B. Comp. Dissertation

# Automated Meta-Programming to Support High-Performance OCaml Codes

By

Arnold Christopher Koroa

Department of Computer Science

School of Computing

National University of Singapore

2014/2015

B. Comp. Dissertation

# Automated Meta-Programming to Support High-Performance OCaml Codes

By

Arnold Christopher Koroa

Department of Computer Science

School of Computing

National University of Singapore

2014/2015

Project No: H018590

Advisor: Assoc. Prof. Chin Wei Ngan

Deliverables:

Report: 1 Volume

## Abstract

Meta-programs are programs that manipulate other programs. This allows for much opportunity for analysis and optimization and so we want to have tools which allows us to write meta-programs easily. MetaOCaml is one such tool, allowing us to write meta-programs by inserting staging constructs into OCaml code. However, the complexity of manually staging a code base can be a burden to users and we want to simplify this process. In this report, we introduce a system of annotating and preprocessing OCaml codes to enable the automatic staging of OCaml codes with minimum input from the users. The staged OCaml code can then be applied to the original code's static arguments, partially evaluating the original code on the static arguments to produce code which is specialized and optimized to the static arguments.

Subject Descriptors:

- **Software and its engineering~Automatic programming**
- *Software and its engineering~Automated static analysis*
- *Software and its engineering~Source code generation*
- *Software and its engineering~Preprocessors*
- Software and its engineering~Software performance

Keywords

meta-programming, MetaOCaml , automated staging, partial evaluation
program analysis and optimization

Implementation Software:

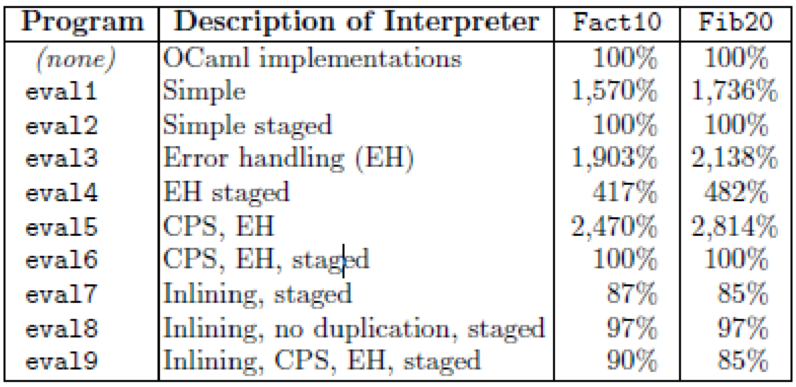
OCaml 4.02.1, BER MetaOCaml N102

Table of Contents

## 1. Introduction

Meta-programs are programs that manipulates other programs. This allows for powerful optimizations by means of static analysis and transformation (Kiselyov, Swadi, & Taha, 2004; Taha, 2004), code specialization by partial evaluation and parameter removal (Carette, 2006; Frigo, 1999; Jonnalagedda, Coppey, Stucki, Rompf, & Odersky, 2014; Taha, 2004), automatic generation of efficient, potentially parallel, low-level codes (Bourgoin & Chailloux, 2014; Herrman, 2005; Langhammer, 2005; Spampinato & Puschel, 2014), and empirical or machine-learning based generation of platform specific code (Puschel, 2011; Puschel et al., 2005; Whaley, Petitet, & Dongarra, 2001). Meta-programming is also useful for compiler generation and domain-specific language (DSL) implementation (Czarnecki, O'Donnell, Striegnitz, & Taha, 2004; Futamura, 1999; Herrmann & Langhammer, 2006).

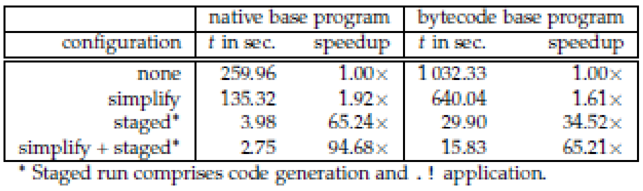
The performance gain achievable through meta-programming can be very significant. For example, Taha (2004) implemented various staged (optimized using meta-programming, this will be explained in section 2) and unstaged DSL interpreters using OCaml and compared their performance in a table reproduced below



| Program | Description of Interpreter | Fact10 | Fib20 |
|---|---|---|---|
| *(none)* | OCaml implementations | 100% | 100% |
| eval1 | Simple | 1,570% | 1,736% |
| eval2 | Simple staged | 100% | 100% |
| eval3 | Error handling (EH) | 1,903% | 2,138% |
| eval4 | EH staged | 417% | 482% |
| eval5 | CPS, EH | 2,470% | 2,814% |
| eval6 | CPS, EH, staged | 100% | 100% |
| eval7 | Inlining, staged | 87% | 85% |
| eval8 | Inlining, no duplication, staged | 97% | 97% |
| eval9 | Inlining, CPS, EH, staged | 90% | 85% |

Here, the various interpreters' (eval1-9) performance are compared with the native OCaml implementation of the same functions. We can see that while the DSL implementations interpreted using standard interpreters (eval2, 3, 5) performs 15 to 20 times slower than the native OCaml implementation, DSL implementations interpreted by staged interpreters (eval2, 3, 6-9) can be as fast as or even faster than the native OCaml implementation.

Herrmann and Langhammer (2006) used the same techniques to implement a DSL using interpretation and compared the performance of the staged and unstaged version of their DSL interpretation. Their results as summarized by Langhammer (2005) is reproduced below

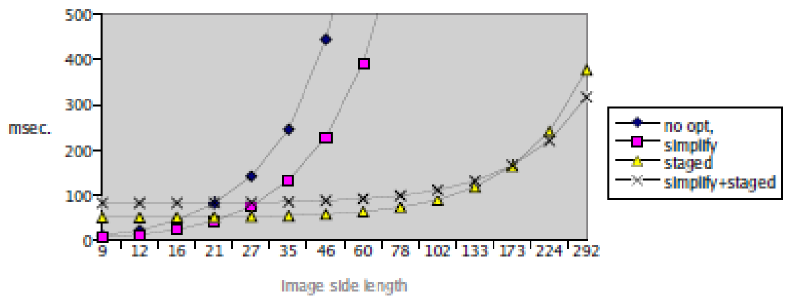


| | native base program | | bytecode base program | |
|---|---|---|---|---|
| configuration | $t$ in sec. | speedup | $t$ in sec. | speedup |
| none | 259.96 | 1.00× | 1 032.33 | 1.00× |
| simplify | 135.32 | 1.92× | 640.04 | 1.61× |
| staged* | 3.98 | 65.24× | 29.90 | 34.52× |
| simplify + staged* | 2.75 | 94.68× | 15.83 | 65.21× |

* Staged run comprises code generation and . ! application.

Table 4.2: Overall execution times for input image of size 1000 × 1000.

Native base program:



Bytecode base program:

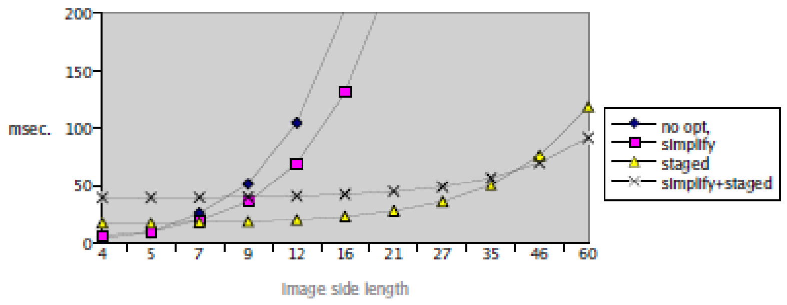
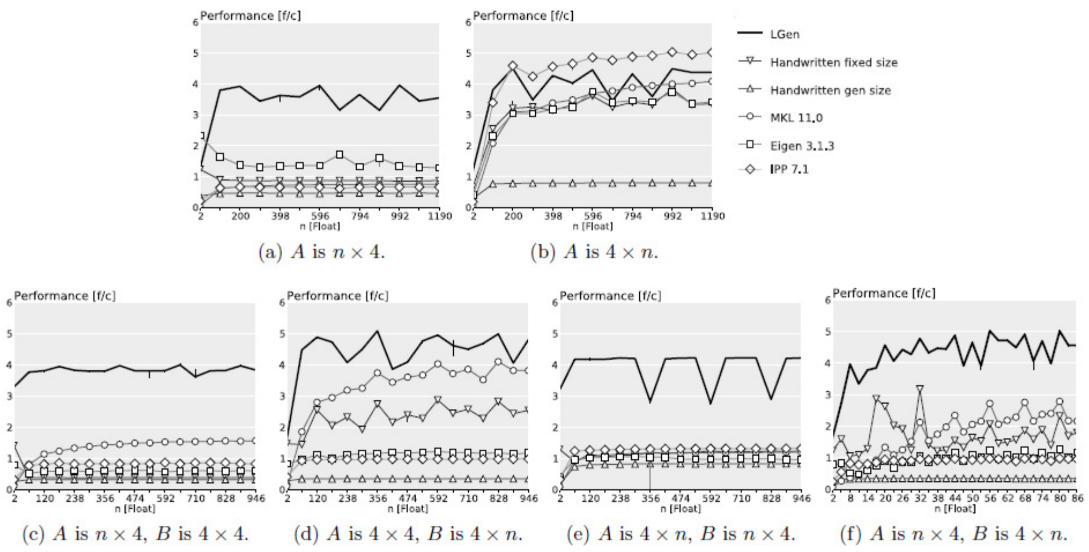


Figure 4.3: Overall execution times for square images of different sizes.

