META JAVASCRIPT

Yannis Apostolidis

Heraklion, November 2015

The work reported in this thesis has been conducted at the Human Computer Interaction (HCI) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology - Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship.

META JAVASCRIPT

Yannis Apostolidis
Master's Thesis

University of Crete
Computer Science Department

Abstract

Generative metaprogramming is the ability to treat source code as first-class data and emit it to the program itself, thus modifying it. In multi-stage metaprogramming the above procedure can be repeated in infinite nesting level. Usually, the metaprogramming language is quite different from the normal language, overall resulting in two distinct languages with separate respective implementations.

In our work, we focused on JavaScript and extended the language with a metaprogramming layer programmable directly in JavaScript. This way, we managed to deliver a full-scale metaprogramming framework where: (i) the normal language is a pure subset of the meta-language; and (ii) they share the same underlying implementation. Our JavaScript multi-staging support has been implemented by extending the open source Mozilla SpiderMonkey JavaScript engine. We indicate the minimal modifications required in SpiderMonkey by performing minor extensions in the syntax, parser and internal AST structures, and the addition of an unparser, a staging loop, some library functions and a debugger backend component for AST inspection. Our approach handles each stage as an isolated entity by collecting all the meta-annotations for the specific depth, respecting the order, and creating a coherent program.

We validate our implementation through the implementation of an email application using meta-programming to fully generate the user-interface code. Additionally, we studied JavaScript source code patterns from the real world libraries and reengineered various parts to achieve improved reusability and performance through metaprogramming. Finally, we supported metaprogram debugging directly in the browser environment suing the available tools. For this purpose, we implemented a debugger client as a web page application that communicates with the extended SpiderMonkey debugger backend. The client provides extra features particular to metaprogramming debugging process, including visualizations of ASTs and respective unparsed source code fragments.

META JAVASCRIPT

Γιάννης Αποστολίδης
Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Ο μεταπρογραμματισμός είναι η ικανότητα διαχείρισης του πηγαίου κώδικα ως πρώτης κλάσης δεδομένα στο μετα-πρόγραμμα και εισαγωγής τους στο πρόγραμμα. Στον μεταπρογραμματισμό πολλαπλών επιπέδων η παραπάνω διαδικασία μπορεί να επαναληφθεί σε άπειρο βάθος, δημιουργώντας δυνητικά άπειρη αναδρομή. Συνήθως η γλώσσα μεταπρογραμματισμού είναι διαφορετική από την κανονική γλώσσα προγράμματος.

Στην δουλεία μας, εστιάζουμε στην JavaScript και την επεκτείνουμε, υποστηρίζοντας μεταπρογραμματισμό. Προς αυτήν την κατεύθυνση δημιουργήσαμε ένα σύστημα μεταπρογραμματισμού όπου η απλή γλώσσα είναι υποσύνολο της μετα-γλώσσας και μοιράζονται την ίδια υλοποίηση. Υλοποιήσαμε τον μεταπρογραμματισμό επεκτείνοντας την ανοιχτού λογισμικού μηχανή JavaScript SpiderMonkey του Mozilla. Υποδείξαμε τις μινιμαλιστικές αλλαγές που χρειάζεται το SpiderMonkey κάνοντας επεκτάσεις στο λεξικογραφικό αναλυτή, στο συντακτικό αναλυτή, στις εσωτερικές δομές το αφαιρετικού συντακτικού δέντρου (ΑΣΔ), τις συναρτήσεις βιβλιοθήκης, τους μηχανισμούς ανάλυσης της γλώσσας, καθώς και την προσθήκη ενός συστήματος για αποσφαλμάτωση του ΑΣΔ. Στην δική μας προσέγγιση κάθε στάδιο του μεταπρογραμματισμού είναι μια ανεξάρτητη οντότητα όπου συλλέγει τα μετα-

σύμβολα για το συγκεκριμένο βάθος, σεβόμενο την σειρά, δημιουργώντας ένα ανεξάρτητο πρόγραμμα.

Επαληθεύουμε την υλοποίηση μας έχοντας δημιουργήσει μία εφαρμογή ηλεκτρονικού ταχυδρομείου χρησιμοποιώντας μεταπρογραμματισμό για να παράξουμε αυτόματα τον κώδικα για τις σελίδες της εφαρμογής. Επίσης, μελετήθηκαν επαναληπτικά μοτίβα πηγαίου κώδικα σε JavaScript, από βιβλιοθήκες που χρησιμοποιούνται στην βιομηχανία, και τις βελτιστοποιήσαμε στα πλαίσια του μεταπρογραμματισμού με σκοπό να πετύχουμε καλύτερη απόδοση και βέλτιστη επαναχρησιμοποίηση τους. Τέλος, υποστηρίζουμε την αποσφαλμάτωση του μετα-κώδικα απευθείας στο περιβάλλον του προγράμματος περιήγησης. Για αυτό το σκοπό υλοποιήσαμε μία εφαρμογή σε μορφή ιστοσελίδας που επικοινωνεί με το κομμάτι αποσφαλμάτωσης του SpiderMonkey. Η εφαρμογή προσφέρει χρήσιμα εργαλεία για την αποσφαλμάτωση του μετα-προγράμματος όπως την απεικόνιση των ΑΣΔ, την εξαγωγή του κώδικα που αντιστοιχεί σε κάθε κόμβο του ΑΣΔ, καθώς και χρήσιμες πληροφορίες για το περιβάλλον εκτέλεσης.

Acknowledgements

First of all, I would like to thank my supervisor, professor of the University of Crete, Anthony Savidis, initially for trusting me and then for his continuous support and his valuable advice. I would also like to thank the Computer Science Department of the University of Crete for offering a high level of academic education and the HCI Laboratory of ICS-FORTH for providing a high-level research environment.

I would also like to thank my friends for supporting me all through this period. Finally, most of all, I would like to thank my parents Eleni, Panayiotis, and my brother Pantelis for their tolerance and their oversupply to any of my needs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the history of software developing, the programming languages involved when advanced programmers massively begin to design and write source code using a specific technique. For example, when programmers were trying to associate data with code in one entity, the Object Oriented Programming (OOP) was introduced. Design Patterns are the current motivation, because programmers are developing source code using specific techniques. In this context, there is a problem that had not occurred in the pipeline of programming history. Programming languages have rapidly grow up and the quantity of Design patterns is huge. Additionally, new software engineering requirements came up for better code reusability. We solve the requirements of this motivation using **multi-staged metaprogramming.** Metaprogramming is the process of creating a program that creates a program. Multi-staged metaprogramming can be described as the process of creating a program that creates a program in infinite level.

Multi-staged programming languages [14] evaluates programs sequentially before the final program, this programs are called **metaprograms**. Metaprograms treat source code as first-class data and emit source code to the program. First-class data, in a programming language, is all the possible types that can be assigned in a variable (integer, string etc.). We extend those data with the ability to carry source code. Usually the form of source code in variables modeled as an Abstract Syntax Tree (AST) [23].

Each of the metaprograms is an isolated entity, where its purpose is to emit source code to the next program. This procedure is called **stage**. The execution order of each stage

defines the staged depth. Potentially, multiple stages can be executed, thus we use the term multi-staged metaprogramming. Briefly, it is the way to *create a program that creates a program* in infinite depth. Essentially, when a program A emits source code to a program B, and program B emits source code to a program C, program A is a metaprogram in stage level 2, program B is a metaprogram in stage level 1, and program C is the final program. Figure 1 illustrates a typical metaprogramming staged process. *Original program* contains the source code of our program annotated with code (metacode) that will run before the main program (staged program). The *staging process* is essentially the staged execution phase. The staged program evaluates the metacode and generate the source code that will be emitted to the next program. *Updated program* contains the original program with the source code that emitted from the staging process.

There are different approaches regarding the execution time of metaprogramming. Mainly, there are two categories. The metaprogramming executes during the compilation or during the runtime phase. In our work, we focus on compile-time metaprogramming. Compile-time metaprogramming offers the ability to write sophisticated and performance heavy metaprograms without effecting the runtime performance.

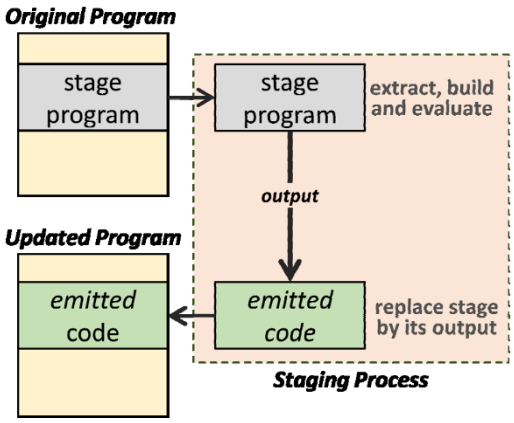

Figure 1 – Stage evaluation process in multi-staged languages.

There is a variety of applications under the umbrella of metaprogramming:

- Code analysis: Observe the structure of a program to compute a value, unroll loops, inline functions.

- Aspect-oriented programming: Apply behavior and functionality to source code segments, such as add a log statement in the beginning of each function.

- Exception handling: Abstraction of the exception handling logic, eliminating the boilerplate code of an exception statement.

- Source code validation: Verify the source code well-formedness.

- Model-Driven engineering: Solve maintenance issues in MDE development process [22]

- Custom source code manipulation: Manually manipulate the source code aiming to collapse code repetitions or generate helpful structures.

1.1. JavaScript

Multi-staged metaprogramming can be implemented as a feature in any language. We choose to implement metaprogramming on a widely used language. JavaScript is standardized on ECMAScript [24] language specification. It is a dynamic prototype-based object-oriented programming language. Initially used as an integral part of scripting in web browsers. By the years, as the web users increased, better JavaScript implementations integrated. Many mechanisms developed in the language implementation stack, such us Just-In-Time compilation (JIT), bytecode generation, dead code eliminations, loop-invariant code motion and fold constants, aiming to accelerate the runtime performance. JavaScript is currently used in many application fields. Usually, the applications that embed JavaScript, keeps the language functionality invariant and inject a custom object, providing access to the internal structures. We briefly mention the most known tools on each application field at Table 1.

Application field	Implementation	Details
OS for desktop	https://node-os.com	An operating system build entirely in JavaScript and managed by npm [26]
OS for mobile/TV	https://www.tizen.org	An operating system based on the Linux kernel, providing application development tools based on the JavaScript
Robotics	http://sbstjn.github.io/noduino	A simple and flexible JavaScript and Node.js Framework for accessing basic Arduino controls from Web Applications using HTML5, Socket.IO and Node.js
Windows/Linux/ Web/Phone apps all in one	http://electron.atom.io	A framework to develop cross-platform applications using JavaScript, HTML and CSS. It is based on io.js and Chromium
Graphics	http://voxeljs.com	A collection of projects to create 3D voxel games like Minecraft all in the browser running WebG
Graphic card drivers	http://www.slideshare.net/jarrednicholls/javascript-on-the-gpu	Running JavaScript on GPU
Game engines	http://docs.gameclosure.com	HTML5 JavaScript game development kit compatible on Browsers, Android, IOS
Server	https://nodejs.org	A platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications

Table 1 - JavaScript application fields.

1.2. SpiderMonkey

JavaScript engine has many implementations in different platforms and programming languages. We cannot admit that a specific implementation is more powerful than other. Each of the current JavaScript engines carries different advantages and disadvantages. We choose to extend the SpiderMonkey [27] JavaScript engine. SpiderMonkey initially developed at Netscape Communications. Now it is an open source project maintained by the Mozilla Foundation. Spidermonkey embedded in various well-known applications like Mozilla Firefox, Thunderbird, Adobe Acrobat, Dreamweaver and Yahoo widgets and GNOME 3. Internally, it is written in C/C++ (cross-platform) and contains an interpreter, a JIT compiler (IonMonkey), a garbage collector and mechanisms like fold constants. We thoroughly choose to implement metaprogramming in Spidermonkey because it is open source, widely used and clearly documented.

1.3. Implementation

During the design and the implementation of the multi-staged model, we aim to convenience the user of the language, but also simplify the implementation process for the author of the language. To address those challenges, we focus on the maximum reusability of the current language tools, avoiding to reinvent the wheel. Putting all the requirements on a straight line, we implement a multi-stage metaprogramming extension on top of Spidermonkey JavaScript engine. As we outline in Figure 2, we thoroughly implement some minor extensions. As a result, we yielded a full-scale multi-stage metaprogramming system for JavaScript. Furthermore, we enrich the staging model with extra features, introduced in [15].

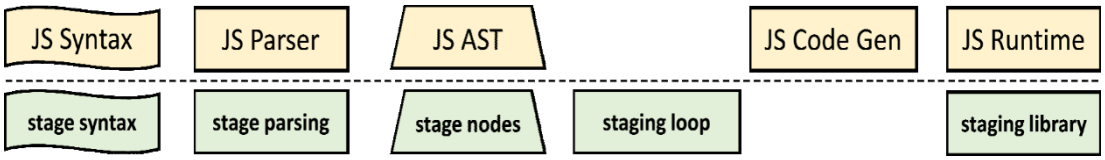

Figure 2 - Outline of the original JavaScript components in the SpiderMonkey implementation (top layer) and the respective extensions we have introduced to support staging (bottom layer).

We enabled the use of browser debugging tools, in order to debug the stages (metaprograms). The frontend of the debugger support: (i) inspection and visualization of ASTs, with multiple views and code unparsing; and (ii) seamless transition to the next stage with the normal step debugging command issued on the last statement of the current stage.

Chapter 2

Related Work

Our approach operates as a multi staged macro system. In bibliography, we meet a huge variety of macro systems, several with staged techniques. The latter years, a lot of JavaScript code and documentation introduced to solve erroneous practices, generate optimizations, organize the source and extend the language. We focus on a multi staged macro approach, considering that we minimally extend the SpiderMonkey engine.

A macro is a rule or pattern that specifies how a certain input sequence should be mapped to a replacement output according to a defined procedure. The mapping process that instantiates a macro, used inside a specific sequence, known as macro expansion. A typical macro system evaluation depicts in Figure 3. The first entity in the figure depicts the process of evaluating a macro and emits the result to the *normal program*. This procedure repeated until no macro definition occur. After the previous procedure, pure JavaScript source will be exported. Pure JavaScript source means that there will be only pure source code without the macro definitions. There are two kind of macros, (i) Text-Based; and (ii) Syntax-Based.

