyts (Ed.). ACM: New York, USA, 2005; 49–63. Available at: http://doi.acm.org/10.1145/1146841.1146846.
8. Tratt L. Domain specific language implementation via compile-time metaprogramming. *ACM Transactions on Programming Languages and Systems TOPLAS* 2008; **30**(6):1–40. Available at: http://doi.acm.org/10.1145/1391956.1391958.
9. Fleutot F. 2007. Metalua manual. Available at: http://metalua.luaforge.net/metalua-manual.html. [last accessed 18 October 2013].
10. Weise D, Crew R. Programmable syntax macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*, Robert Cartwright (Ed.). ACM: New York, NY, USA, 1993; 156–165, June 1993, Available at: http://doi.acm.org/10.1145/155090.155105.
11. Bawden A. Quasiquotation in Lisp. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio. University of Aarhus, Computer Science Department. Invited talk, 1999; 88–99. Available at: http://repository.readscheme.org/ftp/papers/pepm99/bawden.pdf. [last accessed 18 October 2013]
12. Sheard T. Accomplishments and research challenges in metaprogramming. In *Proceedings of the Second International Workshop on Semantics, Application and Implementation of Program Generation (SAIG'01)*, Florence, Italy, Springer LNCS 2196, 2001; 2–44. Available at: http://dx.doi.org/10.1007/3-540-44806-3_2.
13. Seibel P. *Practical Common Lisp*. Apress, 2005; ISBN 978–1590592397.
14. Savidis A. Dynamic imperative languages for runtime extensible semantics and polymorphic meta-programming. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2005)*, Heraklion, Crete, Greece, Springer LNCS 3943, 2005; 113–128. Available at: http://dx.doi.org/10.1007/11751113_9.
15. Savidis A. The Delta programming language. 2010. Available at: http://www.ics.forth.gr/hci/files/plang/Delta/Delta.html. [last accessed 18 October 2013].
16. Stroustrup B. *The C++ Programming Language Special Edition*. Addison-Wesley, 2000.
17. Abrahams D, Gurtovoy A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.
18. Veldhuizen TL. Using C++ template metaprograms. *C++ Report* 1995; **7**(4): 36–43.
19. Dybvig RK. *The Scheme Programming Language (fourth edition)*. The MIT Press, 2009; ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93.
20. Skalski K, Moskal M, Olszta P. 2004. Meta-programming in Nemerle. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.8265&rep=rep1&t%20ype=pdf. [last accessed 18 October 2013].
21. Palmer Z, Smith SF. Backstage Java: making a difference in metaprogramming. In *Proceedings of the 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*, ACM: New York, USA, 2011; 939–958, Available at: http://doi.acm.org/10.1145/2048066.2048137.
22. Kernighan BW, Ritchie DM. *The C Programming Language*, second edn. Prentice-Hall: Englewood Cliffs, NJ 07632, USA, 1988.
23. Kohlbecker E, Friedman DP, Felleisen M, Duba B. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming (LFP '86)*, ACM: New York, NY, USA, 1986; 151–161. Available at: http://doi.acm.org/10.1145/319838.319859.




DOI: 10.1002/spe

24. Microsoft Corporation. Using IntelliSense. Available at: http://msdn.microsoft.com/en-us/library/hcw1s69b.aspx. [last accessed 18 October 2013].
25. Czarnecki K, John O'Donnell JS, Taha W. DSL implementation in MetaOCaml, Template Haskell, and C++. In Domain-Specific Program Generation, Springer LNCS 3016, 2004; 51–72, Available at: http://dx.doi.org/10.1007/978-3-540-25935-0_4.
26. Savidis A, Bourdenas T, Georgalis J. An adaptable circular meta-IDE for a dynamic programming language. In *Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007)*, Luxemburg, 2007; 99–114. Available at: http://www.ics.forth.gr/hci/files/plang/sparrow.pdf. [last accessed 18 October 2013].
27. Lilis Y, Savidis A. Supporting compile-time debugging and precise error reporting in meta-programs. In the *50th International Conference on Objects, Models, Components, Patterns*, Prague, Czech Republic, Springer LNCS 7304, 2012; 155–170, Available at: http://dx.doi.org/10.1007/978-3-642-30561-0_12.
28. Goodenough JB. Exception handling: issues and a proposed notation. *Communications of the ACM* 1975; **18**(12): 683–696. Available at: http://doi.acm.org/10.1145/361227.361230.
29. Lilis Y, Savidis A. Implementing reusable exception handling patterns with compile-time metaprogramming. In *Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems*, Pisa, Italy, Springer LNCS 7527, 2012; 1–15. Available at: http://dx.doi.org/10.1007/978-3-642-33176-3_1.
30. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Professional, 1994. November 1994; ISBN 978–0201633610.
31. Veldhuizen TL. C++ templates are Turing Complete. Technical Report, Indiana University Computer Science, 2003. Available at: http://ubietylab.net/ubigraph/content/Papers/pdf/CppTuring.pdf. [last accessed 18 October 2013]
32. Porkolab Z, Mihalicza J, Sipos A. Debugging C++ template metaprograms. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, ACM: New York, NY, USA, 2006; 255–264, 2006, Available at: http://dx.doi.org/10.1145/1173706.1173746.
33. Nicklisch-Franken J, Mackenzie H, Frank A, Gruber C. Leksah: an integrated development environment for Haskell, 2010. Available at: http://leksah.org/leksah_manual.pdf. [last accessed 18 October 2013].
34. Subramaniam V. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf, 2013; ISBN 978-1-93778-530-7.
35. JetBrains. IntelliJIDEA – *Groovy and Grails*. Available at: http://www.jetbrains.com/ idea/features/groovy_grails.html. [last accessed 18 October 2013].
36. Dybvig RK. Writing hygienic macros in Scheme with syntax-case. Technical Report, Indiana University Computer Science Department, 1992. Available at: http://www.cs.indiana.edu/~dyb/pubs/tr356.pdf. [last accessed 18 October 2013].
37. Moore JS. The InterLisp virtual machine specification. Technical Report CSL 76–5, Xerox Palo Alto Research Center, 1976. Available at: http://www.cs.utexas.edu/~moore/publications/interlisp-vm.pdf. [last accessed 18 October 2013].
38. Westbrook E, Ricken M, Inoue J, Yao Y, Abdelatif T, Taha W. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI '10)*, ACM: New York, NY, USA, 2010; 400–411. Available at: http://doi.acm.org/10.1145/1806596.1806642.




DOI: 10.1002/spe