Again, we see that the staged version of the interpreter performs much better than its unstaged counterpart. The two results above are very encouraging as it shows that it is possible to implement a DSL only by using interpretation (instead of by building a custom compiler, which is more complex) while still being able to achieve high performance.

Other than in the area of DSL implementation, the performance gain obtainable in the area of scientific and mathematical computations are also significant. In this area, meta-programming enables us to build generic linear algebra calculators which can automatically adapt to the problem it is solving and the platform it is running in while still being competitive or even better performance-wise than hand-optimized codes for the same problems and platforms. This can reduce the programmer effort required to hand-optimize the calculators for the different types of problems and platforms.

For example, Spampinato and Puschel (2014)'s results is reproduced below



(a) $A$ is $n \times 4$. (b) $A$ is $4 \times n$.

(c) $A$ is $n \times 4$, $B$ is $4 \times 4$. (d) $A$ is $4 \times 4$, $B$ is $4 \times n$. (e) $A$ is $4 \times n$, $B$ is $n \times 4$. (f) $A$ is $n \times 4$, $B$ is $4 \times n$.

**Figure 4: Simple BLACs. (a)–(b):** $y = Ax$; **(c)–(f):** $C = AB$.

Here, LGen is the generic linear algebra calculator while IPP, MKL and the handwritten codes are the hand-optimized codes. We can see that LGen performs better than the hand-optimized code in most cases.

Puschel et al. (2005)'s results is also reproduced below

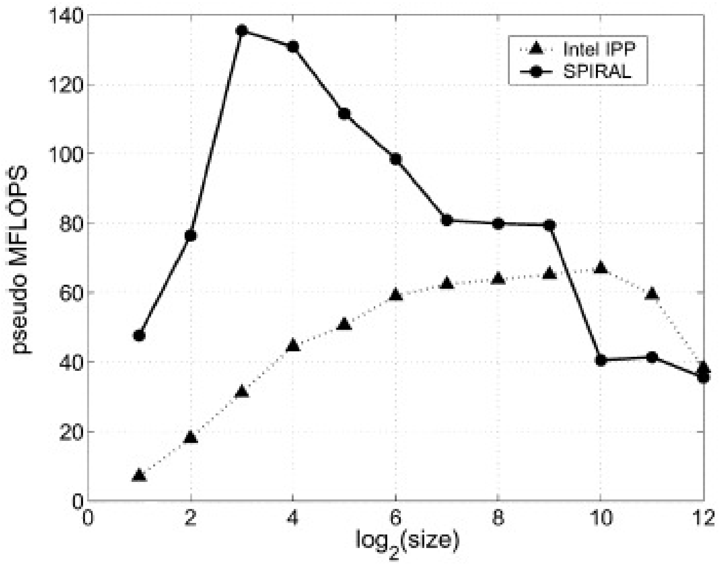


**Fig. 11.** Performance of SPIRAL generated fixed-point DFT code for sizes $2^1$–$2^{12}$, on IPAQ versus Intel IPP 3.0. Platform: ipaq-0.4-lin.

Here, SPIRAL is the generic linear algebra calculator (specific to Discrete Fourier Transform) while IPP is the hand-optimized code. Again, we see that meta-programming allows genericity and performance at the same time.

The many benefits provided by employing meta-programming techniques motivates us to find a suitable tool with which we can develop meta-programs easily. MetaOCaml, a meta-programming dialect of OCaml (Kiselyov, 2010), is one of such tools, allowing us to create meta-programs using a high-level language, OCaml, as its base. However, MetaOCaml requires its users to manually annotate their source code with staging constructs. This requires the users to have a deep understanding of the meaning and possible usages of the constructs before the user can build an efficient multi-staged program. Furthermore, manual annotation of existing code can be quite tedious, which increases the likelihood of human error and careless mistakes.

This project attempts to automate the staging process, enabling users to reap the benefits of multi-staged meta-programming without having to deal with the complexities of manually staging source programs. This is done by writing a preprocessor that receives an annotated OCaml source program as input and outputs a staged MetaOCaml version of the input program. Here, the annotation required is only which functions to stage and what static information does the function have access to; which is in a higher level, and is thus much simpler, than manually inserting staging constructs to the input program.

There are other works that similarly attempted to automate or abstract over the staging process. Light Modular Staging (Rompf & Odersky, 2010) is a technique based on abstract interpretation (Kiselyov et al., 2004) that uses the type system to indicate that a certain code element is a representation (i.e. staged). It is then necessary to provide concrete graph-representation to model the code elements and define operators that works on these code representations. Lastly, a code generator is required to produce code (printed instruction strings) according to the code representation node. While this approach is powerful as it allows programmers to define operators that traverses and optimizes code graphs before the resulting code is generated, it requires significant effort from the users to model and define all the code elements and operators (even though once written these models and operators can be reused).

Delite (Sujeeth et al., 2014) and Forge (Sujeeth et al., 2013) uses LMS to build systems that allows users to make use of staging without having to deal with most of the groundwork of defining the graph representation and implementing general optimizations. Delite till requires its users to define their own data structures and operations that works on their representation but Forge actually built on that further and only requires its users to provide a declarative specification. These projects, however, are targeted at the implementation of domain-specific languages and is less general.

This report is organized as follows: Section 2 introduces the tools used in the project, elaborating more on multi-staged programming using MetaOCaml, the OCaml constructs used for the automated staging annotations and how the annotations can be processed. Section 3 describes the actual automated staging annotations, how they are used and how they are processed. Section 4 shows some sample results obtainable by using the system and Section 5 concludes the report and points out areas of possible future work.

The source code for the system implemented in this project can be obtained from https://github.com/ackoroa/ppx_toMeta

## 2. Tools

This section describes the tools used in this project. They include MetaOCaml, the platform used for meta-programming in OCaml, and the OCaml extension points and ppx preprocessor used to implement the automatic staging of OCaml codes.

### 2.1. MetaOCaml

One of the main tool used in this project is MetaOCaml, a multi-staged flavor of the OCaml programming language (Kiselyov, 2010). The MetaOCaml version used in this project is BER MetaOCaml N102 which is a derivative from the original MetaML (Taha, 1999) for OCaml version 4.02.1.

MetaOCaml provides three constructs on top of the OCaml programming language to enable multi-staging: *bracket*, *escape* and *run*.

#### 2.1.1. Bracket `.< … >.`

Brackets delay the computation written inside them. In other words, brackets stage the computation it contains, turning the computation into object code fragments. As an example, let's take a staged function `plus2` that adds its integer argument by 2.

```
# let plus2 x = .< x + 2 >.;;
```

Here, the `x + 2` inside the brackets means that the addition will be delayed and not immediately computed. So, if you call the function as below

```
# plus2 3;;
```

You'll get the following code fragment:

```
- : int code = .< 3 + 2 >.
```

Notice that the result is not 5 but instead a code that adds the argument 3 by 2. Also notice that the type of the expression is `int code` instead of `int`, signifying that it is a staged code fragment that produces an integer when it is eventually executed.

Using brackets, we can thus stage a computation and delay its execution to a later stage. It is also possible to nest brackets to produce code that further manipulate other codes (i.e. multi-staged code) which might be useful if we are trying to stage programs which generates other program generators (a compiler-compiler (Futamura, 1999)).

### 2.1.2. Escape .~

Escapes evaluate code pointed by the operator to produce code fragments which is then spliced (inserted) into its surrounding bracket. This implies that escape can only be applied to values of type code or an applied function that returns a value of type code; and that escape can only be placed inside a bracket.

For example, we can apply the `plus2` function above to different arguments and then multiply the results together as below:

```
# .< .~(plus2 3) * .~(plus2 4) >.;;
- : int code = .< (3 + 2) * (4 + 2) >.
```

As can be seen, `plus2` is evaluated to produce code fragments that adds 2 to their respective arguments and then these two fragments are spliced into the bigger code fragment the escapes was located in.

### 2.1.3. Run !. (Runcode.run)

The run construct unstages a code fragment, compiling and executing the code fragment within the bracket. For example, if we put a run operator in front of the previous example as below

```
# !. .< .~(plus2 3) * .~(plus2 4) >.;;
```

We'll get an integer, 30, which is (3+2) * (4+2)

```
- : int = 30
```

The run operator only unstages one delayed computation stage at a time. For example, in the code below the if-else-then in the first stage will be computed but not the additions in the second stage

```
# let x = 1 in !. .< if x = 0 then .< 2 + 3 >. else .< 4 + 5 >. >.
- : int code = .< 4 + 5 >.
```

A quick note, the `!.` operator is syntactic sugar for the method `run` defined in the module `Runcode` and so `open Runcode` needs to be appended in front of the program source for `!.` to be usable. Alternatively `Runcode.run` can be used directly. Also, in older versions of MetaOCaml this operator was written as `.!` (dot in front of exclamation mark).