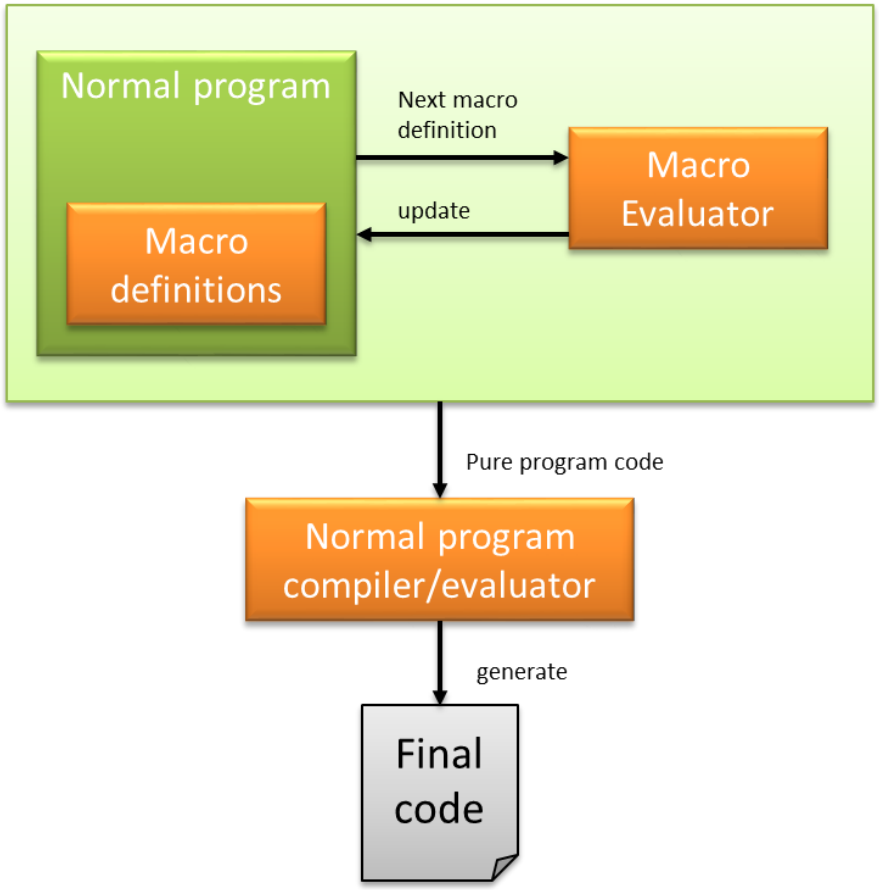

Figure 3 – Typical macro system evaluation process.

2.1. Text-Based Macros

Text-Based Macros are external preprocessors that are usually independent from the programming language. Especially, this type of macros lack of knowledge about the host language. Text-Based macros work by simple textual search-and-replace the token, rather than the character level. They process the target source as a pure text file and perform text substitutions, generating the final source file. The language execution environment is unaware of the appearance of the preprocessor. This procedure generates many problems in the correctness of the final source code. For example, our text substitution rules could potentially harm the language keywords, or generate a syntax that is not valid for the specific language. Observably, the possibility of side

effects and erroneous results in extremely probable. Text-Based preprocessors do not provide any safety about the substitution environment.

There is no guaranty about the source code stability and the result of bindings in the enclosed environment.

The most popular and representative Text-based preprocessor is the C preprocessor. The C preprocessor is the default macro preprocessor for the C and C++ programming languages. The preprocessor performs preliminary operations on C and C++ files before they are passed to the compiler. Preprocessor can be used to conditionally compile code, insert files, specify compile-time error messages, and apply machine-specific rules to sections of code. In practice, it is a separated program invoked by the compiler as the first part of translation. The example bellow illustrates a typical preprocessing example:

```
#define IF_VALID else if(state.isValid())

if(state.ready() {                  if(state.ready() {
printf("ready");        ⇨           printf("ready");
}IF_VALID                           }else if(state.isValid())
printf("valid");                    printf("valid");
```

The preprocessor will correctly produce an if-else-if statement. In contrast, a misuse of the IF_VALID definition may generate an invalid source code. The next example depicts a modified version of the previous example:

```
printf("validating");       printf("validating");
IF_VALID                ⇨   else if(state.isValid())
printf("valid");            printf("valid");
```

The preprocessor, as a pure Text-Based tool, will not identify the inaccuracy of the result. Subsequently, the error will appear at the later compiler steps.

2.2. Syntax-Based Macros

Syntax-Based macros are evaluated on the Abstract Syntax Tree (AST) of the source code. Usually, during the parsing phase of the programming language, they transform the macro expressions to ASTs. The parser identifies the AST, evaluates it and tries to substitute into the enclosing AST. A further step in the macro procedure that Text-Based processing does not offer, is that the macro generated AST will only be substituted, if the enclosed AST is syntactically matched. For example, a macro that is enclosed by a while block statement expects a single statement, in case that the macro generate a different AST, an error will be occurred. This procedure offers a proper safety and it guaranties that the substituted macro will be well-formed.

An interesting Syntax-Based macro approach is **Sweetenjs** [1]. Sweetenjs propose a novel solution to solve the bidirectional dependency issue between lexical and parsing phase in the JavaScript implementations. The Sweetenjs authors argue that the specific disambiguate methodology can be applied to any other language. After the separation of lexical and parsing phase, they implement a hygienic macro system for JavaScript. Especially, they insert an extra component between the lexical and the parsing phase, called *reader*. Reader takes tokens from the lexer as input and constructs a token tree. Autonomously, it decides when a token tree is ready. Finally, the token tree intends to be the input of the parsing phase. This procedure cuts the inform dependency circle between lexer and parser. Furthermore, they introduced the term enforestration [3], which is a methodology to convert a flat stream of tokens into an S-expression form. Essentially, S-expression form consists the token tree. **Honu** [4] is the first programming language that adopt this technique. Additionally, Sweetenjs implements a hygienic macro system for JavaScript. It is a custom language (different from JavaScript) that provides the ability to extend the JavaScript syntax. The extended syntax is the following:

```
macro <name > {
  rule { <pattern > } => { <template > }
}
```

Exploiting the advantages of this syntax, we gain the ability to develop many interesting expressions. For instance, we can generate a function declaration expression, as depicts bellow:

```
macro def {
  rule {
    $name ( $params ( ,) ...) { $body ... }
  } => {
    function $name ( $params ...) {
     $body ...
  }
}
```

After the specific declaration, we can use this function definition form, as a macro, to the source code:

```
def id (x) { return x; }
//expands to:
function id (x) { return x; }
```

ExJS [13] is an extension of JavaScript that offers syntactic extensibility of the core language. These macros defined by a simple pattern matching technique, similar to syntax-rules pattern that appears in Scheme. ExJS uses a multi-staged parsing technique and generates context-dependent syntax rules. The following steps outlines the pipeline process of parsing:

i. Analyze the syntax of macro forms.

ii. Generate PEG grammars.

iii. Combine JavaScript and macro to a macro-aware form.

iv. Parse the macro-enabled program.

v. Convert the AST to S-expression.

vi. Macro-expand the AST to S-expression.

vii. Convert the S-expression back to macro-free JavaScript.

Programmer can use ExJS to generate expressions and statements in JavaScript using a special syntax. For example:

```
statement unless {
  expression : C;
  statement : S;
  { unless (C) S => if (!C) S }
}
```

is a macro directive that defines a custom *unless* statement in JavaScript. *Unless* statement constitutes of the following ingredients, an expression C, a statement S and the corresponding macro expansion. Henceforth, programmer can produce statements like:

```
unless(key=="A"){
  console.log("A not pressed");
  if(key=="B")
    console.log("B pressed");
  else
    console.log("A and B not pressed");
}
```

After the evaluation of the macro, the expression will be expanded to:

```
if(!key=="A"){
  console.log("A not pressed");
  if(key=="B")
    console.log("B pressed");
  else
    console.log("A and B not pressed");
}
```

2.3. Metaprogramming

Haskell [5] is a standardized, strongly-typed, general-purpose, purely-functional programming language. It support compile-time meta-programming through some semantics, similar to Multi-Staged JavaScript. Haskell macro system inlines source code

to the main program. The macro procedure evaluates at compile time. Haskell generates macros by creating the AST from the corresponding source code. It manipulate and substitute the AST in the context. Metaprogramming facilities are implemented by reusing all the pure language features. The ASTs are modeled as data structures. **Template Haskell** adopts two basic semantics. *Quazi-quote* brackets, illustrated by [| ... |], used to yield the corresponding AST from the enclosing expression. *Splice* brackets, $(...), used to convert the AST to source code.

Converge [6] is a dynamically typed object orientated language which allows compile-time meta-programming in the spirit of Template Haskell. Quasi-quote, [| ... |], produce AST. In Converge, ASTs called ITrees. Similar to Template Haskell, Converge contains the splice annotation $<<...>>. Splice evaluates its enclosed expression and substitutes itself with the ITree yielded from the evaluation. We can generate interesting macros when we combine Quazi-quotes and splicing. Noteworthy is the fact that the compile-time metaprogramming is not effecting the final bytecode and the Virtual Machine instruction generation, since it acts at parsing time. We depict a typical implementation of the function *make_power* in Converge:

```
func expand_power(n, x):
  if n == 0:
    return [| 1 |]
  else:
    return [| $c{x} * $c{expand_power(n - 1, x)} |]

func make_power(n):
  return [|
    func (&x):
      return $c{expand_power(n, [| &x |])}
  |]

power3 := $<make_power(3)>
```

Metalua [7] is an extension of Lua. Lua is dynamically typed. It exposes the generic principles of meta-programming assets like meta-levels in sources, AST representation of code and meta-operators. It provides compile-time metaprogramming and offers the ability to extend the syntax of Lua. Metalua have special individual metaprogramming

semantics. Quazi-quotes, +{ ... }, produce the AST of the enclosed expression and increase the staged level by one. Programmer is free to produce unlimited staged levels. The splicing functionality produced by the contrary -{ ... } semantic. Splicing expression reduces the staged level by one. The AST will be emitted to the source code when the staged level decrease to -1. For example:

```
-{stat:
  function ternary (cond, b1, b2)
    return +{ (function()
                    if -{cond} then
                      return -{b1}
                    else
                      return -{b2}
                    end
                end)() }
  end }

lang = "en"
hi = -{ ternary (+{lang=="fr"}, +{"Bonjour"}, +{"Hello"}) }
print (hi)
// prints "Hello"
```

Additionally, Metalua provides some syntactic sugar extensions,

```
function(arg1, arg2, argn) return someExpr end
```

can be written:

```
|arg1,arg2,argn| someExp
```

Metalua contains a wide variety of tools, in order to implement the metaprogramming logic. We briefly quote the most essential subset. MLP is the dynamically extensible Metalua parser. GG is the grammar generator. MATCH is an structural pattern matching. WALK is a code walker generator. HYGIENE offers hygienic macros. Runtime libraries for staging and AST manipulation (metalua.runtime, metalua.compiler). Although the original Lua language components are fully available in stages, Metalua consists of a separated scanner, parser, and a custom reimplementation of its virtual machine for stage evaluation.

Groovy [8] is an object-oriented dynamic programming language for Java. Usually, it is used as scripting language for Java environment. Groovy supports runtime and compile-time metaprogramming. We will focus on compile-time metaprogramming, since runtime metaprogramming is a runtime reflection-related mechanism that JavaScript already encompasses using eval. Compile-time metaprogramming in Groovy allows code generation. It modifies the AST during the compilation process. The metaprogramming procedure operates by using semantic annotations at the target source segment. Groovy offers two metaprogramming alternatives. First, it provides a huge mixture of AST transformations. Those transformations covers a large portion of the programming requirements. Some of those operates on reducing the boilerplate source code, implementing design patterns, logging, declarative concurrency, cloning, safer scripting, tweaking the compilation, implementing Swing patterns, testing and eventually managing dependencies. For example, we mention a metaprogramming example in the context of *ToString* transformation facility:

```
import groovy.transform.ToString

@ToString
class Person {
  String firstName
  String lastName
}

def p = new Person(firstName: 'Jack', lastName: 'Nicholson')
assert p.toString() == 'Person(Jack, Nicholson)'
```

Secondly, programmer can manually define a custom AST transformation by implementing the *ASTTransformation* interface. This interface provides an entry point to access and manipulate the AST. The transformation applies by annotating the source code with the regarding markers.

In the aspect of language authoring, the implementation of metaprogramming in Groovy reuses the language compiler and the runtime system. Furthermore, it provides the proper debugging mechanisms to inspect the AST transformations directly from the IDE [9]. In contrary, programmer has not the ability to see the exported AST. Moreover,

Groove does not support multi-staging, hence, it is impossible to implement sophisticated staged evaluations, such as metagenerators. Additionally, transformation components are isolated, thus, integrated metaprograms that systematically transform the main program are unachievable.

Lisp is a fully parenthesized or s-expressions prefix notation programming language, introducing many primary compute science notions like artificial intelligence research, tree data structures, dynamic typing, self-hosting compiler. A premium data structure is the linked list. The entire Lisp source code consists of lists. Many of the Lisp implementations share the same interpreter for both language and macros. Its main derivations are **Common Lisp** [11] and **Scheme** [10]. Common Lisp manipulate the source code as data structure using Macros. ASTs can be generated using list processing functions or quasi-quotes directive. Programmer has the ability to traverse and manipulate the generated AST.

```
(defmacro until (test &body body)
(let ((start-tag (gensym "START"))
      (end-tag   (gensym "END")))
  `(tagbody ,start-tag
            (when ,test (go ,end-tag))
            (progn ,@body)
            (go ,start-tag)
            ,end-tag)))
```

```
(until (= (random 10) 0)
  (write-line "Hello"))
```

Lisp yields the surrounding source code to a transformed source form. This procedure repeated until the elimination of macros. The final source code is executed at runtime. Common Lisp provides name capturing and a hygienic macro mechanism that guarantees the uniqueness of variables using the directive *gensym*. Scheme macros are transformation procedures accompanied by a simple pattern matching sublanguage. It provides a hygienic macro system using syntax-case clauses. These clauses offers the ability to selectively apply variable capturing. The pure execution system of Lisp traced via the normal debugger. In contrary, programmer must use a special macro stepper to inspect and trace the AST transformations. Generally, Lisp macro system does not

provide a coherence metaprogram. Programmer treats each macro as an individual expression, so it is impossible to combine macros in the same context. Hence, Lisp macro system does not collects all the macros to a staged coherence metaprogram, decreasing the expressive power.

MetaML [12] is a general-purpose functional programming language that extends the ML programming language. It reuses the pure language features and tools, like the compiler and the runtime system. MetaML belongs to the multi-staged runtime metaprogramming family, based on annotations. More specific, it uses three annotations. Brackets <>, to delay the reduction of the enclosed expression. Escape ~, to splice multiple delayed expressions together and generate a larger expression. Inline *run*, to force the reduction of a delayed expression.

For example the power generator function can be written as,

```
fun power n = fn x => if n=0
  then <1>
  else < ~x * ~(power (n-1) x) >)
```

We can use *power* to the following expression,

```
map (run <fn z => ~(power 2 <z>)>) [1,2,3,4,5]
```

after the evaluation of the annotated expression, the result will be:

```
map (fn z => z * z * 1) [1,2,3,4,5]
```

MetaML reuses all the components of the pure language, but it lacks of a metaprogramming debugging system.

Chapter 3

Staged JavaScript

3.1. Semantics

Spidermonkey JavaScript engine has extended to support multi-staged metaprogramming. Staged semantics must be evident and easy to use. The general purpose of this thesis is to achieve hygienic, well-formed, ease-of-use invocations for the language user. Reflecting these principals, we define staged the syntax tags and semantics.

3.1.1. AST Manipulation Tags

Staged programs injects AST definitions in the next program, this procedure repeated until the elimination of staged symbols. Essentially, all the staged process elaborates AST creations and concatenations. To achieve this functionality, we introduce the following AST manipulation Tags. Such tags are provided to ease the composition of the AST. Practically, they reify the language parser, the internal AST structures and the functions that creates an AST.

Quazi-quotes (.< >.) are holding JavaScript code definitions in order to convey them to an AST form. Essentially, it is a syntactic sugar tag that converts a source text to AST.