These three constructs can be used to build meta-programs that create and manipulate other programs. Another important feature of MetaOCaml that is worth noting is the ability to use values and functions defined in the code generator in the generated code. This is called cross-staged persistence (CSP). E.g.

```
# let plus2genLevel = (+) 2;;
val plus2genLevel : int -> int = <fun>
# let plus2to3 = .< plus2genLevel 3 >.;;
val plus2to3 : int code = .< (* CSP plus2genLevel *) 3 >.
# !. plus2to3
- : int = 5
```

What's happening here is we have `plus2genLevel` function from the code generator level used in the generated code level. As the code generator level function is in the form of compiled bytecode, the MetaOCaml system cannot pretty print the function body nor splice it in as code and so it is displayed only as `(* CSP plus2genLevel *)`. Then, when the generated code is run the code from the code generator level and the code from the generated code level is linked accordingly and the `plus2genLevel` function is used. This, however, requires that the generator environment be available to the generated code during execution time – which may be the case if code are generated and run immediately on the fly but may not be the case if the generated code is stored first (e.g. as a new .ml source code file) to be run at a later time.

## 2.2. OCaml Extension Points

OCaml extension points (Frisch, 2013a; Inria, 2014; Zotov, 2014), available from OCaml 4.02.1 onwards, are annotations which can be attached onto an OCaml abstract syntax tree (AST). These annotations can be used to attach extra information which can be used to expand OCaml syntax by processing the information with a ppx preprocessor. The details of the ppx preprocessor will be explained in section 2.3.

Each annotation have an id and a payload. The id is any valid OCaml identifier or capitalized identifier. The payload can be an OCaml module item (which can be empty), an OCaml type expression prefixed with `':'` or an OCaml pattern prefixed with `'?'`. Examples of valid payloads are "`let x = 1`", "`: int -> int`" and "`? (a, b)`". For more details please see Inria (2014).

There are three types of annotations that can be attached to the syntax tree: attributes, extension nodes, and quoted strings.

Attributes is a simple attachment to the OCaml AST which is ignored by the OCaml compiler by default and is written as

```
[@id payload]    or    [@@id payload]    or    [@@@id payload]
```

depending on what OCaml AST node the attribute is to be attached to.

`[@id payload]` is attached to OCaml expressions (expresions, type expressions, module expressions, module type expressions, patterns, class expressions and class type expressions). For example, `let x = a + 3 [@id payload]` attaches the attribute to the expression "`3`" and `let x = (a + 3) [@id payload]` attaches the attribute to the expression "`(a + 3)`"

`[@@id payload]` is attached to OCaml structure items (structure items, signature items, class fields, class type fields and each binding in let declarations). For example `let x = a + 3 [@@id1 payload1] and y = b + 2 [@@id2 payload2]` attaches the attribute `[@@id1 payload1]` to the binding "`x = a + 3`" and the attribute `[@@id2 payload2]` to the binding "`y = b + 2`".

`[@@@id payload]` is a floating attribute that is its own structure item and is not attached to any other structure item. For example, `let x = a + 3 [@@@id payload]` will create two OCaml structure items: one for the let binding "`x = a + 3`" and one for the attribute item `[@@@id payload]` and insert both into the structure item list as separate items.

Extension nodes replaces a valid OCaml structure item and is written as

```
[%id payload]    or    [%%id payload]
```

depending on whether the extension node replaces an OCaml expression or structure item.

Extension nodes are useful if we want to specify a custom variant of an existing syntactic construct (Zotov, 2014). For example, if we want to always perform a certain computation on new variables we declare (e.g. add it with 2) we can write `[%%plus2 let x = 3]` and use the preprocessor to translate it to let `x = 3 + 2`. We can even change the construct's semantics completely this way. For example, we can write `[%%update let x = 3]` and translate it to update an existing reference variable `x` to the value `3` (i.e. translate it to `x := 3`) instead of declaring a new variable `x` with the value `3`.

Quoted strings allows for inserting code with syntax unrelated to OCaml code. This is written as

```
{id| … |id}
```

where the content of the quote can be any string literal. The id and string literal can then be extracted and processed by a ppx preprocessor.

For example, we can implement C-style macros by writing `{define| PI 3.142 |define}`, extracting the string `"PI 3.142"` from it, parsing the string and putting the substitution `PI -> 3.142` into some environment variable, and going through the parsed AST to replace all occurrence of the identifier `PI` with its float value `3.142`.

In this project, only attributes are used as only annotating existing OCaml structure items and expressions are required without having any need for replacing OCaml structures or an expression with a totally new syntax. This annotation using attributes is described in section 3.

## 2.3. ppx Preprocessor

A ppx preprocessor is in essence a function that maps an OCaml AST to another OCaml AST. While the default OCaml parser will parse extension point structures and attach it to the AST, the compiler will ignore attribute nodes, reject extension nodes, and treat quoted strings like a normal string literal by default. It is then necessary to extract and transform them with the preprocessor to get the desired functionality of the extensions.

To facilitate this, the ppx preprocessor is able to access the OCaml AST data structure through the libraries Asttypes (Inria, 1995a) and Parsetree (Inria, 1995c). The library Ast_mapper (Inria, 2013c) provides a default mapper that does a deep identity mapping of OCaml parse tree and it is possible to override parts of this default mapper to implement our own inspection and transformation of the parse tree. Lastly the library Ast_helper (Inria, 2013a) provides methods to conveniently build OCaml ASTs. For more details on their usage see Zotov (2014).

Basically what needs to be done is to write a preprocessor that accepts the parsed annotated OCaml code's AST, override the default mapper at the parts where an annotation might have been placed, check for the annotation, and either output back the original tree if there is no annotation or return the tree of the staged code if there's an annotation. The details of how this is done is described in section 3.

The preprocessor, once written, is then compiled into a binary. The preprocessor binary is then passed into the MetaOCaml compiler using the –ppx flag when compiling our target source code so that the MetaOCaml compiler uses the preprocessor binary on the source code first before compiling the source code. This is done like so

```
metaocamlc –ppx ./ppx_toMeta.native target_source.ml
```

We can also pretty-print the produced code by using the compiler's debugging feature like so

```
metaocamlc -dsource -ppx ./ppx_toMeta.native target_source.ml
```

Putting them all together, an OCaml source code is first annotated by attaching the attribute extension point structure at appropriate places. Then, the source code is processed by a ppx preprocessor and rewritten into a staged MetaOCaml code. The resulting MetaOCaml code can then be compiled using the MetaOCaml compiler or loaded to the MetaOCaml toplevel to be used.

On a side note, other tools for extending OCaml syntax other than the extension points and ppx preprocessor exist such as Camlp5 (de Rauglaudre, 2003) and Camlp4 (Agarwal & Lesourd, 2014). Camlp5 is an older version of Camlp4 that is no longer included in standard OCaml. Camlp4 is the newer version of Camlp5 and is still current to the latest version of OCaml but it lacks documentation and can be hard to learn. While both Camlp5 and Camlp4 is more powerful than the ppx preprocessor as they can arbitrarily extend OCaml syntax, we only need to do simple annotations in this project and therefore does not need to make use of Camlp5 and Camlp4's powerful functionalities. Combined with the problems described above, ppx preprocessor is chosen as the syntax extender tool for this project.

## 3. Automated Staging of OCaml Code

This section describes the implementation of the system of annotation and automated translation of OCaml code to staged MetaOCaml code. The goal of the system is to allow users to simply annotate functions to indicate if it is to be staged and what static information (arguments of the function which always take the same value) is available to it; then have the system automatically stage the computation into a meta-program that takes in the static information and produces code that is partially evaluated on, specialized and optimized to the given static information.

### 3.1. Translation Rules

We first need to come up with translation rules to translate normal OCaml code into staged MetaOCaml code. To do this, we first start by looking at simple functions and how they can be staged depending on the static information available to them.

#### 3.1.1. Simple functions with no control flow or recursion

One of the simplest function we can write is a function plus which takes in two arguments and returns the addition of the two numbers

```
let plus x y = x + y
```

If one of the arguments is static (for example if y is always 2) we may specialize this function as follows

```
let plus x = x + 2
```

what we want is a staged version of plus that takes in the static value of y and produces the specialized code. In MetaOCaml, we can write such a staged code as follows

```
let plus_staged y = .< let plus x = x + y in plus >.
```

Applying the staged function on 2 will give us the desired specialized function

```
# plus_staged 2;;
- : (int -> int) code = .< let plus x = x + 2 in plus >.
```

What happens here is that the value of `y` from the argument of `plus_staged` is inserted into the produced code and we thus have the function `plus` with the argument `y` replaced with its static value of `2`. Also notice that the produced code now only takes 1 argument, `x`, as the second argument is already received, evaluated and specialized away.

From here we get this translation rule for simple function definitions

```
                              <FunBody> => <FunBodyS>
─────────────────────────────────────────────────────────────────────────────────
                         let <FunName> <Args> = <FunBody>
                                     =>toMeta
         let <FunName>_staged <StaticArgs> =
           .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

where `=>`$_{\text{toMeta}}$ is the operation of automatically staging functions from annotations done by the system described in this report and `=>` is the recursive operation of staging an OCaml expression in function bodies.

We will also write a simple default rule for staging function bodies where we simply bracket the whole function body (we will refine this as we explore more complex functions)

```

                          ─────────────────────────────
                           <FunBody> => .< <FunBody> >.
```

Note that when none of the arguments are static, we get the original function

```
let plus_staged = .< let plus x y = x + y in plus >.
```

### 3.1.2. Functions with control flow

Now we take a more complex function with control flow and branching. For example, we can have a function that takes two arguments and adds or substracts the second argument from the first depending on whether the first argument is larger than 0

```
let f x y = if x > 0 then x + y else x - y
```

Staging this function then depends on which argument is static. When x is not static, the best we can do is to stage the function similarly with the simple function case above

```
let f_staged y = .< let f x = if x > 0 then x + y else x – y in f >.
```

This is because x, the information required to know which branch is to be taken is dynamic and thus the branching can't be evaluated until x is known during actual execution. However, if x is static it wouldn't make sense to stage the function as per the simple case as it will lead to a redundant branch like so

```
let f_staged x = .< let f y = if x > 0 then x + y else x – y in f >.
# f_staged 0;;
- : (int -> int) code =
      .< let f y = if 0 > 0 then 0 + y else 0 – y in f >.
```

We can do better than this by evaluating the branching in the generator level first before splicing in only the taken branch into the produced code. One way to do this is to introduce an auxiliary function that does the actual branching and returns only the code fragment of the taken branch. Then, the main body of the code generator calls this auxiliary function using the supplied static value of x while delaying the computation of the dynamic argument y

```
let f_staged x =
  let aux x y = if x > 0 then .< 0 + .~y >. else .< 0 - .~y >.
  in .< let f y = .~(aux x .<y>.) in f >.
```

Using this code now produces a better specialized code

```
# f_staged 0;;
- : (int -> int) code = .< let f y = 0 - y in f >.
# f_staged 1;;
- : (int -> int) code = .< let f y = 1 + y in f >.
```

Note that in the staging of the function above, we need to escape all occurrences of y in the auxiliary function body as we bracketed it in the auxiliary function call to delay its computation, making it into a code fragment which needs to be spliced into the code produced by the auxiliary function body.

It is also possible that control structures are nested. For example, consider the following function:

```
let f x y =
  if x = 0
    then if y = 0 then true else false
    else false
```

If the inner control variable y is static, we can stage the above function as follows

```
let f_staged y =
  let aux x y = if y = 0 then .<true>. else .<false>. in
  .< let f = if x = 0 then .~(aux x y) else false in f >.
```

where only parts of the body is computed in the auxiliary function as only the inner branching is computable statically. If both x and y is static then we can stage the function as follows

```
let f_staged x y =
  let aux_2 x y = if y = 0 then .< true >. else .< false >. in
  let aux_1 x y = if x = 0 then .~(aux_2 x y) else .< false >. in
  .< let f = .~(aux_1 x y) in f >.
```

where both the outer and inner branching is statically computed before the final code is produced.

From these two examples, we can see that we will need to inspect the branches of a control structure even if we cannot compute the branching statically in case there are nested branchings that can actually be computed statically; and we will need to cater for the possibility of the generation of multiple auxiliary functions from the staging of one expression.

Before we write translation rules for control flows in function bodies, we must first revise our previously established staging rules as now we need to take into account for any auxiliary functions that our translation rules might produce in staging the function body. The default translation rule is now written as

$$\frac{}{\texttt{<FunBody> => ([], .< <FunBody> >.)}}$$

where the first empty list term indicates that no auxiliary function is produced by this rule and the second term is the staged function body. The first term is a list to accommodate the fact that the recursive staging of the function body might produce multiple auxiliary functions.

The rule for function definitions is now written as

```
                    <FunBody> => ([<Aux1>, …, <Auxn>], <FunBodyS>)
─────────────────────────────────────────────────────────────────────────────────
                          let <FunName> <Args> = <FunBody>
                                       =>toMeta
 let <FunName>_staged <StaticArgs> =
   let <AuxName1> = <AuxBody1>
   and …
   and <AuxNamen> = <AuxBodyn> in
   .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

We can now write the following translation rules for if-then-else expressions in function bodies:

```
                                isStatic(CondExp)
                     <ThenBody> => (ThenAuxList, <ThenBodyS>)
                     <ElseBody> => (ElseAuxList, <ElseBodyS>)
    ─────────────────────────────────────────────────────────────────────
                   if <CondExp> then <ThenBody> else <ElseBody>
                                         =>
                        ( ThenAuxList @ ElseAuxList @
         [if <CondExp>
            then .< .~(<ThenBodyS[DynArg->.~DynArg]>) >.
            else .< .~(<ElseBodyS[DynArg->.~DynArg]>) >.]
                                         ,
                    .< .~(aux <Args[DynArg->.<DynArg>.]>) >. )
```

```
                       !isStatic(CondExp)
                <CondExp> => (CondAuxList, <CondExpS>)
               <ThenBody> => (ThenAuxList, <ThenBodyS>)
               <ElseBody> => (ElseAuxList, <ElseBodyS>)
  ____________________________________________________________________
             if <CondExp> then <ThenBody> else <ElseBody>
                                 =>
               ( CondAuxList @ ThenAuxList @ ElseAuxList
                                 ,
  .< if .~(<CondExpS>) then .~(<ThenBodyS>) else .~(<ElseBodyS>) >. )
```

where @ is the list concatenation operator and `[x->y]` means the substitution of every occurrence of x with y in the given structure.

The rule for if-then-else can be extended for the match control structure as follows

```
                           isStatic(CondExp)
             <BranchBody₁> => (BranchAuxList₁, <BranchBodyS₁>),
                                 …,
              <BranchBodyₙ> => (BranchAuxListₙ, <BranchBodySₙ>)
________________________________________________________________________
              match <CondExp> with
                <BranchPatt₁> -> <BranchBody₁>
                | …
                | <BranchPattₙ> -> <BranchBodyₙ>
                                 =>
           ( BranchAuxList₁ @ … @ BranchAuxListₙ @
[match <CondExp> with
   <BranchPatt₁> -> .< .~(<BranchBodyS₁[DynArg->.~DynArg]>) >.
   | …
   | <BranchPattₙ> -> .< .~(<BranchBodySₙ[DynArg->.~DynArg]>) >.]
                                  ,
             .< .~(aux <Args[DynArg->.<DynArg>.]>) >. )
```

```
                              !isStatic(CondExp)
                        <CondExp> => (CondAuxList, <CondExpS>)
                 <BranchBody₁> => (BranchAuxList₁, <BranchBodyS₁>),
                                          …,
                 <BranchBodyₙ> => (BranchAuxListₙ, <BranchBodySₙ>)
   ─────────────────────────────────────────────────────────────────────
                 match <CondExp> with
                   <BranchPatt₁> -> <BranchBody₁>
                   | …
                   | <BranchPattₙ> -> <BranchBodyₙ>
                                          =>
      ( CondAuxList @ BranchAuxList₁ @ … @ BranchAuxListₙ
                                          ,
              .< match .~(<CondExpS>) with
                   <BranchPatt₁> -> .~(<BranchBodyS₁>)
                   | …
                   | <BranchPattₙ> -> .~(<BranchBodySₙ>) >. )
```

### 3.1.3. Functions with recursion

We now examine the staging of the power function. The power function can be defined as follows

```
let rec pow x n = if n = 0 then 1 else x * pow x (n - 1)
```

As per previously, if the static information does not include the information required to compute the result of branching in the code, we simply insert the static value into the body, remove the static argument, and stage the rest of the body as is. One difference, however, now we need to substitute the recursive call to pow with the staged version with only the dynamic argument

```
let rec pow_staged x =
  .< let pow n = if n = 0 then 1 else x * pow (n - 1) in pow >.
```

An example of the above staged function being used is shown below

```
# pow_staged 5;;
- : (int -> int) code =
      .< let pow n = if n = 0 then 1 else 5 * pow (n - 1) in pow >.
```

If the static information includes enough information to compute the branching, we can compute the branching using an auxiliary function as before. However, now the recursive call will be in the auxiliary function body and thus needs to recursively call the auxiliary function instead. The recursive call to the auxiliary function also need to be escaped so that it is computed and expanded

```
let rec pow_staged n =
  let rec aux x n =
    if n = 0 then .< 1 >. else .< .~x * .~(aux x n) >.
  in .< let pow x = .~(aux .<x>. n) in pow >.
```

Using the staged function above will result in an expanded code without any recursive call, removing any function call overhead that may result from the recursive calls

```
# pow_staged 3;;
- : (int -> int) code = .< let pow x = (x * (x * (x * 1))) in pow >.
```

Care should be taken, however, for a recursive call in statically computable nested control structures. For example, take the following function

```
let rec nestedBranchRec x y =
  if x = 0
    then (if y = 0 then 2 else nestedBranchRec (x - 1) (y + 1))
    else 1
```

If both x and y are static, then it can be staged as the following

```
let nestedBranchRec_staged x y =
  let rec aux_2 x y =
    if y = 0 then .< 2 >. else aux_1 (x - 1) (.< .~y + 1 >.)
  and aux_1 x y = if x = 0 then aux_2 x y else .< 1 >. in
  .< let nestedBranchRec = .~(aux_16 x y) in nestedBranchRec >.
```

Notice that the recursive call to the original function inside the else branch of the inner if-then-else is substituted with a call to aux_1 (and not aux_2 where it is located in) so that the branching on x will be correctly computed on the recursive call.