Quazi-quotes assists to generate hygienic ASTs and prevents the programmer to write boilerplate code. Variables that are defined within the quasi-quotes binds its scope in the context where the respective AST will be finally inserted. Essentially those variables are lexically scoped at the insertion point.

```
var xDeclAst = .< var x = 1 + 4; >.;
```

```
var xDeclAst = { loc: {
    start: { line: 1, column: 0 },
    end: { line: 1, column: 5 },
    source: null },
  type: "Program ", body: [
  { loc:..., type: "VariableDeclaration", kind: "var", declarations: [
    {  loc: ..., type: "VariableDeclarator",
      id: { loc: ..., type: "Identifier ", name:"x"},
      init:{loc:..., type: "BinaryExpression", operator: "+",
      left: {
        loc: ...,
        type: "Literal",
        value: 1
      },
      right: {
        loc: ...,
        type: "Literal ", value:4}
      }
    }
]} ] }
```

Escape (.~) can be defined only inside a Quazi-quote definition and its argument can only contain a JavaScript AST object. Essentially, it prevents an expression to convert its context to AST by evaluating its current value. Usually, it used to incorporate a previously generated AST to the current Quazi-quote context. For example, assume the statements *.< var x = rand(); .~assertAst; y = x; >.* , and a variable that has already defined *assertAST = .< assert(x); >.* . Our purpose, after the evaluation of that code snippet, is to prevent the variable *assertAst* of converting as it is, but replace the *.~assertAst* with the carrying *assertAST* value.

Escape-value (.@) can be used when we need to prevent the conversion of an expression to AST by matching the current evaluated value to its corresponding AST

form. So it acts as Escape, the difference is that Escape-value argument is only JavaScript first class value (number, string, Ec.).

3.1.2. Staged Tags

Staged tags generate the metaprogramming logic in the program. Essentially, they define the code that will produce the stage evaluation, the nesting depth and indicates the segments where the staged code will be injected to source.

Inline (.!) injects the declared location with the AST of the evaluated value, eliminating itself. Inline is actually the core of the staging program because it generates the staging process and modifies the main program. When an Inline expression is defined inside a Quazi-quote, it behaves normally, thus the corresponding AST will be created. When an inline expression emit an AST, many interesting functionalities can be produced. A quite common pattern that appears in metaprogramming is the *metagenerators*. A metagenerator is a directive that can produce infinite nesting stage by generating other metagenerators statically or dynamically. For example *.!(.<.!.(<print('stage 1').>).>)* will inject to the source the *.!.<print('stage 1').>*, as a result, this expression will generate an additional staging depth that will inject the *print('stage 1')* to the final program.

Execute (.&) defines a statement that will be executed in the staged process. The staged depth increases when execution symbols defined sequentially. For example, *.&.&.&statement* executes the *statement* in the stage depth 3.

In our approach, the execution environment of each staging depth is isolated. Essentially, each staged depth behaves as a unique program. The only communication between each stage is the potential injection of the enclosed inline. The execution order of the *execute* and *inline* statements during the staging process is corresponding to the definition order in the source code.

For instance, we demonstrate the ordinary power generator function. We aim to accelerate the power functionality by calculating, at preprocess time, the multiplication of the base variable.

```
.& {
  var gen_power = function( baseAst, exponent ) {
    var resAst = .< .~baseAst; >.;
    for(var i=0; i<exponent; ++i) {
      resAst = .<
        .~resAst * .~baseAst;
      >.;
    }
    return resAst;
  };
};

x = Math.random();
y = Math.random();
var rand = .!gen_power( .< x + y; >., 10 );
console.log( rand );
```

After the evaluation of the staged process, the final source will be exported as:

```
x = Math.random();
y = Math.random();
var rand = (x + y) * (x + y) * (x + y) * (x + y)
           * (x + y) * (x + y) * (x + y) * (x + y)
           * (x + y) * (x + y) * (x + y);
console.log(rand);
```

3.2. Staging Process

Multi-staged metaprogramming in Spidermonkey operates as a preprocess tool. Its input is a pure JavaScript source code, enriched with meta-JavaScript source code. After the preprocess evaluation it will export pure JavaScript source code. During the above pipeline process, many steps are formed to achieve the result. Functionalities like parse and evaluation of JavaScript source code are certainly required. As we mentioned, we extend the Spidermonkey JavaScript engine in order to implement the preprocessor logic. For our purposes, we modify and extend many parts of the engine. More specific,

we can separate the preprocess procedure in some logical parts. Figure 4 illustrates the architecture of the process, bellow we list the steps:

1. Call Reflect.parse() JavaScript library function setting the source text as argument and yielding a JavaScript AST object. This object essentially represents the AST of the corresponding source file.

2. Iterate the AST object, identifying the innermost stage level.

3. Assemble the integrated metaprogram for the specific nesting level, respecting the appearing order in the source file. Convert Quazi-quotes definitions to JavaScript AST object incorporating the escape values. Internally mark the inline nodes in the AST.

4. Execute the integrated program. Due to the execution, inline functions will substitute the marked inline nodes with the result of their argument, and will erase themselves.

5. Repeat the process, starting from step 3, until the elimination of the staged symbols.

6. After the previous steps, AST contains only pure JavaScript nodes. Finally, the unparse function generates the final source file from the corresponding AST.

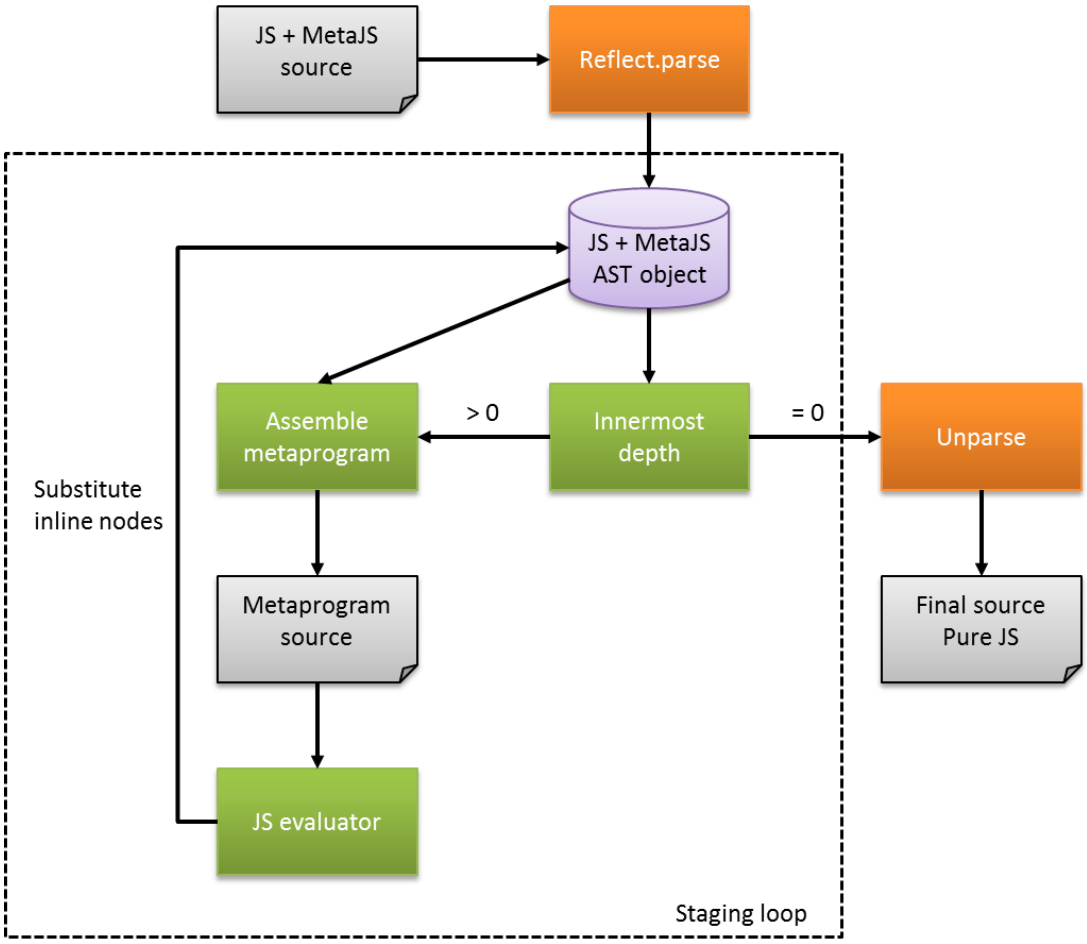

Figure 4 - Diagram of multi-staged metaprogramming execution flow in SpiderMonkey.

Initially, we aim to adopt the internal AST structures of SpiderMonkey (ParseTreeNode in C++) and use them throughout the staging process. Then, we observe that using the original AST structures entails a few issues. For instance, we would miss many features regarding the tree editing and composition, since ParseTreeNode has not designed to be mutable during the translation process. Then, we saw that SpiderMonkey defines the internal AST structures in a form of runtime JavaScript objects. Those objects consists well-formed AST structures. Also it provides methods to convert ParseTreeNode* to JSObject and vice versa. Essentially, this functionality provides a kind of reflection mechanism. Considering the above, we decide to adopt the JavaScript AST object *JSObject**. Hence, all the transaction relating the AST effects JavaScript AST Objects.

The staging loop, in Figure 4, essentially instantiates the overall staging process. It is invoked exactly after the initial parsing process, it takes as input the AST of the program, which includes the meta tags. After the staging evaluation it results a modified AST, without staging-related tags. The assembled metaprogram is a coherence JavaScript program. The JavaScript engine creates a new execution instance that unparse and evaluate the metaprogram. The unparsed source is saved to disc, assisting programmers to view the entire stage, outside the enclosed program.

During the assembly process of the metaprogram, a list of inline nodes is created regarding the definition order in the source. Actually, those references are the positions of the inline tags in the AST of the program. Inline function uses this list in order to substitute the next inline node in the list with the enclosed AST.

3.3. Spidermonkey: Modifications and Additions

3.3.1. Staged Syntax

3.3.1.1. Lexical

In lexical phase, SpiderMonkey process the source characters sequentially and transform them to tokens. Staging tags should look much more different than the rest JavaScript tokens, mainly for two reasons. Apparently, staged tags must have no conflict with the current or potentially future reserved JavaScript tags. Secondly, the staged phase is evaluated preceding the main JavaScript program, so the syntax must seem different. All the staged tokens are prefixed or postfixed by the dot (“.”) punctuation mark. Consequently, the staged tokens are the following:

- Quazi-quotes“ .< ”, “ >. ”

- Escape “ .~ ”
- Escape-value “ .@ ”
- Execute “ .& ”
- Inline “ .~ ”

3.3.1.2. Grammar

The implementation of the parser in SpiderMonkey is custom. Hence, tools for automatic grammar generation are missing. We insert the grammar extensions hardcoded, following the internal implementation of SpiderMonkey in similar grammatical rules. In the syntax of JavaScript, the rules are derivates from either a statement or expression. Our grammar extensions behaves respectively.

Extension Rules

unaryExpression -> .~ expression
unaryExpression -> .! expression
primaryExpression -> .& statement
unaryExpression -> .@ expression
primaryExpression -> .< statements >.

We define the grammar of staged tags.

- Primary-Expression -> Quazi-quotes (.< statements >.)
  Can be posed as derivative of primary expression.
- Unary-Expression -> Escape (.~expression), Escape-value (.@expression), Inline (.!expression)
  For grammar simplicity those rules are chosen to be placed as derivative of unary expression. This decision diminishes expressiveness regarding the usage inside the source, in the other hand, it simplifies the grammar complexity.

- Primary-Expression -> Execute(.&statement)

AST Additions

After the grammar modifications, new AST node types inserted to the AST structure. Each of the grammatical rules have a respectively addition to the SpiderMonkey AST definition. In SpiderMonkey, the AST nodes defined under the *ParseNodeKind* enumerated type (*PNK* also used as a synonym). The following code segments depicts the creation of the meta nodes:

```
new_<UnaryNode>(PNK_METAQUASI,JSOP_NOP,pos,defs)
unaryOpExpr(PNK_METAESC,JSOP_NOP,expr)
new_<UnaryNode>(PNK_METAEXEC,JSOP_NOP,pos,stmt)
unaryOpExpr(PNK_METAINLINE,JSOP_NOP,expr)
```

Finally, many AST structural assertions eliminated. For example, in our approach, an escape or inline expression can be part of a function name or argument. This type of syntax is not supported by the normal language.

3.3.2. Parse Source Text

In programming languages, parsing source text is defined as the procedure of transforming a sequence of characters in a target structure. Our purpose is to transform JavaScript source text to JavaScript AST object. AST object is a valid JavaScript object that essentially encompasses the definition of a JavaScript source code in an AST form. For example, Figure 5 illustrates the AST form of *print('hello')*.

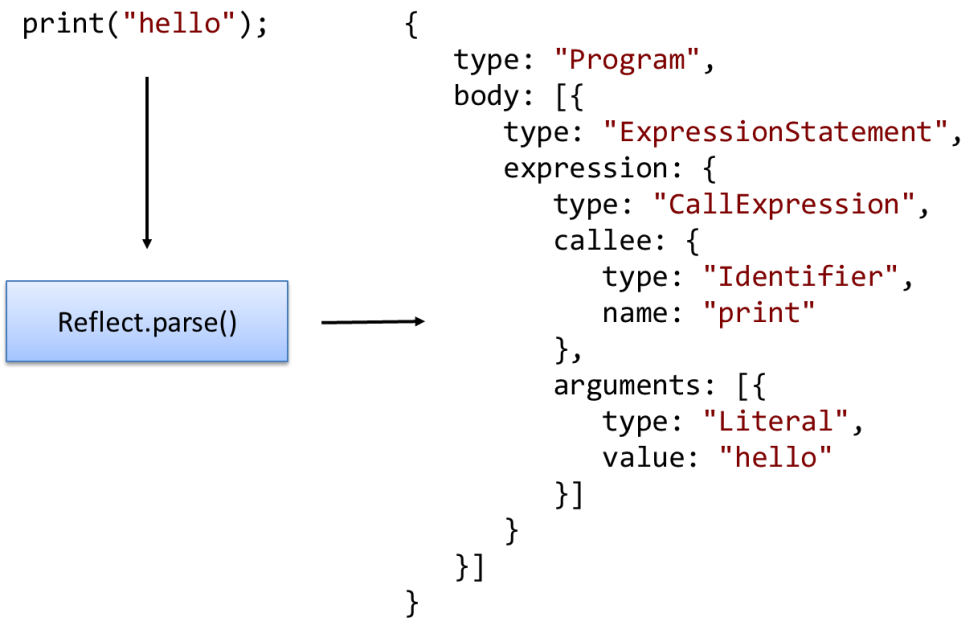

Figure 5 - Reflect.parse() functionality. Transform JavaScript source text to a JavaScript AST object.

SpiderMonkey already incarnates the parse logic, by reflecting the internal parse functionality, using the Reflect.parse [16] extension. A rich variety of tools evolved around this functionality. A notably example is the popular Esprima parser [28], other applications regarding the source code editors, IDEs, and convertors. Almost all the modern projects that contain this kind of functionality, adopt the AST object form of Spidermonkey. Hence, the AST object of Spidermonkey is currently the unofficial standard definition. Our extension over Reflection.parse was the definition of the staged tags. Firstly, in the representation of AST object, and secondly in the core parsing logic.