Another point to take note is that the argument to the auxiliary function which contains the dynamic arguments to the original function is a delayed computation in the form of a code fragment. We need to ensure that this is maintained even when some operations are performed on the dynamic arguments. This is why y above is escaped into a delayed computation of an addition by one instead of directly adding it by 1 – we cannot add y to 1 during code generation as we don't know the value of y yet.

We can incorporate these observations into our translation rules by writing these translation rules for function applications

```
                 isRecursiveCall()   isInAux()
               <Args> => (ArgsAuxList, <ArgsS>)
          <ArgsS'> = <ArgsS[DynArgS->.<DynArgS>.]>
  ____________________________________________________________
                      <FunName> <Args>
                             =>
      ( ArgsAuxList, .< <FirstAuxName> .~(<ArgsS'>) >. )
```

```
              isRecursiveCall()   !isInAux()
          <DynArgs> => (DynArgsAuxList, <DynArgsS>)
      ________________________________________________
                    <FunName> <Args>
                          =>
      ( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

And a default rule for non-recursive function calls that inspect and stages the function's arguments

```
              <Args> => (ArgsAuxList, <ArgsS>)
      ________________________________________________
                    <FunName> <Args>
                          =>
         ( ArgsAuxList, .< <FunName> .~(<ArgsS>) >. )
```

### 3.1.4. Function that uses other staged functions

Lastly, we will see how to stage functions that uses another (automatically) staged function. As an example, we take a function `ff` that uses the power function as follows

```
let rec ff y m = if y = 0 then 0 else (pow y m) + (ff (y - 1) m)
```

if `m` here is static and we have already previously staged `pow` over its second argument, we can stage `ff` as follows

```
let ff_staged m =
  let pow = !. .< .~(pow_staged m) >. in
  .< let rec ff y =
       if y = 0 then 0 else (pow y) + (ff (y - 1)) in ff >.
```

What's happening here is first we produce the code for `pow` specialized to the static argument `m`, and we compile it so it can be used later on. Then, we stage the function ff as normal while making sure that we calld the now specialized function pow with the correct arguments from the produced code for ff. Running the code above will produce the following results

```
# ffstaged 3;;
- : (int -> int) code =
      .< let rec ff y =
           if y = 0 then 0 else ((* CSP pow *) y) + (ff (y - 1))
         in ff >.
```

Where pow is a cross-stage persistent value as it was produced and compiled into binary in the generator code. One might ask why not create the specialized pow inside the main code body so it can be viewed and packed together with the rest of the produced code like so

```
let ff_staged m =
  .< let pow = .~(pow_staged m) in
     let rec ff y =
       if y = 0 then 0 else (pow y) + (ff (y - 1)) in ff >.

# ffstaged 3;;
- : (int -> int) code =
      .< let pow = (let pow x = (x * (x * (x * 1))) in pow) in
         let rec ff y =
           if y = 0 then 0 else (pow y) + (ff (y - 1))
         in ff >.
```

While it seems more natural to place the pow function together with the main body of the code, we'll run into trouble if the main code body is generated dynamically as in the case if y is static in ff

```
let ff_staged y =
  let rec aux y m =
    if y = 0 then .< 0 >. else .< (pow .~m) + .~(aux (y – 1) m) >.
  in .< let pow = .~(pow_staged y) in
        let ff m = .~(aux y .<m>.) in ff) >.
```

Here, we notice that pow which is declared in the produced code is referenced to in the auxiliary function in the generator code. This means we are actually trying to use something that does not exist yet, which is not possible. We should instead declare and create the function pow first before we use it in the auxiliary function

```
let ff_staged y =
  let pow = !. .< .~(pow_staged y) >. in
  let rec aux y m =
    if y = 0 then .< 0 >. else .< (pow .~m) + .~(aux (y – 1) m) >.
  in .< let ff m = .~(aux y .<m>.) in ff) >.

# ff_staged 2;;
- : (int -> int) code =
      .< let ff m = ((* CSP pow *) m) + ((* CSP pow *) m) + 0
         in ff >.
```

One concern might be that the produced code for the specialized function `ff` then needs the generator environment to be available for it to be used as the pow function resides in the code generator environment. However, if we take it as `ff_staged` will always have to be invoked first to produce the specialized `ff` before it is used, the generator environment will always be available to the specialized `ff`.

To incorporate this to the translation rules, we take it that the $=>_{toMeta}$ operation keeps track of what functions have been staged before and checks if any of them is used by the currently staged function (this is done using annotations, which will be described in section 3.2). Knowing the used staged function, we can then revise our rule for function definitions as follows

```
                    <FunBody> => ([<Aux1>, …, <Auxn>], <FunBodyS>)
 ─────────────────────────────────────────────────────────────────────────────────
                          let <FunName> <Args> = <FunBody>
                                       =>toMeta
let <FunName>_staged <StaticArgs> =
  let <StagedFun> = !. .< .~(<StagedFun>_staged <StaticArgs>) >. in
  let <AuxName1> = <AuxBody1>
  and …
  and <AuxNamen> = <AuxBodyn> in
  .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

Furthermore, we will need to add the following translation rule for function applications

```
                                isPrevStagedFun()
                      <DynArgs> => (DynArgsAuxList, <DynArgsS>)
                 ──────────────────────────────────────────────────
                                <FunName> <Args>
                                       =>
                 ( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

Using the above translation rules, we now know how to transform OCaml code into staged MetaOCaml code given which information is static to the original OCaml code. The full list of final translation rules can be referred to in Appendix A.

## 3.2. Source Code Annotation

We now describe the means by which we indicate which information is static in the OCaml source program.

To indicate which arguments of a function is static, we attach an attribute to the function declaration's binding with id "static" and the list of static arguments as payload. For example, the power function with the argument n static is annotated as follows

```
let rec pow x n =
  if n = 0 then 1 else x * (pow x (n - 1))
[@@static [n]]
```

Here we use the `[@@id payload]` variant of attributes as a let-bound function declaration is classified as a structure item in OCaml.

It is also possible to generate multiple versions of the staged function depending on the different possible static information that may be available to the function. For example, if we want to cater to all the possible scenarios for the power function, we can annotate it as follows

```
 let rec pow x n =
  if n = 0 then 1 else x * (pow x (n - 1))
[@@static []] [@@static [x]] [@@static [n]] [@@static [x;n]]
```

and the preprocessor will generate the staged function of pow for all the different combinations in which the arguments to the function can be static.

For staged functions that uses other staged functions, we will attach an attribute with id "static.use" to the function call to indicate that it should be linked to a previously staged function. For example the `ff` function can be annotated as follows

```
let rec ff x n =
  if x = 0 then 0 else pow x n [@static.use] + ff (x - 1) n
[@@static [n]]
```

Here we use the [@id payload] variant of attributes as the function call is classified as an expression in OCaml. Also, there is no need to specify which argument of the power function is static as we can figure it out by running a static analysis on the two arguments. The preprocessor will then see if a function called `pow` had been staged with the appropriate static arguments previously and uses it if it is found. If no such staged function is found or if the `[@static.use]` annotation is removed from the above code, the produced staged code of `ff` will try to directly use a generator-level function called `pow` instead.

## 3.3. ppx Preprocessor

With the OCaml source code annotated using attributes, we are now ready to process it and generate a staged MetaOCaml code. To do this, we first need to get OCaml to parse our source code, parse the resulting AST to our preprocessor and then analyze and transform the AST accordingly. How to parse the OCaml source code and pass the resulting AST is already described in section 2 so in this section we will focus on the implementation of the actual ppx preprocessor.

This section will deal extensively with the OCaml AST. AST snippets will be used to illustrate how some things can be done but for a clearer overall picture, the full ASTs of the snippets used can be referred to in Appendix B. Details on the OCaml AST and the helper functions available to create them can be found in (Inria, 1995a, 1995c, 2013a).

### 3.3.1. Hooking to the default mapper

By default, the preprocessor will pass the OCaml AST to the default mapper which will do a deep identity mapping of the AST before passing the AST on to the compiler. To be able to perform the required extraction of attributes and generation of MetaOCaml code, we will need to overwrite some parts of the default mapper.

First, we will need to know which part of the default mapper to overwrite. Inspecting the OCaml source code's parse tree, we can find that an OCaml program is a structure containing a list of structure items. From the Ast_mapper API (Inria, 2013c) we see that we can either overwrite the mapper for structure or the mapper for structure_item. As we are going to generate multiple staged version of a particular function and append them together, we will overwrite the mapper for structure and access the list of structure items from there. In the implementation code, this is done as follows