3.3.3. Unparse AST

Unparse AST is the procedure of transforming an AST to a precise source text representation. Figure 6 depicts an example of *print('hello')*. Unparse AST functionality is completely missing from the SpiderMonkey assets, hence, we implement it from scratch. Generally, we argue the equality of the following formula:

Unparse(Parse(Source-Text)) == Source-Text.

This is not precisely equal in JavaScript. Intuitively, this peculiarity seems to generate enormous inaccuracies. For instance, a string definition in JavaScript can be defined using either (‘’) or (“”). During the parsing procedure, when a string is recognized, an AST node that contains the string value is generated. Henceforth, there is no way to separate the string punctuation definition origin. Thus, during the unparsing procedure, each string node in the AST will be interpreted using (“”) punctuation delimiters. Therefore, this bidirectionality leads to the same coherent program. Conclusively, we argue that even if the equality is not strictly the same, essentially it results the same program.

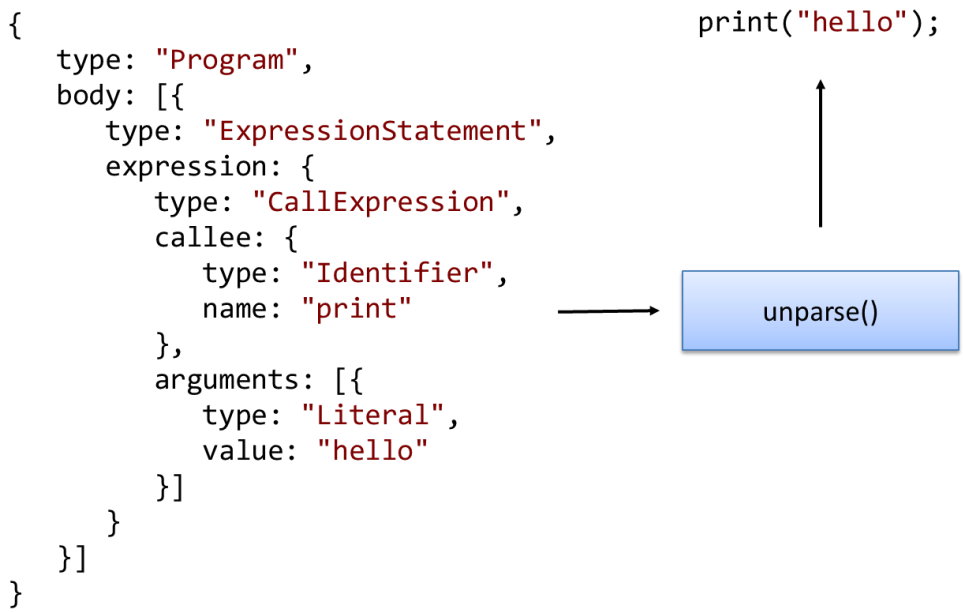

Figure 6 - Unparse functionality. Transform JavaScript AST object to JavaScript source text.

During the unparse procedure, some AST nodes have special treatment. Specifically, as we previously mentioned, the AST manipulation tags are syntactic sugar for the AST manipulation. Considering this, we treat the unparse procedure of those nodes conceptually.

Quazi-quotes is essentially the syntactic sugar expression for convening a source text of a JavaScript definition to its AST form. To achieve that, we use the SpiderMonkey AST

definition. Thus, a JavaScript source definition around Quazi-quotes is strictly equivalent with a JavaScript AST object that describes the same definition. Consequently the unparse procedure yields source text from JavaScript AST object.

Escape, as we mentioned, prevents an expression inside a Quazi-quote definition to be parsed as simple AST. We handle this, by replacing the escape node with a function that takes the Escape definition as argument and concatenates the corresponding AST with the Quazi-quote parent AST, as shown in Figure 7.

Escape-value acts precisely as Escape, the only difference is that Escape-value argument is pure JavaScript first class value (boolean, number, string, etc.). For example, assume the expression *.<print(.@x, 4)>.*, assume that *x* is previously defined and contains the string "hello". This expression evaluates *to print("hello", 4)*. Therefore, during the unparse procedure, the Escape-value node is substituted with the corresponding runtime JavaScript function *escape_value*.

Figure 7 indicates a comprehensive example of transformation of Quazi-quote and escape.

Figure 7 - JavaScript meta-constructor before and after the staged phase.

3.3.4. Runtime JavaScript Library Functions

In order to execute the metaprogram in the normal Spidermonkey runtime environment, some JavaScript Global functions inserted.

Unparse function takes a JavaScript AST object as argument and returns the corresponding source text. Unparse function is the reflection of the internal Unparse AST procedure. Essentially, it incarnates the staging process 3.2. Consequence, unparse is the gemstone of the JavaScript metaprogramming edifice.

Inline function takes a JavaScript AST object as argument and substitutes itself with the enclosing program. A Runtime Exception will be thrown in cases that Inline argument is different than JavaScript AST object. The position of the inline node in the AST of the

program is internally marked during the staged phase. Inline is not appropriated for custom usage.

Escape function takes a JavaScript AST object as argument and concatenates it to the enclosed JavaScript AST object. To archive this functionality, arguments are parameterized depending on each of the following enumerated case. More specific, the escaped AST in combination with the enclosed AST, generates four possible concatenation cases. Each of the concatenation methods modifies the Escape function arguments. The escape signature is either

meta_escape(true, {normal ast nodes, and postition}, {escape ast nodes, and postition}, isFromStmt) Figure 8

or *meta_escape(false, {escape node}, isFromStmt)* Figure 9.

- Multiple statements,
  source code example: *if(1){ print(1); .!x; t=2; }*,
  formation: *meta_escape(true, [{node:print(1), index:0}, {node: t=2;, index:2}], {node:x, index:1}, true)*

- Multiple Expressions,
  source code example: *print(1, .!x, "world")*,
  formation: *meta_escape(true, [{node:print(1), index:0}, {node: t=2;, index:2}], {node:x, index:1}, false)*

- Single Statement,
  source code example: *if(1).!x*,
  formation: *meta_escape(false, x, true)*

- Single expression,
  source code example: *return .!x*,
  formation: *meta_escape(false, x, false)*

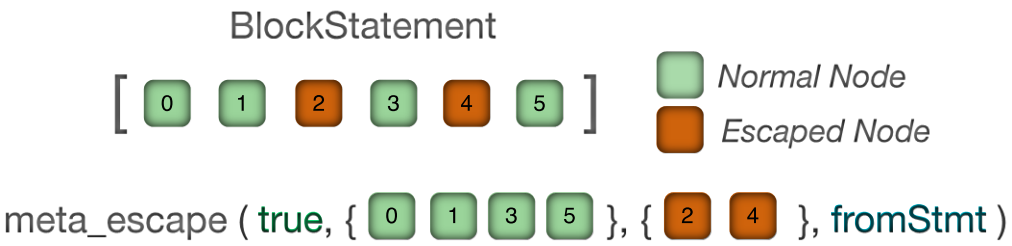

Figure 8 – Treating escape() as list of expressions or statements.

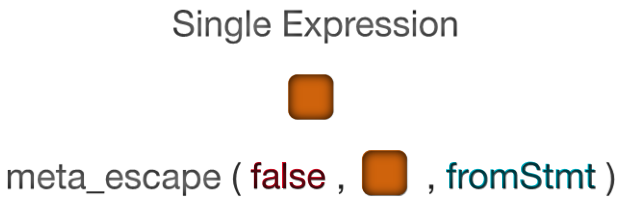

Figure 9 – Treating escape() as a single expression or statement.

Figure 10 depicts an *escape* excerpt from an AST segment that clarifies the transformation form.

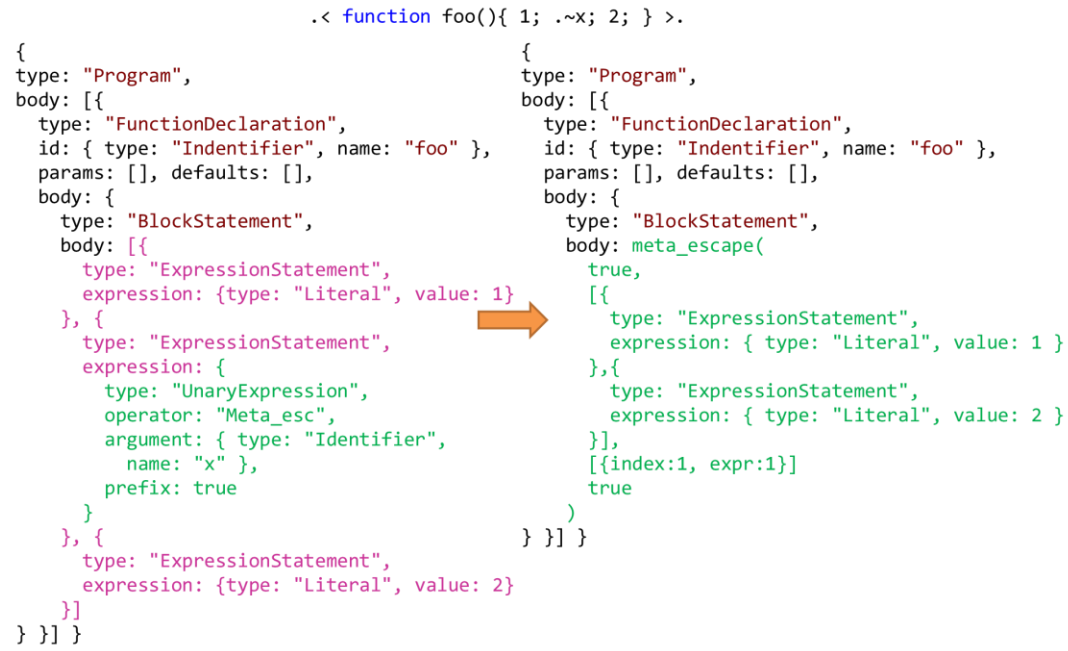

Figure 10 – Treating escape expression function call inside in quazi-quote.

escape_value function takes a pure JavaScript first class value as argument, creates and return an AST node, containing the specific value. A Runtime Exception will be thrown in cases that Escape-value argument is different than JavaScript first class value. This is very useful in cases that we need to append simple values to an AST. For example, assume the following code:

```
function genLargeElement( size ) {
  return .<
    div = document.createElement('div');
    div.style.width( .@size + 'px' );
  >.;
}
```

We set the size of the generated div element according to the *size* variable, where *size* is a pure JavaScript number.

We will continue be extending the source code in Figure 1. We compose two ASTs that carry the variables *point2dCtor* and point3dCtor, additionally we insert the following:

```
.!point2dCtor; (*staged code, 1st inline in I*)
var pt2d = new Point2d(10,20);
.!point3dCtor; (*staged code, 2nd inline in I*)
var pt3d = new Point3d(10, 20, 30);
```

In this case, during the staging phase, the inline statements (*point2dCtor* and *point3dCtor*) appended after the code block of the execute tag (.&). The assembly logic during the staged phase has been outlined earlier, in the staging loop at Figure 4.

As discussed, during the staged process, an inline list created that contains the node references for the expressions *.!point2dCtor* and *.!point3dCtor*. In the evaluation phase of the metaprogram, the inline tags rewrites the AST of the program, and replace the inline directives with the content of *point2dCtor* and *point3dCtor*. Figure 11 depicts the final source code of the program, after the evaluation of the staged phase.

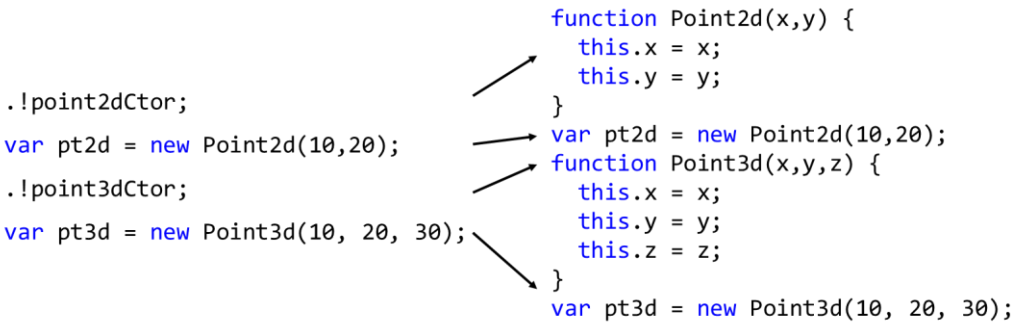

Figure 11 – Source code produced after the staged evaluation.

Chapter 4

Staged Errors and Debugging

The proper tools for meaningful error reporting and debugging are foundation concepts of an integrated development environment. A development environment that lack of essential inspection tools, consists a flavorless medium that wastes the programmer's resources. In the context of metaprogramming, we integrate a vital error reporting functionality and an essential debugging mechanism.

4.1. Error Report

The main task of the programmer during the metaprogramming, is the elaboration of the AST. Through this process, several possible errors may occur. Ideally, we would like to have a precise error report, which embodies the full trajectory of the AST concatenation. To accomplish this requirement, we reuse the source location information of the AST node, produced by the *Reflect.parse()* routine. Those information encompasses the begin and the end of the AST node. During the concatenation, that produced by *escape*, the concatenated string remains unharmed. When an error occurred regarding the AST manipulation, programmer can inspect the origin of the

corresponding AST composition parts. The following source code indicates an example of the location information attached in the AST:

```
var AstOfNumberTwo = .< 2; >.;.
```

```
var AstOfNumberTwo = {
  loc: { start: { line: 1, column: 20 },
         end: { line: 1, column: 22 }, source: null
  }, type: "Program", body: [{
    loc:{start:{line:1, column:20},
           end:{line:1, column:22}, source:null
    }, type: "ExpressionStatement",
    expression: {
      loc: { start: { line: 1, column: 20 },
             end: { line: 1, column: 21 }, source: null
      }, type: "Literal",
      value: 2
    }
  }]
};
```

Additionally, we insert many runtime sanitize checks to the staged JavaScript library function. Those checks tries to clarify the soundness of a wide variety of exploits and errors. Inline, Escape and Escape-value checks the validity of their arguments. After each AST composition, we check the hygienic of the outcome AST. In case of an error, a runtime exception will be thrown.

4.2. Debugging

Many sophisticated JavaScript debuggers involved due to the raise of applications that use JavaScript, some current examples are the Chrome developer tools [17] and Firefox inspection tools [18]. In our work, each program that assembled and evaluated during the staged process is an isolated standalone coherence JavaScript program. Consequently, metaprograms treated as pure JavaScript programs, hence metaprograms could potentially evaluated in any JavaScript engine. We export the

internal staged procedure, aiming to evaluate each stage in any external JavaScript engine. This kind of reflection, entails minor modifications to the current staged integration, but it provides the luxury of using any JavaScript debugging infrastructure. Thus, programmer is free to choose a preferred JavaScript debugger.

4.2.1. Debugging Model

Staged debugging is separated in two isolated architectural modules, which communicate each other using http requests. Backend is the component that initialize and manage the debugging flow. Specifically,

- Read the source file.
- Assemble the next staged program and mark the inline points.
- Initiate an http server to serve the staged program.

The frontend component contains two fundamental processes. The first process is the debugging evaluator. It is a web page that communicates with the backend via Ajax Json requests. Precisely,

- Receives from the backend the source code of the next stage.
- During the execution, when an inline directive reached, it notifies the server to execute the corresponding inline, assigning the AST object value that the evaluator currently contains.
- Provides the ability to inspect and visualize any AST object. To achieve this, the *STG_inspectAst(astObject)* function injected as a global library function. This function essentially sends an AST inspection request to the backend.

The second frontend component is the debugging visualizer. Essentially, it depicts useful information about the debugging process. The synchronization with the current

debugging state in the backend achieved by periodically sending Ajax requests. Precisely,

- Shows the current staging depth.
- Shows the remaining and the executed quantity of Inlines in the current stage.
- Each programmer's inspection triggered by the *STG_inspectAst*(*astObject*), rendered as an enriched interactive tree view.

We summarize the debugging architecture in Figure 12.

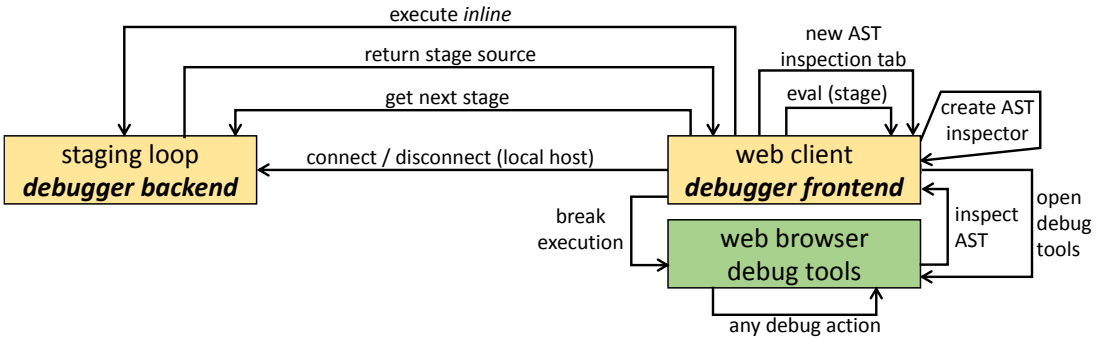

Figure 12 - Stage debugging architecture by splitting responsibilities between staging loop for stage extraction, the web browser tools for typical debugging activities, and a custom web-client for stage evaluation and AST inspection.

4.2.2. Backend

To support debugging for stages, we implement a backend service loop an extended part of the main staging loop. Basically, it responds in two requests (besides the apparent connect/disconnect ones): (i) extract and return the next stage; and (ii) apply an inline directive. Functionalities, like the method escape, implemented as JavaScript functions in the frontend, because they effect locally the ASTs. Thus, we avoid some extra messages. In contrast, this is not possible with the inline method, because it effects the main AST instance in Spidermonkey backend. The backend process flow is

the equivalent to the normal staged flow until the execution of the corresponding stage. At this point, the backend debugger initialize an http server that intended to remote debug the staged execution. SpiderMonkey does not contain any coherence http library. Thus, we include a simple comprehensive web server for handling http requests (LibMongoose [19]). LibMongoose is an open source lightweight crossplatform C/C++ http and Websocket library. We create an abstract layer over the LibMongoose library, in order to adapt it in our requirements. Henceforth, the installation of the http routes and handlers is very simple and straightforward. For example:

```
httpHandler.installRoute(
  httpRequestHandlerInfo(
    "/execinline", "application/json", "POST", &execInline
  )
);
```

Can be read as: "When we receive a post request that its content type is *application/json*, trigger the server at */execinline* route, and call the *execInline* function as the request handler". Seemingly, this http abstraction can be used in a variety of applications. Using this specific pattern, for the needs of the debugging process, we install several routes at the backend. All the routes accepts only *POST* methods and *application/json* content type requests. The responses are always in *Json* format and wrapped to a response template message:

```
{ "msg": responseContent }
```

Due to the flexibility of the http protocol, any custom debugger client can be integrated, in any programing language and environment. Essentially, the debugger client should only implement the following methods, in order to communicate with the backend.

Method: `/nextstage`
Parameters: Ø
Response:
```
{ "depth": currentStagedDepth, "srcCode": srcCode, "inlines": totalInlines }
```
Or
```
{ "depth": 0 }
```

The client debugger call this method in order to get the source code of the current staged execution. If no stage remaining, 0 will be given.

Method: `/execinline`
Parameters: The AST JavaScript object that will be assigned as argument to the inline execution. Using this specific pattern.
Response: `{ "inlines": remainingInlines }`

Informs the backend to execute the next inline directive of the current stage, assigning the corresponding argument.

Method: `/inspectast`
Parameters: The AST JavaScript object to be inspected. Using this specific pattern
Response: `true`

Informs the backend that the specific AST object must be inspected.

Method: `/closesession`
Parameters: Ø
Response: `closed`

Notify the backend to finish the current debugging session.

Method: `/syncdbg`
Parameters: Ø
Response:

```
{
  "inlines": remainingInlines,
  "stage": {
    "depth": currentStagedDepth,
    "srcCode": stagedSrcCode
  },
  "inspectAst": ASTJavascriptObject
}
```

This method consists the main informer about the current staged debugging process. It contains useful information related to the remaining Inlines, the current stage depth, the stage source code and if there is any AST object to be inspected.

Figure 13 depicts an example of terminal's log launched by the backend debugger. Firstly, (1) it opens the file that intended to be debugged, it initializes the http server and notifies that it waits for a client debugger to be attached in a user-defined port. During the backend process (2), information verbose the programmer about the routes that the frontend has reached.

In our instance, client has initially request the next JavaScript stage (2). Then, during the stage debugging, client inspects two ASTs. After the inspections, it executes an inline expression (3). After the final inline execution, the staged evaluation at the specific depth has finished. Subsequently, the backend debugger saves (4) the regarding source code in a JavaScript file. Debugger client requests the next stage, therefore, it repeats the debugging loop.

Figure 13 – Backend debugger terminal log on action.

4.2.3. Frontend

Frontend debugger is an independent architectural component, implemented as a web page. The debugger separates the view in two parts and communicates with the server using Ajax requests. The first view, called *Evaluator*, follows the main debugging flow and it exploits the browser's debugging tools. The second view, called *Visualizer*, display all the necessary debugging information regarding the process.

4.2.3.1. Evaluator

Due to the prosperity of JavaScript in a variety of applications, many sophisticated debuggers involved. We grab this advantage, by exporting the staged execution process to the frontend. During the implementation, we came up with many challenges. We will briefly describe the frontend evaluation flow.

- Firstly, we prompt the user to open the browser's inspect element tools, Figure 14 depicts an instance.
- Hereafter, backend sends the source and the *Evaluator* evaluates it in the context of the browser using the *eval* JavaScript function.
- While the *Evaluator* evaluates the source, programmer is able to inspect the staged source, trace the execution flow, add breakpoints and watchers, and obtain the benefits of the JavaScript inspect element tools.

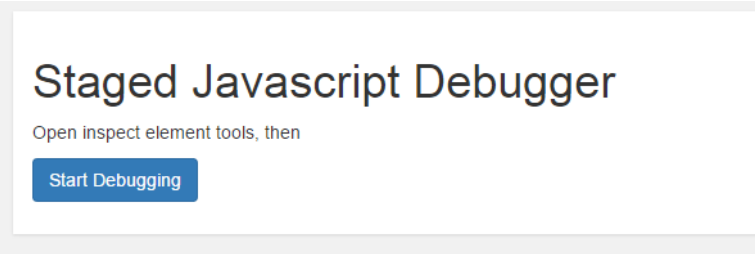

Figure 14 - User prompt to open the inspect element tools.

```
function startDebugging() {

  var onNextStageStart = function(msg) {
    if(msg.stage===0) // was the last stage
      openPage('Finished stage debugging');
    else {
      var src = unescape(msg.src);
      openPage('Debugging stage', msg.stage);
      setOnLoadedPage(function(){
        eval(src); // execute current stage
        startDebugging(); // debug next one
      });
    }
  };

  stageDebugSend({ //backend communication
    route:'NextStage',
```

```
    success: onNextStageStart,
    fail: onNextStageError
  });

  openPage('Start stage debugging', startDebugging);
}
```

As shown, the *Evaluator* execute the following procedure: (i) The *Evaluator* requests to the backend to extract the nest stage, also it sets a local response handler function *onNextStageStart*; (ii) When the *Evaluator* receives the response, it checks if there is a remaining stage *(msg.stage===0)*. In case that there is a remaining stage, it converts the escaped characters from the staged source and evaluates it. In order to trigger the debugger of the browser, the evaluator inserts a debugger; statement at the first line of the staged source. Figure 15 shows an instance. This methodology triggers a breakpoint when an *eval* function invoked. Henceforth, programmer is able to use all the possibilities of the inspect element tools. Figure 16 depicts an example of the debugging process using the Chrome inspect element tools [17].

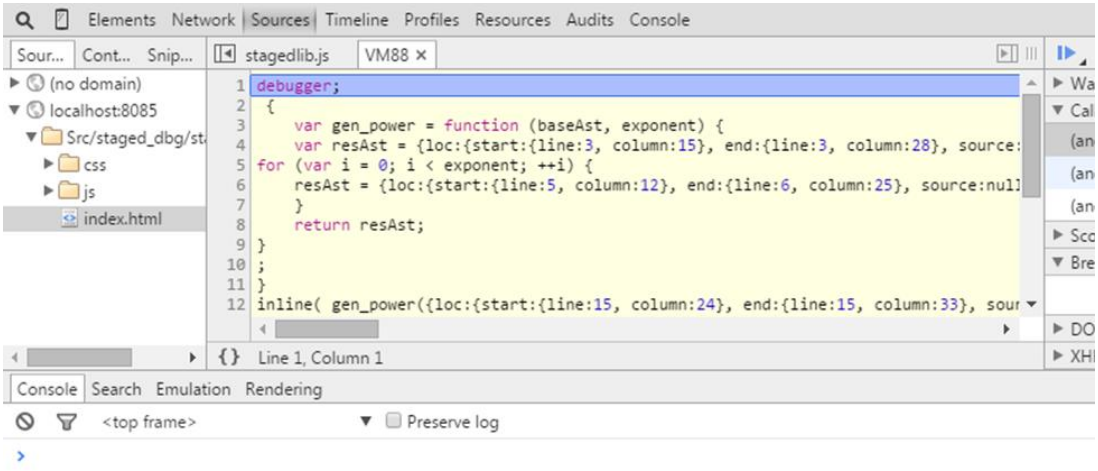

Figure 15 - Debugger evaluator, a 'debugger' statement inserted at the first line to trigger a breakpoint.

Eval function and first-line breakpoint was an issue that came up early during the integration of this procedure. We was searching a way to stop the execution before the evaluation of the staged source. The problem occurs when the *eval* function is triggered. It instantly executions the source, so there is no inspection room for the programmer.

As we mentioned, we resolve this issue by inserting a debugger statement (*debugger;*) in the first line of the staged source code. When the execution flow reach the first line, and programmer has open the inspect element tools, debugger statement behaves as a breakpoint. Hence, the execution stops there and programmer is free to inspect the source.

The debugging evaluation of the staged source take place in the frontend. Frontend execution environment is different than Spidermonkey. As a result, all the global functions of SpiderMonkey along with our additional functions, are totally missing from the JavaScript global object. Consequently, we had to replicate all the functionalities.

We modify the functionality of the inline method by sending a synchronized Ajax request to backend. We force the backend to execute the corresponding inline, assigning the argument of the frontend's inline function. Escape and Escape-value are reproduced in a few lines of pure JavaScript code.

SpiderMonkey global functions offers a variant of functionalities. Some of them are open/write files, evaluate source text, dump execution information, etc. Obviously, several of these functionalities cannot reproduced in the client execution environment. For example, the SpiderMonkey's function *dumpHeap* dump information about the current heap. Browsers cannot gain access to this kind of data, mostly for security reasons. In case of these functions, a *NotSupposedException* thrown. Admittedly, after the study cases, we conclude that some SpiderMonkey functionalities like open/write files are very useful and commonly used in the staged process. We simulate this functionality by sending a synchronous Ajax request to backend requesting to open/write a file in the corresponding filepath.

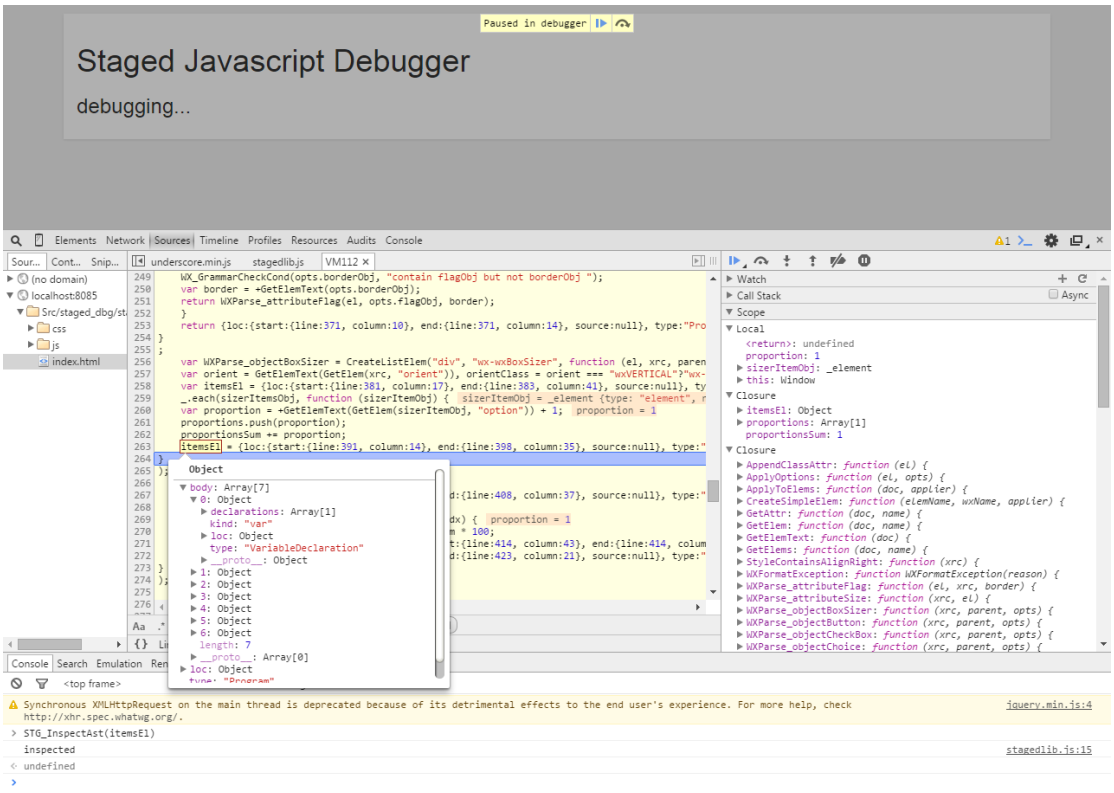

Figure 16 - Debugger evaluator during the execution.

4.2.3.2. Visualizer

Visualizer is responsible for visualizing the debug state. Those information assists the programmer to debug the process. Visualizer depicts the current staged depth, the total and remaining quantity of inline functions, a read-only view of the staged source text and the most important feature, an interactive tree view of AST JavaScript object. The initial state of debugger visualizer shown in Figure 17.