```
let toMeta_mapper argv =
  { default_mapper with
    structure = fun mapper structure_item_list -> … }
```

The next thing to do is to traverse over the structure_item_list to find what we are interested in: function definitions. When we encounter function definitions in the structure list, we will process it to see if it contains the appropriate annotations. If we encounter non-function definitions or if the function definition is not annotated, we can pass the structure item back to the default mapper. In the implementation code it is written as follows

```
match structure_item with
  {pstr_desc = Pstr_value (_, _)} ->
      if hasToMetaAnnot structure_item
        then (* do generation of staged code *)
        else [default_mapper.structure_item mapper structure_item]
  | _ -> [default_mapper.structure_item mapper structure_item]
```

where `{pstr_desc = Pstr_value (_, _)}` is how a function definition is represented in the OCaml AST. A list of structure items is returned as the staged code generator might return multiple items (multiple versions of the staged code).

Lastly, we'll need to register our custom mapper by executing the following

```
let () = register "toMeta" toMeta_mapper
```

### 3.3.2. Extracting information from the annotated function definition

To be able to process the function definitions we need to extract several information from it. Firstly, we need to know whether it is annotated or not and, if it is annotated, what static information can the function expect to have access to (and also the different variations possible). We also need to know other information like the function's name, full argument list and whether or not it is recursive.

Firstly, we'll take a look at how the AST node for function definition looks like:

```
{pstr_desc = Pstr_value (rec_flag, value_binding_list)}
```

Here we have a structure whose descriptor is a `Pstr_value`. The `Pstr_value` in turn have a `rec_flag` that indicates whether the structure defined in the definitions are recursive or not; and the `value_binding _list` which is the actual list of definitions. From here we can get whether or not the function definition is recursive (by checking if the `rec_flag` is `Recursive` or `Nonrecursive`). We then take a closer look at the value binding list:

```
[{pvb_pat = {ppat_desc = Ppat_var {txt = "pow"}};
  pvb_expr = (* function argument list and body *)
  pvb_attributes =
    [({txt = "static"},
      PStr [(* payload structure description *)])]}]
```

The binding list above is taken from the annotated definition of the power function with n marked as static. We see that the value binding contains a `pvb_pat` which contains the function name, `pvb_expr` that contains the function argument list and function body, and `pvb_attributes` that contains the attributes attached to the definition (if any). We can also see that attributes are of the form

```
({txt = id}, PStr [payload])
```

and the payload is of the form

```
{pstr_desc =
   Pstr_eval ({pexp_desc = Pexp_construct ({txt = Lident "::"},
     Some {pexp_desc = Pexp_tuple
       [{pexp_desc = Pexp_ident {txt = Lident "n"}};
        {pexp_desc = Pexp_construct ({txt = Lident "[]"}, None)}]}))}
```

So what we can do is pattern-match down the attribute AST node to extract out the id in the `{txt = id}` structure and also pattern-match down the payload structure to get all the static arguments whenever we see `{pexp_desc = Pexp_ident {txt = Lident argName}}` in the list AST structure.

The function's full argument list can be extracted similarly by pattern-matching down the pvb_expr part of the function definition. We also check if the function uses any previously staged function by checking if there is any function application, Pexp_apply, that has the [@static.use] attribute attached to it.

### 3.3.3. Building the staged function

After we extract out the information we need we then build the staged function itself. For every [@@static …] attribute attached to the function definition, we build a staged version of the function expecting to receive the arguments indicated in the attribute's payload as static information.

The algorithm for building the staged function follows the translation rules we derived in section 3.1 closely. For example, the code for staging if-then-else control structures looks like this:

```
match exp with
  {pexp_desc = Pexp_ifthenelse(condExp, thenExp, elseExpOpt)} ->
      if isStaticExp condExp statVars
        then
          let auxName = fresh "aux" in
          let (auxAux, auxBody) = buildAuxBody …
          in let auxCall = buildAuxCall …
          in (auxAux @ [(auxName, auxBody)], auxCall)
        else
          let (condAux, condExp') = stage condExp in
          let (thenAux, thenExp') = stage thenExp in
          let (elseAux, elseExpOpt') =
            begin match elseExpOpt with
              None -> ([], None)
              | Some elseExp ->
                  let (aux, e) = stage elseExp in
```

```
                (aux, Some (applyEsc e))
            end in
          let body = Exp.ifthenelse ~loc ~attrs (applyEsc condExp')
                       (applyEsc thenExp') elseExpOpt'
          in (condAux @ thenAux @ elseAux, applyBracket body)
```

and buildAuxBody for if-then-else looks like this

```
match exp with
  {pexp_desc = Pexp_ifthenelse(condExp, thenExp, elseExpOpt)} ->
    let (thenAux, thenExp') = buildStagedBody thenExp …
    in let (elseAux, elseExpOpt') =
      begin match elseExpOpt with
        None -> ([], None)
        | Some elseExp ->
            let (aux, e) = buildStagedBody elseExp …
            in (aux, Some e)
      end in
    let body = Exp.ifthenelse ~loc ~attrs
                 condExp thenExp' elseExpOpt'
    in (thenAux @ elseAux, body)
```

As can be seen, if the condition of the if-then-else expression is static, the code simply builds the auxiliary function that computes the branching statically and stages its branches, builds a code fragment that calls the built auxiliary function, and returns the auxiliary function call as the staged function body together with the auxiliary function itself attached to whatever other auxiliary function the staging of the auxiliary function branches might have produced. This is exactly the same as computing all the conditions and recursions of the if-then-else staging rule then applying the prescribed transformation directly.

The main work that needs to be done in building the staged function are the creation of new AST nodes, the substitution of existing nodes in the original function body and the insertion of MetaOCaml staging constructs into the code.

The creation of new AST nodes is made simpler by using the various helper function available in the Ast_helper library. The library contains constructors for all the possible nodes in the AST. An example in the code snippet above would be the `Exp.ifthenelse` constructor for the if-then-else node.

Substituting nodes (for substitution of recursive calls and previously staged function calls) is done as we stage the function body recursively. When we are staging the AST node for function application, we check if it is a recursive call to the original function (by it's name) or if it is a call to a previously staged function (by checking for the `[@static.use]` attribute) and substitute the node and/or the argument nodes attached to it accordingly.

One point to be careful of is that as we traverse and recreate the AST, we'll need to extract information other than the OCaml expressions itself like the nodes' location in the source file and attributes not related to the auto-staging annotations from the original code to ensure that we recreate the surrounding code correctly as we rebuild the AST.

Adding MetaOCaml constructs to the OCaml source code is surprisingly very simple. MetaOCaml brackets and escapes is internally implemented as OCaml attributes. For example, `.<x + y>.` is equivalent to `(x + y) [@metaocaml.bracket]`. Similarly, `.~x` is equivalent to `x [@metaocaml.escape]`. So, all we need to do is add the appropriate `metaocaml` attributes on the AST node of the expressions we want to stage or escape. `!.` is just an application of a function called `"!."` or `"Runcode.run"`.

### 3.3.4. Combining the results

Recall that as we traverse through the structure item list in the OCaml program structure, we produces a list of structure items for every structure item we process as it might be the case that we need to generate multiple staged version of a function definition. Therefore, it is necessary to flatten the now list of list of structure items into a single list before it is returned by the preprocessor. In the implementation code, it is written as the following

```
List.flatten
  (List.map (structure_item_mapper mapper) structure_item_list)
```

where `structure_item_mapper` is the function that process each structure item in the original `structure_item_list`.

## 4. Results

In this section we will show some of the results obtained from running the system on some sample functions. For each sample result, the annotated original OCaml code will be shown first, followed by the automatically staged MetaOCaml code. Some sample runs will then be shown. The letter suffixes on the staged function names shows which arguments are static. The number suffixes are automatically added by MetaOCaml during code generation to avoid name capture.

### 4.1. Simple function without branching nor recursion

Original OCaml code:

```
let plus x n = x + n
[@@static []][@@static [x]][@@static [n]][@@static [x; n]]
```

Staged MetaOCaml code:

```
let plus_ = .< let plus x n = x + n in plus  >.
let plus_x x = .< let plus n = x + n in plus  >.
let plus_n n = .< let plus x = x + n in plus  >.
let plus_xn x n = .< let plus = x + n in plus  >.
```

Sample runs:

```
let _ = plus_x 3
- : (int -> int) code = .<let plus_33 n_32 = 3 + n_32 in plus_33>.

let _ = plus_n 5
- : (int -> int) code = .<let plus_35 x_34 = x_34 + 5 in plus_35>.

let _ = plus_xn 3 5
- : int code = .<let plus_36 = 3 + 5 in plus_36>.
```

## 4.2. Function with both branching and recursion

Original OCaml code:

```
let rec pow x n = if n = 0 then 1 else x * (pow x (n - 1))
[@@static []][@@static [x]][@@static [n]][@@static [x; n]]
```

Staged MetaOCaml code:

```
let pow_ =
  .<let rec pow x n = if n = 0 then 1 else x * (pow x (n - 1))
    in pow  >.