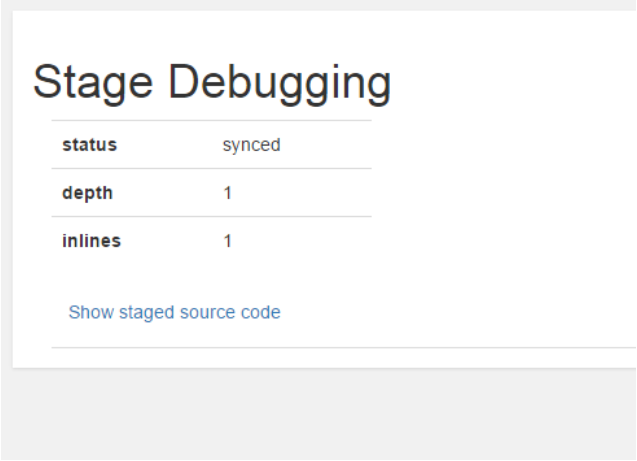

Figure 17 - Debugger visualizer initial state.

Interactive AST Inspection View

The staged process mainly based on AST creation and composition. Thus, a common requirement during the integration of a metaprogram is a helpful AST inspection. Programmer must be able to traverse the AST nodes and inspect the enclosed values in a clear and easy inspection interface. A further step to the AST inspection process could be the ability to instantly display the source text of the corresponding AST node.

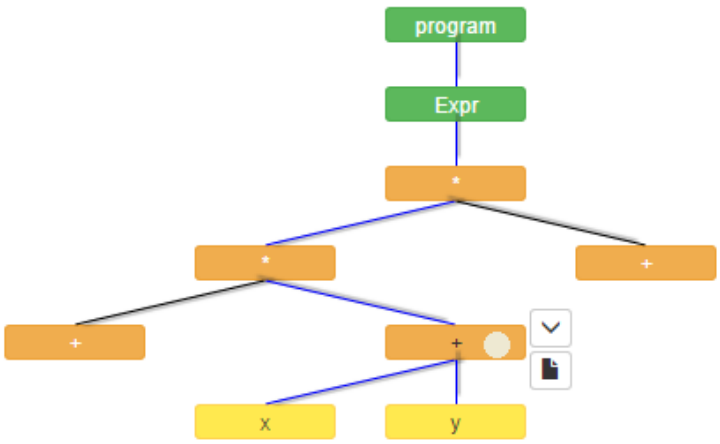

Figure 18 – Interactive AST inspection view.

Our interactive AST view tries to assist the programmer by providing all the essential AST inspection assets. Figure 18 depicts an instance of the view. A tree view represents the AST structure. We separate the node types by marking them with a different color. Statements are painted green, expressions are orange, literal values are yellow, variable declarations are blue, and lists are white. By painting each node a different color, programmer can instantly separate the inner AST structure. The initial view state of the tree contains only the root node. When user hovers the cursor to the node, an option menu appears. In Figure 19, screenshots (1), (2) shows the option menu. There are two options. The first option provides the collapse functionality. When this functionality triggers, the parent nodes appear as children of the target node. After this, the button is toggled in order to provide the reserved functionality. The second button provides the unparse AST functionality. When programmer trigger it, a popup window appears containing the corresponding source text. Figure 19, screenshot 3 depicts an example of this functionality. This functionality is very useful during inspection process. It prevents the programmer from manually interpreting each node of the AST. When the programmer hovers the cursor to a collapsed node, a blue line shown the path from the root to the target node, as shown at (1) in Figure 19.

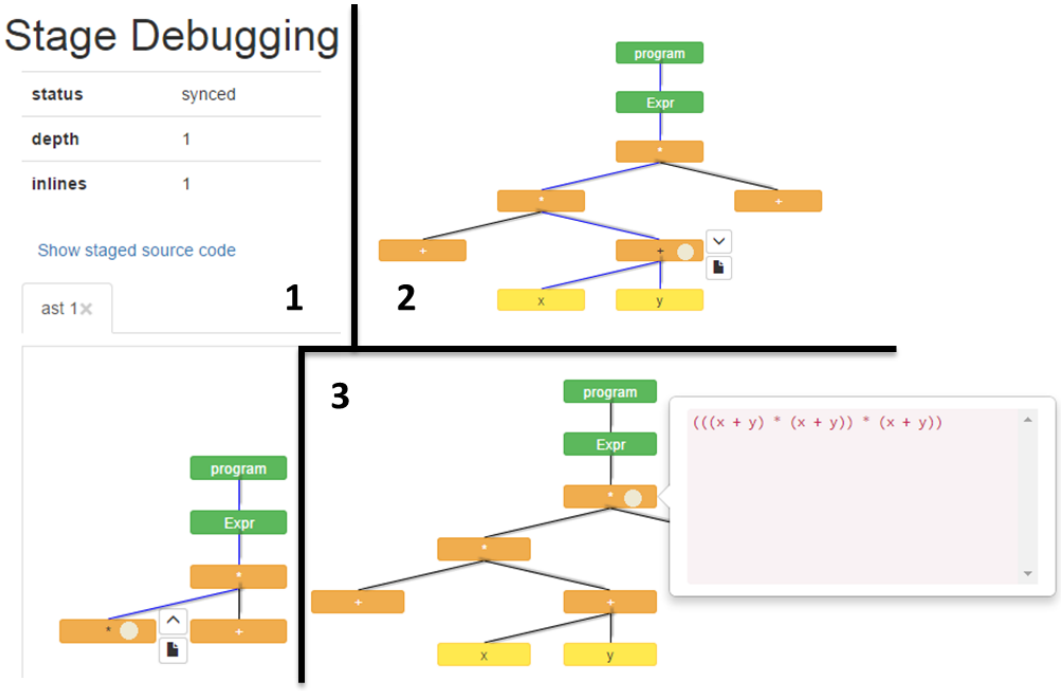

Figure 19 - Interactive AST inspection instances.

We additionally offer a variety of AST inspection view options. Figure 20 depicts the options. User can customize the height of the view, the width and height of each node, the width and height gap of each node, the edge line weight, the maximum length of the name of the node (after that limit it collapsed and it is visible only by hovering the name) and the view padding. Furthermore, we support multiple AST view tabs. This functionality provides the flexibility to explore across the inspected AST views.

Figure 20 - Debugger visualizer options

Chapter 5

Case Studies and Practices

The core purpose besides metaprogramming is the ability to archive code reusability in a longstanding time scale. As a side effect of this, metaprogramming can be used to bootstrap the JavaScript execution time and generate enormous performance improvements. We prove the JavaScript metaprogramming principles by:

- Demonstrate improvements on commonly used JavaScript libraries.
- Create design pattern generators in order to cover missing JavaScript features.
- Exploit the principles of code generation by automating the model-driven process.

5.1. Code Snippets

We refactor code segments from the real world, using metaprogramming, aiming that JavaScript metaprogramming can be meaningful used in commonly used libraries.

Underscore.js

Underscore.js [29] is a popular JavaScript library that provides a whole collection of useful functional programming helpers. While we was inspecting the source, we discover that two ordinary functions was quite similar. _.*max* returns the maximum, while _.*min* returns the minimum element of a collection. For performance reasons, these functions are separated, replicating the essential algorithm logic. We consolidate them using a min-max generator. We figure an excerpt of _.*max* functionality logic.

```
_.max = function(obj, iteratee, context) {
  var result = -Infinity, lastComputed = -Infinity, ...;
  ...
    _.each(obj, function(value, index, list) {
      if (computed > lastComputed
        || computed === -Infinity && result === -Infinity) {
         result = value;
         lastComputed = computed;
      }
    });
  }
  return result;
}
```

We observe that there are two differences between min and max functions.

Max	Min
-Infinity	Infinity
computed > lastComputed	computed < lastComputed

Table 2 - Differences between min and max functions.

We create a single function that generate min or max function according to a boolean argument. This function preserves the algorithm logic and modifies only the segments that are responsible for the min/max calculation (Table 2). Hence, we have a single concrete function that encloses the algorithm logic. As a result, when we refract or improve this function, the modification will affect both min and max functions.

```
function genMinMax( sign ) {
```

```
  if(sign){
    objCompAst = .< computed < lastComputed; >.;
    infCompAst = .< Infinity; >.;
  }else{
    objCompAst = .< computed > lastComputed; >.;
    infCompAst = .< -Infinity; >.;
  }
  return .<
    function(obj, iteratee, context) {
      var result = .~infCompAst, lastComputed = .~infCompAst...;
      ...
        _.each(obj, function(value, index, list) {
          if (.~objCompAst
            || computed === .~infCompAst && result === .~infCompAst) {
              result = value;
              lastComputed = computed;
          }
        });
      }
      return result;
    }
  >.;
}
```

Backbone.js

Backbone.js [30] is a frontend framework for the web, giving structure to the MVC model. Models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling are some of the API conveniences that Backbone.js provides to the programmer. The specific framework has many repetitions in the source, mostly to accelerate the runtime performance. For instance, we excerpt the following function:

```
var triggerEvents = function(events, args) {
  var ev, i = -1, l = events.length, a1 = args[0], a2 = args[1], a3 = args[2];
  switch (args.length) {
    case 0: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx); return;
    case 1: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1); return;
    case 2: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1, a2); return;
    case 3: while (++i < l)
      (ev = events[i]).callback.call(ev.ctx, a1, a2, a3); return;
    default: while (++i < l)
      (ev = events[i]).callback.apply(ev.ctx, args); return;
  }
```

```
};
```

This function is an optimized internal dispatch function for triggering events, it tries to speedy the usual cases. In several JavaScript libraries we meet many functions that repeat source segments to improve the performance. We aim to recognize the repetition pattern, in order to generate it automatically. This abstraction provides many advantages to the programmer. It helps the programmer to develop only the significant segments of the source. This way, it prevents the boilerplate repetition. Additionally, it gives the ability to replicate the pattern as many times as required.

```
function genMultiParameterOptimizer(argumentsLength){
  var optimizedAst = .<>.;
  var casesArgsAst = .< ev.ctx >.;
  for(var i=1; i<=argumentsLength; ++i){
    var caseArgsAst = .< .~casesArgsAst, args[.@i] >.;
    optimizedAst = .< .~optimizedAst;
      case .@i: event.callback.call(.~caseArgsAst); return;
    >.;
  }
  return .<
    (function(event, args) {
      var ev, i = -1;
      switch (args.length) { imparsial
        case 0: event.callback.call(ev.ctx); return;
        .~casesArgsAst;
        default: event.callback.apply(ev.ctx, args); return;
      }
    });
  >.;
}
```

We create a function that generates the switch case pattern in the *triggerEvents* as many times as the function's argument. Henceforth, programmer can easily append methods for function acceleration, by calling meta-functions like `genMultiParameterOptimizer`.

5.2. Design Pattern Generators

Design patterns are defined as general reusable solutions to commonly occurred problems within a given context in software design. It is not a finished design that can be transformed directly into source code. It is a recipe for how to solve a specific problem that can be adapted to a variety of applications. Specifically, it is a set of strictly finite rules that helps the programmer to solve and tactically integrate common software design issues. Typically, it appears in cases that programming language cannot provide the appropriate tools and facilities. Design patterns accelerate the development process by providing tested and proven development paradigms. They prevents the appearance of problems that will only be visible when it is too late during the development process, effecting dramatically the source entropy. Most of the Design patterns are programming language independent, meaning that patterns is a guide that could be adapted in the requirements of the corresponding language.

In our work, we aim to create design pattern generators. We abstract and shift to metaprogramming commonly used design patterns in JavaScript. Consequently, we detach from the programmer, the tedious routine of reproducing the boilerplate code of the corresponding design pattern. Consequence, programmer can exclusively fill the content of the design pattern template.

5.2.1. Memoized

Memoized is an optimization technique used primarily to speed up the runtime performance. It stores the results of expensive function calls (called content function) and returns the cached result when the same input repeated. Obviously, the external environment must not affect the content function. So that, the same arguments always resulting the same result.

```
function genMemoized(funDef){
  return .<
    (function() {
      var funcMemoized = function() {
```

```
        var result, cacheKey =
          JSON.stringify(Array.prototype.slice.call(arguments));

        if (!funcMemoized.cache[cacheKey]) {
          result = (.~funDef).apply(null, arguments)
          funcMemoized.cache[cacheKey] = result;
        }
        return funcMemoized.cache[cacheKey];
      };
      funcMemoized.cache = {};
      return funcMemoized;
    })();
  >.;
}
```

We create a function that takes the AST of the function that intended to be memoized as argument and yields the AST of the content function, enclosed by the memoized design pattern. Thereafter, we can reuse the memoized pattern by inlining the metaprogramming *genMemoized* function. For example, after the execution of

```
cos = .!memoized(.< function(x){ return Math.cos(x); }; >.);
```

cosine function will remember the previous results.

5.2.2. Singleton

Singleton is one of the most commonly used design pattern. Singleton guarantees the solely instantiation of a class or object. This is useful when exactly one object is needed to coordinate actions across the system. Singleton assurance that a specific class has only a single instance that is globally accessed.

```
function genSingleton(funDef){
  return .<
    (function() {
      var instance;
      myclass = function() {
        if (instance) {
          return instance;
        }
        instance = this;
        (.~funDef).apply(this, arguments);
      };
      return myclass;
    }());
  >.;
```

```
}
```

We create the *genSingleton* function that takes the function AST that we need to retain one instance as argument, and yields the singleton wrapped version. Consequently, an individual instance will always be given, even if we try to create a new instance using the *new* keyword.

```
var myclass = .!genSingleton(
    function(name) {
      this.name = name;
      print(name);
    }
  );

var class1 = new myclass('class1');
var class2 = new myclass('class2');

>class1 // printed from new myclass('class1')
>class1 // printed from new myclass('class2')
```

5.2.3. Revealing Module

This pattern creates public and private methods. Revealing module has been engineered as a way to ensure that all methods and variables kept private until they exposed as public. A frequency technique under the implementation of revealing module, is the creation of an object that publishes the required closure variables. Revealing frustrated by the fact that we are condemned to repeat the name of the closured variable, in order to publish it as public. Bellow we see an example of the revealing pattern. We observe that we have to repeat the variables that intended to publicize (*distance, init*):

```
(function(){
  var x,y;

  var init = function(_x, _y){
    x = _x; y = _y;
  };

  var distance = function(){
    return Math.sqrt(x*x + y*y);
  };

  return {
    distance: distance,
    init: init
```

```
  };
})();
```

We create a revealing module generator that takes a list of public and private methods and orchestrate the revealing logic. Hence, programmer is only responsible for the module functionality, leaving the technical implementation of revealing to the generator.

```
function genClass(mdl) {
  function getName( attr ){
    return attr.type === 'variableDeclaration' ?
    attr.declarations[0].id : attr.id
  }
  var modules = .<>.;
  for(var i=0; i<mdl.private.length; ++i) {
    modules = .< .~modules; .~mdl.private[i]; >.;
  }
  var exposed = .<>.;
  for(var i=0; i<mdl.public.length; ++i) {
    modules = .< .~modules; .~mdl.public[i]; >.;
    var name = getName(mdl.public[i]);
    exposed = .< .~exposed; expObj[.@name] = .@name; >.;
  }
  return .<
    (function(){
      .~modules;
      return (function(){
        var exprObj = {};
        .~exposed;
        return exprObj;
      })();
    });
  >.
};


var point = .!genClass(
  {
    private: [
      .< var x; >.,
      .< var y; >.
    ],
    public: [
      .< function getSum(){ return x + y; }; >.,
      .< function init(_x,_y){ x = _x, y = _y }; >.
    ],
  }
);


point.init(2,3);
print(point.x);
```

```
print(point.getSum());


>undefined
>5
```

5.3. Staged Model-Driven Generators

What is MDE?