let pow_x x =
  .<let rec pow n = if n = 0 then 1 else x * (pow (n - 1))
    in pow  >.

let pow_n n =
  let rec aux_8 x n =
    if n = 0 then .< 1  >. else .< .~x * .~(aux_8 x (n - 1))  >. in
  .< let pow x = .~(aux_8 .< x  >. n) in pow  >.

let pow_xn x n =
  let rec aux_7 x n =
    if n = 0 then .< 1  >. else .< x * .~(aux_7 x (n - 1))  >. in
  .< let pow = .~(aux_7 x n) in pow  >.
```

Sample runs:

```
let _ = pow_x 3
- : (int -> int) code =
.<let rec pow_62 n_63 =
    if n_63 = 0 then 1 else 3 * (pow_62 (n_63 - 1)) in pow_62>.
```

```
let _ = pow_n 3
- : (int -> int) code =
.<let pow_65 x_64 = x_64 * (x_64 * (x_64 * 1)) in pow_65>.

let _ = pow_xn 3 3
- : int code = .<let pow_66 = 3 * (3 * (3 * 1)) in pow_66>.
```

### 4.3. Function with nested control structures

Original OCaml code:

```
let nestedBranch x y =
  if x = 1
  then match y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy"
  else "_"
  [@@static []][@@static [x]][@@static [y]][@@static [x; y]]
```

Staged MetaOCaml code:

```
let nestedBranch_ =
  .<let nestedBranch x y =
      if x = 1
      then match y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy"
      else "_" in nestedBranch  >.

let nestedBranch_x x =
  let rec aux_4 x y =
    if x = 1
    then .< match .~y with | 0 -> "x0" | 1 -> "x1" | _ -> "xy"  >.
    else .< "_"  >. in
  .< let nestedBranch y = .~(aux_4 x .< y  >.) in nestedBranch  >.
```

```
let nestedBranch_y y =
  let rec aux_3 x y =
    match y with
     | 0 -> .< "x0"  >. | 1 -> .< "x1"  >. | _ -> .< "xy"  >. in
  .<let nestedBranch x =
      if x = 1 then .~(aux_3 .< x  >. y) else "_" in
    nestedBranch  >.

let nestedBranch_xy x y =
  let rec aux_2 x y =
    match y with | 0 -> .< "x0" >. | 1 -> .< "x1" >. | _ -> .< "xy" >.
  and aux_1 x y = if x = 1 then aux_2 x y else .< "_"  >. in
  .< let nestedBranch = .~(aux_1 x y) in nestedBranch  >.
```

<u>Sample runs:</u>

```
let _ = nestedBranch_x 0
- : (int -> string) code =
.<let nestedBranch_5 y_4 = "_" in nestedBranch_5>.

let _ = nestedBranch_x 1
- : (int -> string) code =
.<let nestedBranch_7 y_6 =
    match y_6 with | 0 -> "x0" | 1 -> "x1" | _ -> "xy" in
  nestedBranch_7>.

let _ = nestedBranch_y 0
- : (int -> string) code =
.<let nestedBranch_9 x_8 =
    if x_8 = 1 then "x0" else "_" in nestedBranch_9>.
```

```
let _ = nestedBranch_y 5
- : (int -> string) code =
.<let nestedBranch_11 x_10 =
    if x_10 = 1 then "xy" else "_" in nestedBranch_11>.

let _ = nestedBranch_xy 1 2
- : string code = .<let nestedBranch_12 = "xy" in nestedBranch_12>.
```

### 4.4. Function over lists

Original OCaml code:

```
let rec double xs =
  match xs with
    | [] -> []
    | x::xs -> (2 * x) :: (double xs)
[@@static []][@@static [xs]]
```

Staged MetaOCaml code:

```
let double_ =
  .<let rec double xs =
      match xs with | [] -> [] | x::xs -> (2*x)::(double xs) in
    double  >.

let double_xs xs =
  let rec aux_11 xs =
    match xs with
    | [] -> .< []  >.
    | x::xs -> .< (2 * x) :: .~(aux_11 xs)  >. in
  .< let double = .~(aux_11 xs) in double  >.
```

Sample runs:

```
let _ = double_xs [3;5;9]
- : int list code = .<let double_29 = [2 * 3; 2 * 5; 2 * 9] in double_29>.
```

## 4.5. Function that uses another staged function

Original OCaml code:

```
let rec ff y m =
  if y = 0 then 0 else ((pow y m)[@static.use ]) + (ff (y - 1) m)
[@@static []][@@static [y]][@@static [m]][@@static [y; m]]
```

Staged MetaOCaml code:

```
let ff_ =
  let pow = Runcode.run .< .~pow_  >. in
  .<let rec ff y m =
      if y = 0 then 0 else (pow y m) + (ff (y - 1) m) in
    ff  >.

let ff_y y =
  let pow = Runcode.run .< .~(pow_x y)  >. in
  let rec aux_21 y m =
    if y=0 then .< 0 >. else .< (pow .~m) + .~(aux_21 (y - 1) m) >. in
  .< let ff m = .~(aux_21 y .< m  >.) in ff  >.

let ff_m m =
  let pow = Runcode.run .< .~(pow_n m)  >. in
  .<let rec ff y = if y = 0 then 0 else (pow y) + (ff (y - 1)) in
    ff  >.

let ff_ym y m =
  let pow = Runcode.run .< .~(pow_xn y m)  >. in
  let rec aux_20 y m =
    if y = 0 then .< 0  >. else .< pow + .~(aux_20 (y - 1) m)  >. in
  .< let ff = .~(aux_20 y m) in ff  >.
```

Sample runs:

```
let _ = ff_y 3
- : (int -> int) code =
.<let ff_101 m_100 =
    ((* CSP pow *) m_100) +
      (((* CSP pow *) m_100) + (((* CSP pow *) m_100) + 0)) in
  ff_101>.

let _ = ff_m 3
- : (int -> int) code =
.<let rec ff_104 y_105 =
    if y_105 = 0
    then 0
    else ((* CSP pow *) y_105) + (ff_104 (y_105 - 1)) in
  ff_104>.

let _ = ff_ym 3 3
- : int code = .<let ff_107 = 27 + (27 + (27 + 0)) in ff_107>.
```

From the sample results above we can see how, just by using the annotations supplied, the system is able to stage the original functions into meta-programs that produces partially evaluated, specialized code when it is applied to the static arguments of the original function.

# 5. Conclusions

This section summarizes the results obtained in this project, discusses known limitations of the system and proposes directions for which the project can be brought forward to.

## 5.1. Summary

In this report, we introduced a system of annotating and preprocessing OCaml codes using the OCaml extension points and ppx preprocessor to enable the automatic staging of simple OCaml functions. The staged OCaml functions, when applied to its static arguments, partially evaluates the original function to the static arguments, producing a specialized and optimized code with removed recursive call, branching and parameterization overheads.

The system implemented in this project removes the necessity of manually inserting staging constructs in the meta-programming language MetaOCaml by its users, simplifying the process of staging an existing code base. It also requires less input from the users as compared to available techniques such as abstract interpretation and LMS which requires the users to model the possible code patterns as abstract code nodes. Finally, the system is aimed to be able to produce staged versions of any arbitrary code and is not limited only to certain domains.

## 5.2. Limitations and Recommendation for Future Work

### 5.2.1. Scope of the translation rules

The translation rules used in this project is derived from a few examples of common simple functions and thus does not cover many other programming language constructs and code patterns.

One example of code patterns which the translation rules currently cannot handle is mutating static variables. The current translation scheme completely eliminates static arguments from recursive calls. However, consider the following function:

```
let rec f x n = if n = 0 then 1 else x * (f (x+1) (n-1))
[@@static [x]]
```

This is currently staged into

```
let f_staged x =
  .< let f = if n = 0 then 1 else x * (f (n-1)) in f >.
```

which lost the information that x is incremented for every recursive call and so `(x * (x * …)` is produced instead of `(x * (x + 1) * (x + 2) * … )`

Another example are recursions which involve the mutation of dynamic control variables. For example this function with y static

```
let nestedBranchRec x y =
  if x = 0
    then if y = 0 then 2 else nestedBranchRec (x + 1) (y - 1)
    else 1
```

is currently staged as