The concept of MDE [20] created by the requirements of software development process. The core philosophy under MDE is that we firstly develop a model or view of the system, and then we transform it into a real entity. MDE tries to:

- Maximize the compatibility between systems. Ideally, developers should have an abstract view of the project, secluded from any medium requirements.
- Simplify the process of design. The model design is an individual component inside the software development process. Thus, any member or team of the development process can participate without the knowledge of the entire system.
- Optimize the communications between individual teams that work on the system, leading to better decisions.

Interface builders

In our approach regarding the user interfaces, we define MDE as the utilization of all the necessary tools and processes, in order to solve the problem of automated user interface engineering. The current MDE tools, like *wxFromBuilder* [21] and *eclipse [25] MDE plugin*s, are focused on the process of exporting user interface source code. In our work, we emphasize on these tools. The main idea under these tools is to generate models that are independent from the environment. Those models automatically

transform and export the environment dependent models and the source code. Thenceforth, programmer use the exported source code in the application logic. This process can be achieved with two different methods. The first method is to link the source to the application. The second is to extend the exported source code by filling it with the application logic.

MDE problem

Even the rapid evolution and the sophisticated integration of MDE tools, a fundamental issue remains. The design automation of user interfaces can never cover all the aspects and the requirements of application. In real scenarios, programmer elaborate and modify the exported source code with event handlers, interaction controls and abstract behaviors. Specially, Figure 21 illustrates the pipeline of user interface developing process. After the modification process, the source code suffers from dependencies. Consequently, programmer insert dependencies to the exported source code, unaware of the MDE tool.

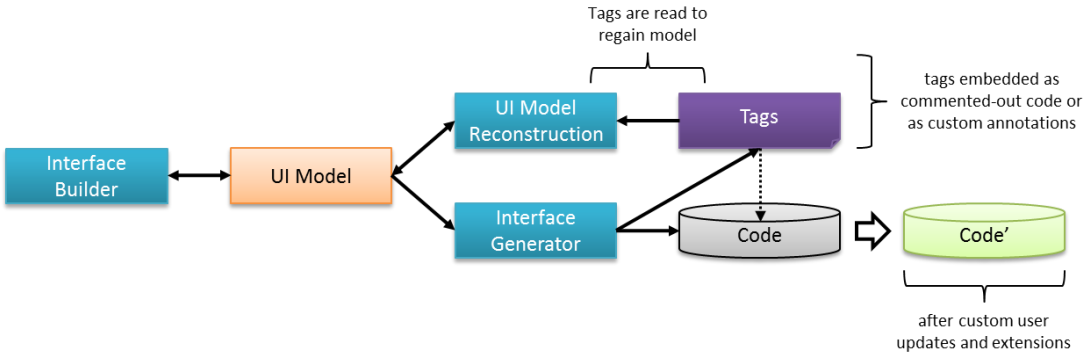

Figure 21 – Pipeline of interactive interface builders involving: 1) interaction with the Interface builder; 2) code generation from an explicit model; 3) tags inserted in the generated source code.

The problem under the above scenario is that it generate maintenance issues. More specific, when user interface code remain unchanged, the exported source code is hygienic and precise. Figure 22 illustrates the process. The typical MDE-based develop process is:

1. Generate the user interface code, as designed from the user interface builder.

2. Embed the user interface code to the application code.

3. During the integration, application code generates dependencies to the user interface code and vice versa.

The maintenance problem appears when the generated user interface code is updated.

- Tag editing and misplacing may break the model reconstruction, while any code that manually inserted outside the MDE tool cause a model-implementation conflict.

- The regenerated source code overwrites the previous source code. Hence, the manual source updates will be lost.

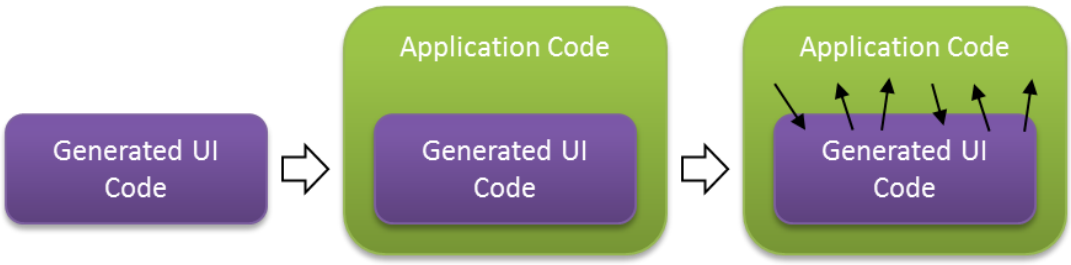

Figure 22 - Common develop process of a MDE-based design.

The MDE reconstruction issue in very critical for a real life application lifecycle. Even the design power and the development conveniences that MDE tools provides, it is difficult to survive in a large scale project. We solve this problem exploiting metaprogramming. We identify that using staged metaprogramming, we gain the ability to preserve the exported source code of MDE tool untouched, letting the application source code grow independent. By introducing the AST in the user interface build process, programmer can manipulate, process and transform the user interface. Using metaprogramming in the MDE build process, we solve the maintenance problem. Additionally, for first time in the MDE tools, we set code manipulation as a first-class concept and reveals the value of using a metaprogramming language in this context. Our work is based to a primary work

that have been developed using Delta programming language [22], essentially we adapt the basic idea to JavaScript.

The MDE staged metaprogramming process swifts the user interface source code generation at build-time. Specifically, the following steps participates at staged level in order to build the final application:

- MDE tools produce user-interfaces in a user-interface description language (UIDL). Thus, we preserve the MDE tool to remain independent from the programming language of the application.

- During the staged process, the exported user-interface files are taking as input, transformed and generate code in an AST form. Essentially, the AST contains the regarding source code of creating user interface elements in a tree structure form.

- Programmer can elaborate the AST. He can add source code snippets, such as event handlers, and customize the exported source code at will.

- Insert the source fragments in the required positions in order to produce the final application.

Litemail, an email client

Our case study inspired from a real-life application. We aim to bring to surface the flexibility and the powerful possibilities on developing using metaprogramming. Our application content is an email client, called *Litemail*. Figure 23 depicts the main application view. Email clients contain many challenging parts. While we was inspecting many elementary email clients, we came up with many interesting components like multiple view segments, input fields, list of items and menus. We choose this example to saw that even the large demanding of the application, we easily address all the parts of the development process.

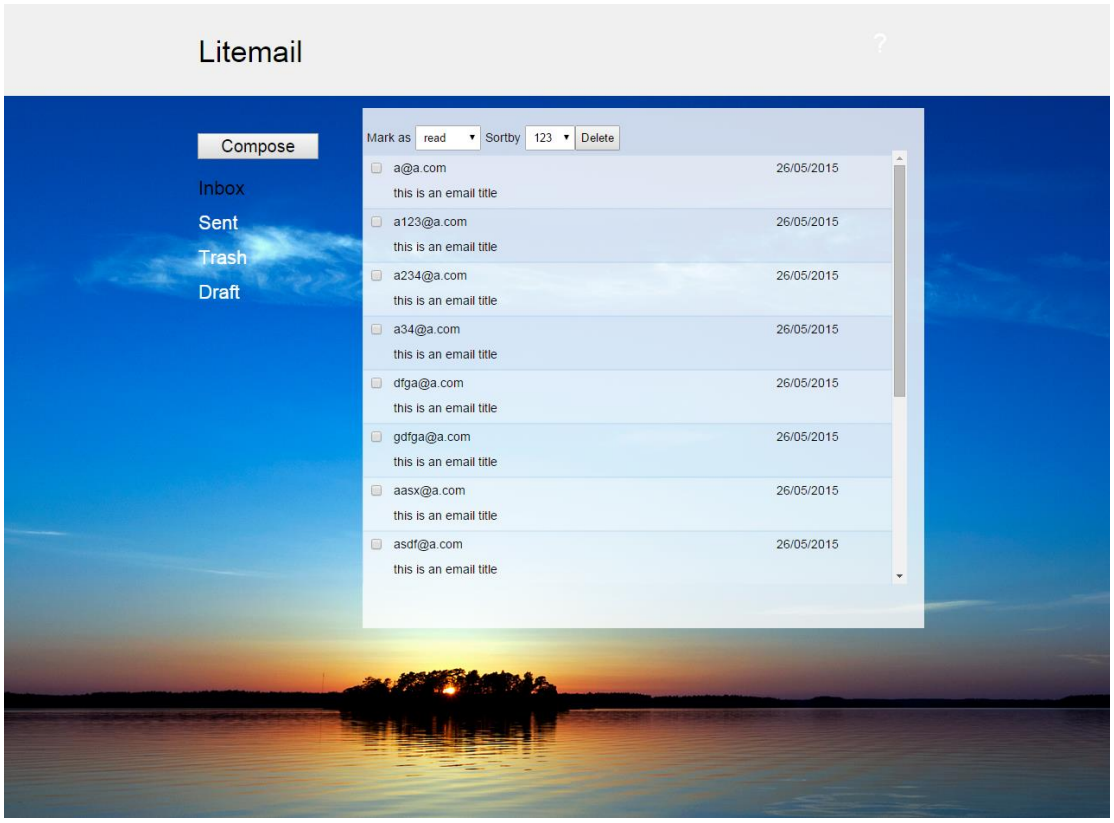

Figure 23- Litemail landing page.

We adopt the MDE authoring tool WxFormBuilder [21], for the requirements of the application. WxFormBuilder is an open source GUI designer application, written in C++, which allows creating cross-platform applications. WxFormBuilder is the corresponding MDE tool that we use to design our application. The user-interface description language (UIDL) that WxFromBuilder uses is the XRC, a XML-like markup language. Our initial step in the development process is to use the WxFormBuilder to design the email client user interface and export it in XRC format, as shown in Figure 24. We separate each email client view in an individual WxFormBuilder project. The significant view elements marked by a representative unique name, simplifying the element selection in the latter steps.

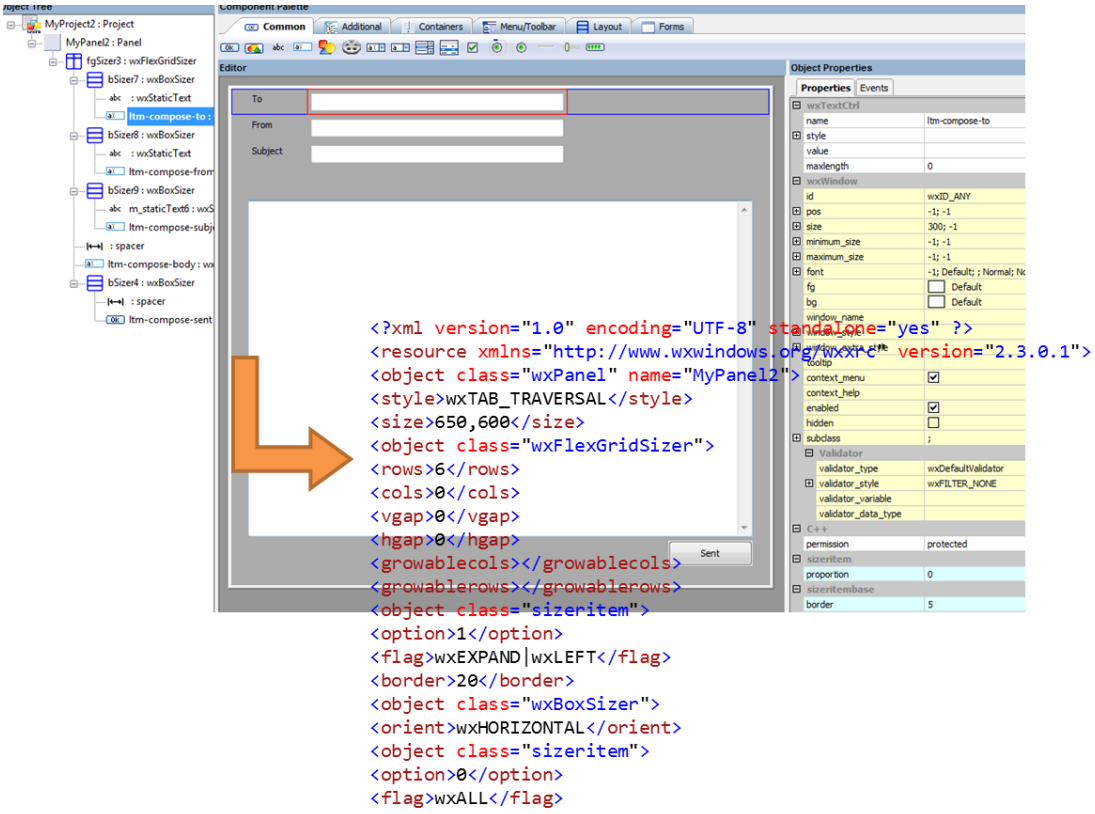

Figure 24 - WxFormBuilder adopted as user interface tool. The exported UI model defined in XRC format.

The next steps takes place during the staged process. Each XRC file parsed as XML source text, so a JavaScript object produced encompassing the XML structure. Afterwards, the XRC to AST transformation process begins. Each XRC UI element transformed to the corresponding AST that represents the creation of an element using the JavaScript DOM API. Considering the application of the case study, we adopt a subset from the original XRC format. Mainly, we choose elements that are commonly and frequently used in related applications. Figure 25 mentions the grammar of the subset XRC.

```
resource ->
object[class="wxPanel"] -> size, ?Sizer

object[class="wxFlexGridSizer"] -> rows, cols, *object[class="sizeritem"]

object[class="sizeritem"] -> flag, border, {1}sizeritem

object[class="wxBoxSizer"] -> wxVERTICAL, *object[class="sizeritem"]

object[class="wxStaticText"] -> label

object[class="wxChoice"] -> content

object[class="wxButton"] -> label, size

object[class="wxCheckBox"] -> size, label

object[class="wxListBox"] -> size

object[class="wxHyperlinkCtrl"] -> label, url

object[class="wxTextCtrl"] -> size, value, maxlength

content -> *item

Sizer -> object[class="wxFlexGridSizer"]
    | object[class="wxBoxSizer"]

sizeritem -> object[class="wxPanel"]
      | object[class="wxBoxSizer"]
      | object[class="wxStaticText"]
      | object[class="wxChoice"]
      | object[class="wxButton"]
      | object[class="wxCheckBox"]
      | object[class="wxListBox"]
      | object[class="wxHyperlinkCtrl"]
      | object[class="wxTextCtrl"]

item -> STRING

border -> NUMBER

size -> NUMBER,NUMBER

orient -> wxVERTICAL|wxHORIZONTAL

flag -> wxALIGN_BOTTOM
    |wxALIGN_CENTER
    |wxALIGN_CENTER_HORIZONTAL
    |wxALIGN_CENTER_VERTICAL
    |wxALIGN_LEFT
    |wxALIGN_RIGHT
    |wxALIGN_TOP
    |wxALL
    |wxBOTTOM
    |wxEXPAND
    |wxFIXED_MINSIZE
    |wxLEFT
    |wxRIGHT
    |wxSHAPED
    |wxTOP
```

Figure 25 - The subset of XRC format.

Each AST element composed in order to follow the same tree structure of the XRC. Addressing the WxFromBuilder style definitions, we create CSS rules respectively. Table

3 shows the transformations occurred from XRC to CSS. The transformation procedure separates in 3 cases,

I. Direct map a single XRC to a single CSS rule (wxLEFT->padding-left)

II. Single XRC rule to multiple CSS rules (size->width, height)

III. Single XRC rule to DOM elements, in case that there is no CSS rule to simulate the XRC functionality. Thus DOM elements combined in order to simulate it.

XRC	CSS
wxALIGN_RIGHT	float:right
wxALL	margin: 5px
wxEXPAND	display: block
wxALIGN_CENTER_HORIZONTAL	text-align: center
Size x,y	width:x px, height: y px

Table 3 - XRC to CSS mapping rules.

Afterwards, when the AST has prepared and all the necessary handlers has registered, programmer can customize the views according to the needs. For example, a very common pattern is a UI for a list of items. In this type of view, user can see a list of items and scroll through them. Each item automatically inserted to the list using an adapter that pulls content from the source. In our case, following the common pattern, designer creates two views. The first view regarding the visualization of the list and the second view depicts how a single item seems, essentially it play the role of the adapter. Therefore, in our UI assembly process, the AST of a specific item and the AST of the list constitute two individual AST objects. Programmer has the responsibility to compose the ASTs in a way to make possible the creation of a list of items.

```
var FunctionMappings = {
```

```
  "wxPanel"   :      function( xrc, parent, opts ) {
            var child = xrc.GetClild( 'object' );
            return child ? this[child.GetClass()](child, xrc, opts) : .< ; >.;
          },
  "wxButton"  :      function( xrc, parent, opts ) {
            var label = GetElemText( GetElem( xrc, 'label' ) );
            return .<
              ( function( parent ) {
                var divroot = document.createElement( 'button' ) );
                var textNode = document.createTextNode( .@label );
                divroot.appendChild( textNode );
                .~addAttributes( xrc.GetAttributes(), .< divroot; >. );
                parent.appendChild( divroot );
              } )( .~parent );
            >.;
          },
  //more functions creating html UI elements...
}
var XrcToHtml = function( xrc ) {
  return .<
    var divroot = document.createElement( 'div' );
    .~FunctionMappings[xrc.GetClass()]( xrc,  .< divroot; >. );
   >.;
}
var GenerateUIFragment = function( xrcPath ) {
  var xrcStr = ReadFile( xrcPath );
  var xrc = ParseXml( xrcStr );
  return XrcToHtml( xrc );
};
var GenerateAll = function() {
  return .<
    .~GenerateUIFragment('basicLayout.xrc');
    .~GenerateUIFragment('leftMenu.xrc');
    .~GenerateUIFragment('emailList.xrc');
    .~GenerateUIFragment('emailListItem.xrc');
    .~GenerateUIFragment('composeEmail.xrc');
    .~GenerateUIFragment('header.xrc');
  >.;
};
.!GenerateAll();
```

The above source outlines the staged source of *Litemail*. Essentially, the transformation procedure splits in three parts:

I. Load specifications into an array of objects carrying element creation information.

II. A set of generator functions composing ASTs for the respective widget creation code, including statements which apply the visual attributes carried in the element argument.

III. A loop iterating the elements and invoking generators while compositing the final AST that is eventually inlined to flush the respective code inside the program.

The root AST object is inlined in the final application source code. Consequently, the source text results the staged procedure is a batch of enclosure anonymous functions. Each function creates and appends a DOM element, enrich with the corresponding CSS and custom code (e.g. event handlers), to the parent element. The implementation size of the staged generator is about 350 lines. The code size of the generator remains constant, while the XRC size is variable to the size of user-interface. Hence, the generated user-interface code is around 1000 lines. Finally, during the debugging of staged phase of *Litemail*, we realize the crucial role of an essential staged debugger. We briefly depict the outline of Litemail developing cycle in Figure 26.

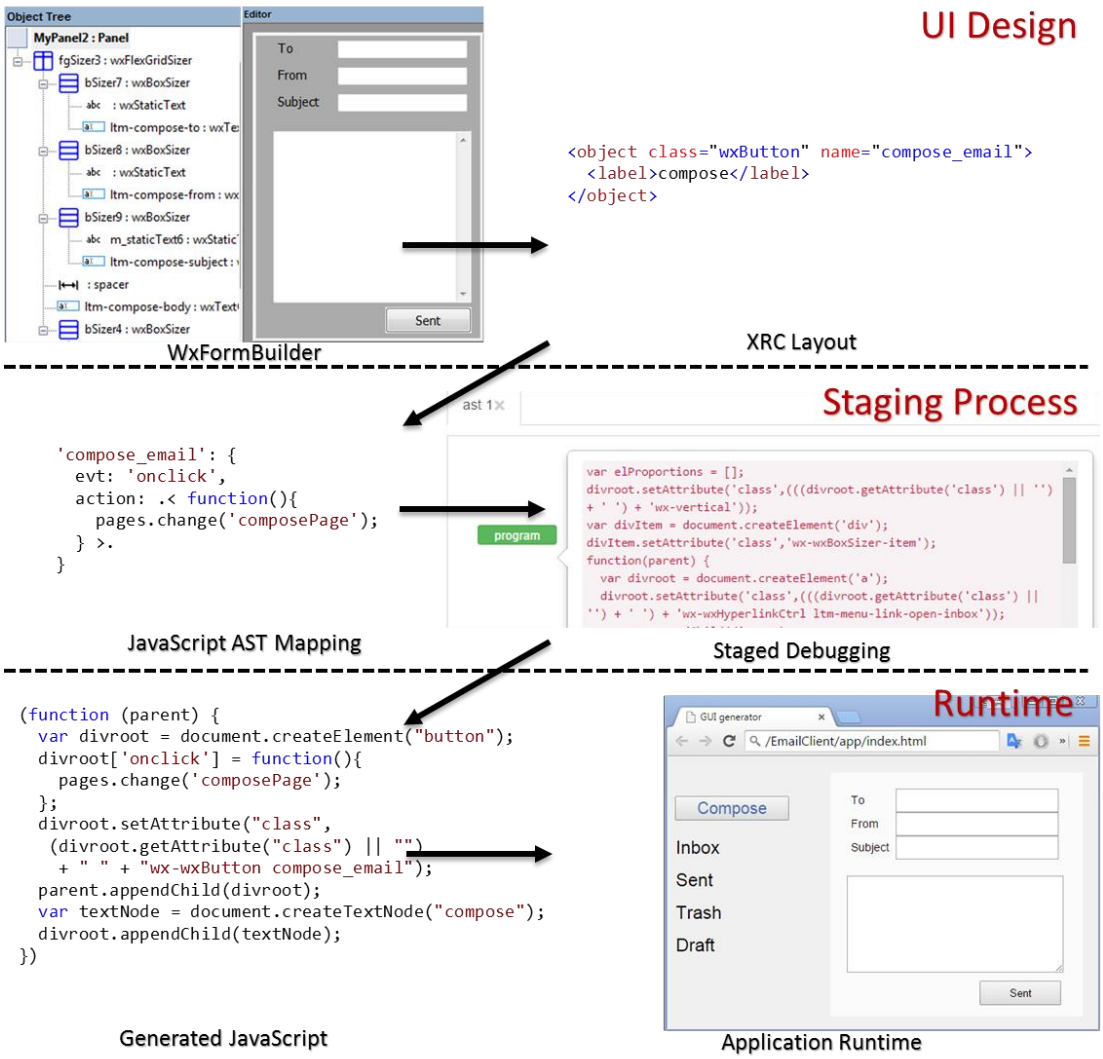

Figure 26 - Outline of Litemail developing lifecycle invlolving UI design, UI model, Javascript mapping process in staged level, debugging process, the generated JavaScript, application runtime at browser.

Chapter 6

Conclusions and Future Work

6.1. Summary

In our work, we implement a multi-staged metaprogramming system in JavaScript. We study the current implementation approaches in multi-staged programming languages. We conclude that the majority of programming languages separate the metaprogram from the normal program. We aim that metaprograms are essentially normal programs, consequently the metaprogramming language should be the same with the normal language. We adopt Spidermonkey JavaScript engine and fully reuse all the components such us lexer, parser, AST, runtime system, and minimally extend them to support multi-staged metaprogramming. We indicate a methodological integration between the metaprogram and normal program, and we argue that the development cost of such integration is minor. Essentially, we demonstrate a metaprogramming model that is at least as expressive as the traditional approaches, but offers significant advantages.

JavaScript debugging systems are innovative and sophisticated. In this context, we export the staged functionality in order to reuse the current debugging facilities. We validate the debugging functionality in well-known web browsers such as Chrome, Mozilla Firefox and Internet Explorer. Furthermore, we assist the debugging process by

creating a collection of proper meta-debugging tools. Those tools provide an AST view mechanism and useful information about the stage process.

We pass the boundaries of small-scale macro-related examples appearing in the literature, as we have accustomed from related work, by implementing a large scale application scenario. We address the MDE maintenance issues and offer a solution that exploits the metaprogramming principals. We implement an email client, called *Litemail*, showing all the stages of the development lifecycle. Additionally, we investigate JavaScript source from the real world, proposing source modifications in the aspects of metaprogramming, aiming to achieve runtime performance and reusability. Furthermore, we formulate the design patterns by implementing design pattern generators. These generators can be used to generate features that are missing from JavaScript or organize the source code.

6.2. Future work

In this Thesis, we focus on the process of creating a coherence Multi-Staged metaprogramming environment. We show that the process of extending a current language in order to achieve metaprogramming is low and manageable. We aim that metaprogramming can be integrated in any language, using thoughtful and tactical extensions. An interesting work could be the integration of multi-staged metaprogramming to other programming languages such as C++. Especially in C++, it would be very interesting to replace the current template language and the preprocessing system.

During the development of the study cases, we came up with many requirements regarding the coding assisting. Those requirements could massively integrated to a powerful IDE, providing meaningful coding utilities. Firstly, a useful functionality is the autocomplete. The problems that came up with the implementation of autocomplete

regarding the nature of JavaScript. Autocomplete is a common issue for untyped languages such as JavaScript, since the type of a variable is not constantly determined, but defined in runtime. Even the most sophisticated JavaScript IDEs have not completely solve the specific issue. Hence, autocomplete is not a future requirement only for metaprogramming assisting, but also for pure JavaScript developing.

Another useful requirement is the interactive editing views. When a programmer is coding in a multi-staged metaprogramming language, the code is logically separated in staging levels. Respectively, the editor view could separate the staging levels. For instance, it could mark each stage with a different background highlighting. Additionally, it would be very useful, to integrate a graphical tool that tries to identify the different transformations and flows of an AST. This tool could identify the creation of an AST and the flow until the embodiment to the final program.

Finally, an improved debugger could add a lot of possibilities to the developing process chain. Extended watchers could support powerful expressions like AST conditions regarding the concatenations and compositions. Furthermore, the enumeration of inlines, escapes and quazi-quotes could instantly provide to the programmer options for breakpoints and proper inspections. We note that the above functionalities could build under our fundamental staged debugging system.

Multi-staged metaprogramming repository

https://github.com/apostolidhs/MetaMonkey [31]

Bibliography

[1] Disney, T., Faubion, N., Herman, H., Flanagan, C. (2014). Sweeten your JavaScript: hygienic macros for ES5. DLS 2014: 35-44.

[2] Brian W. Kernighan and Dennis M. Ritchie. 1988. The C programming language. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition.

[3] J. Rafkind. Syntactic extension for languages with implicitly delimited and infix syntax. PhD thesis, 2013

[4] J. Rafkind and M. Flatt. Honu: Syntactic Extension for Algebraic Notation through Enforestation. Proceedings of the 11th International Conference on Generative Programming and Component Engineering, 2012.

[5] Sheard, T., Jones, S.P. (2002). Template metaprogramming for Haskell. SIGPLAN Not. 37, 12, pp. 60-75

[6] Laurence Tratt. 2005. Compile-time meta-programming in a dynamically typed OO language. In Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05). ACM, New York, USA, pp. 49-63. Available at: http://doi.acm.org/10.1145/1146841.1146846.

[7] Fleutot, F. (2007). Metalua Manual. http://metalua.luaforge.net/metalua-manual.html

[8] Subramaniam, V. (2013). Programming Groovy 2: Dynamic Productivity for the Java Developer. Pragmatic Bookshelf, ISBN 978-1-93778-530-7.

[9] JetBrains. 2013. IntelliJIDEA - Groovy and Grails. http://www.jetbrainscom/idea/features/groovy_grails.html. Accessed 11/2013.

[10] R. Kent Dybvig. 2009. The Scheme Programming Language (fourth edition). The MIT Press, ISBN 978-0-262-51298-5.

[11] Peter Seibel. 2005. Practical Common Lisp. Apress, ISBN 978-1590592397.

[12] Tim Sheard. 1998. Using MetaML: A Staged Programming Language. In: Advanced Functional Programming. 12-19 September, Braga, Portugal, Springer LNCS 1608, pp. 207-239. Available at: http://dx.doi.org/10.1007/10704973_5.

[13] Hygienic Macro System for JavaScript and Its Light-weight Implementation Framework. ILC 2014, 8th International Lisp Conference. http://dl.acm.org/citation.cfm?id=2635653

[14] Taha, W. (2004). A gentle introduction to multi-stage programming. In Domain-Specific Program Generation, Germany, March 2003, C.Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer LNCS 3016, 30-50

[15] Lilis, Y., Savidis, A. (2015). An integrated implementation framework for compile-time metaprogramming. Softw., Pract. Exper. 45(6): 727-763

[16] Mozilla SpiderMonkey Reflect.parse API https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API

[17] Chrome Developer Tools https://developer.chrome.com/devtools

[18] Firefox Developer Tools https://developer.mozilla.org/en-US/docs/Tools

[19] Libmongoose, an embedded HTTP and Websocket library https://cesanta.com/libmongoose.shtml

[20] Schramm, A., Preussner, A., Heinrich, M., Vogel, L. (2010). Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques. In proceed-ings of MODELS 2010, 13th International Conference on Model Driven

Engineering Languages and Systems Oslo, Norway (October 3-8), Springer LNCS 6394, 271-285

[21] WxFormBuilder (2006). A RAD tool for wx GUIs. http://sourceforge.net/projects/wxformbuilder

[22] Yannis Valsamakis. 2013. Improved Model-Driven Engineering with Staged Code Generators. Master's Thesis.

[23] Abstract syntax tree https://en.wikipedia.org/wiki/Abstract_syntax_tree

[24] ECMAScript http://www.ecma-international.org/publications/standards/Ecma-262.htm

[25] Eclipse IDE https://eclipse.org/

[26] Npm package manager https://www.npmjs.com/

[27] Mozilla SpiderMonkey https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[28] Esprima parser http://esprima.org/

[29] Underscore.js, a JavaScript library that provides functional programming helpers, http://underscorejs.org/

[30] Backbone.js, gives structure to web applications, http://backbonejs.org/

[31] Repository of our work https://github.com/apostolidhs/MetaMonkey