```
let nestedBranchRec_y y =
  let rec aux x y =
    if y = 0 then .< 2 >. else aux .< .~x + 1 >. (y - 1) in
  .< let nestedBranchRec x =
        if x = 0 then .~(aux .< x >. y) else 1 in
     nestedBranchRec  >.
```

which lost the information that on the recursive call the value of `x` changed and thus the `if x = 0` branching should be re-evaluated. Instead the produced code only re-evaluates the `if y = 0` branching. With the current translation rules where statically uncomputable branchings stay in the main body, it is unclear how to return the control back to the `if x = 0` branching.

These examples shows some ways in which the current translation rules are still inadequate. A more general and comprehensive translation scheme that covers the complete range of code patterns is a subject for further research.

### 5.2.2. Scope of the ppx preprocessor

Related to the above, the preprocessor's coverage of the OCaml language is also rather limited as it doesn't yet recognize many OCaml language structures and so cannot process them.

One known structure which cannot yet be processed is curried functions as they have no identifier and often is an application of a previous application to further arguments, making the AST for the application non-linear as the applied structure can be tree-like instead of a simple identifier. Anonymous functions also cannot currently be processed.

Similarly, the ppx preprocessor also don't recognize all possible code patterns. For example, a previously staged function can be called multiple times in another function (whether it is in recursive calls of the other function or just simply in separate calls) with a different static argument for each calls. In this case, different versions of the specialized function need to be built from the previously staged function to cater for the different static arguments. However, this requires the preprocessor to be aware of and keep track the various different static arguments and specialized function versions, which it currently is not able to do.

### 5.2.3. Removing annotations from helper functions

Currently, we have to “predict” which versions of the staged helper functions will be needed by a later-defined function for that later-defined function to be able to use the staged version of the helper function. However, it should be possible to analyze the static information available to the helper functions without any need for annotation from the body and annotations of the main function, and generate the required staged version of the helper function “on the fly”.

### 5.2.4. Static analysis and optimization

The current generated code can contain some inefficiencies. For example, the code produced by the staged plus function where both arguments are static looks like this

```
let plus_xn x n = .< let plus = x + n in plus  >.
# plus_xn 3 5;;
- : int code = .< let plus = 3 + 5 in plus>.
```

Here, `3 + 5` is not evaluated before it is spliced into the code even though this is a static computation that can be eliminated from the produced code like so

```
let plus_xn x n = let r = 3 + 5 in .< let plus = r in plus  >.
# plus_xn 3 5;;
- : int code = .< let plus = 8 in plus>.
```

We should be able to detect static computations such as this and float them out to be evaluated in the generator level first before they are spliced into the produced code. This will involve doing a static analysis to figure out if a function application can be statically computed and then adding a translation rule that handles the insertion of the let-bound evaluation into the staged function and the substitution of the computation in the function body with the result of the let-bound evaluation.

Another type of inefficiency can be seen in the produced code of one of the previous examples:

```
.< let ff m = ((* CSP pow *) m) + ((* CSP pow *) m) + 0 in ff >.
```

We can see that the call `(* CSP pow *) m` is unnecessarily computed twice due to code duplication. This is a known problem with generative meta-programming known as code explosion with various proposed solution such as writing the function to be staged in continuation-passing style, in monadic form or by using abstract interpretation (Jonnalagedda et al., 2014; Rompf & Odersky, 2010; Swadi, Taha, Kiselyov, & Pasalic, 2006; Taha, 2004). From all the proposed solution abstract interpretation and LMS (Kiselyov et al., 2004; Rompf & Odersky, 2010) seems to be the most promising if we can implement most of the models and operators at the back-end such that the users of the system does not have to deal with them for the most cases.

Abstract interpretation and LMS might also be able to allow rewritings for cases where the computation can be simplified. The simplest example is the removal of redundant computations such as the case with the staged power function:

```
.< let pow x = (x * (x * (x * 1))) in pow >.
```

Here we have a redundant multiplication by 1 which can be removed before the code is generated by using abstract interpretation and LMS. We should also be able to detect if a staged computation is fully static (e.g. `.<2 + 3>.`) and compute them during generation time instead of splicing the delayed computation as is using the same technique.

### 5.2.5. Rewriting the preprocessor using AST quasi-quotations

Instead of pattern matching on the AST directly and creating them using the constructors in Ast_helper, we can use AST quasi-quotations (Frisch, 2013c) instead. AST quasi-quotations allows us to embed AST fragments as literals in the source code. For example, it is possible to write `[%expr x * pow x]` to create the AST for the expression `x * pow x`. It is also possible to pattern match over AST fragments. For example to get the left hand side and right hand side of an application of the plus function we can write `[%expr [%e? lhs] + [%e? rhs]]`. This will make the ppx preprocessor source code neater and easier to maintain.

## References

Agarwal, A., & Lesourd, M. (2014). Camlp4 Wiki. from https://github.com/ocaml/camlp4/wiki

Bourgoin, M., & Chailloux, E. (2014). *GPGPU Composition with OCaml*. Paper presented at the Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, United Kingdom.

Carette, J. (2006). Gaussian elimination: a case study in efficient genericity with MetaOCaml. *Sci. Comput. Program., 62*(1), 3-24.

Czarnecki, K., O'Donnell, J., Striegnitz, J., & Taha, W. (2004). DSL Implementation in MetaOCaml, Template Haskell, and C++. In C. Lengauer, D. Batory, C. Consel, & M. Odersky (Eds.), *Domain-Specific Program Generation* (Vol. 3016, pp. 51-72): Springer Berlin Heidelberg.

de Rauglaudre, D. (2003). Camlp5 - Reference Manual. from http://caml.inria.fr/pub/docs/manual-camlp4/index.html

Frigo, M. (1999). *A Fast Fourier Transform Compiler*. Paper presented at the Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA.

Frisch, A. (2013a, 16 April 2014). extension_points.txt. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/experimental/frisch/extension_points.txt

Frisch, A. (2013c). ppx_tools. from https://github.com/alainfrisch/ppx_tools

Futamura, Y. (1999). Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher Order Symbol. Comput., 12*(4), 381-391.

Herrman, C. A. (2005). Generating Message-Passing Programs from Abstract Specification by Partial Evaluation. *Parallel Processing Letters, 15*(3), 305-320.

Herrmann, C. A., & Langhammer, T. (2006). Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Science of Computer Programming, 62*(1), 47-65.

Inria. (1995a, 22 December 2014). asttypes.mli. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/parsing/asttypes.mli

Inria. (1995c, 13 March 2015). parsetree.mli. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/parsing/parsetree.mli

Inria. (2013a, 13 March 2015). ast_helper.mli. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/parsing/ast_helper.mli

Inria. (2013c, 4 October 2014). ast_mapper.mli. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/parsing/ast_mapper.mli

Inria. (2014). Language Extensions - Attributes, Extension nodes, Quoted Strings. from http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec241

Jonnalagedda, M., Coppey, T., Stucki, S., Rompf, T., & Odersky, M. (2014). *Staged parser combinators for efficient data processing*. Paper presented at the Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, Portland, Oregon, USA.

Kiselyov, O. (2010, 10 January 2015). MetaOCaml -- an OCaml dialect for multi-stage programming. from http://okmij.org/ftp/ML/MetaOCaml.html

Kiselyov, O., Swadi, K. N., & Taha, W. (2004). *A methodology for generating verified combinatorial circuits*. Paper presented at the Proceedings of the 4th ACM international conference on Embedded software, Pisa, Italy.

Langhammer, T. (2005). *Tuning MetaOCaml programs for high performance.* Retrieved from http://www.infosun.fmi.uni-passau.de/cl/arbeiten/Langhammer.pdf

Puschel, M. (2011). *Automatic Performance Programming?* Paper presented at the ACM international conference companion on Object oriented programming systems languages and applications companion, Portland, Oregon, USA.

Puschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., . . . Rizzolo, N. (2005). SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, 93*(2), 232-275.

Rompf, T., & Odersky, M. (2010). *Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs*. Paper presented at the 9th international conference on Generative programming and component engineering, Eindhoven, The Netherlands.

Spampinato, D. G., & Puschel, M. (2014). *A Basic Linear Algebra Compiler*. Paper presented at the Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, Orlando, FL, USA.

Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., & Olukotun, K. (2014). Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst., 13*(4s), 1-25. doi: 10.1145/2584665

Sujeeth, A. K., Gibbons, A., Brown, K. J., Lee, H., Rompf, T., Odersky, M., & Olukotun, K. (2013). *Forge: generating a high performance DSL implementation from a declarative specification*. Paper presented at the 12th international conference on Generative programming: concepts &#38; experiences, Indianapolis, Indiana, USA.

Swadi, K., Taha, W., Kiselyov, O., & Pasalic, E. (2006). *A monadic approach for avoiding code duplication when staging memoized functions*. Paper presented at the Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, Charleston, South Carolina.

Taha, W. (1999). *Multi-stage programming: Its theory and applications.* Oregon Graduate Institute of Science and Technology.

Taha, W. (2004). A Gentle Introduction to Multi-stage Programming. from http://www.cs.rice.edu/~taha/publications/journal/dspg04a.pdf

Whaley, R. C., Petitet, A., & Dongarra, J. J. (2001). Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing, 27*(1--2), 3-35. doi: citeulike-article-id:6904486

Zotov, P. (2014). A Guide to Extension Points in OCaml. from http://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/

# Appendix A: Glossary of Translation Rules

## Function Definitions

```
                  <FunBody> => ([<Aux1>, …, <Auxn>], <FunBodyS>)
--------------------------------------------------------------------------------
                       let <FunName> <Args> = <FunBody>
                                   =>toMeta
   let <FunName>_staged <StaticArgs> =
     let <StagedFun> = !. .< .~(<StagedFun>_staged <StaticArgs>) >. in
     let <AuxName1> = <AuxBody1>
     and …
     and <AuxNamen> = <AuxBodyn> in
     .< let <FunName> <DynArgs> = .~(<FunBodyS>) in <FunName> >.
```

## If-Then-Else Control Structure

```
                              isStatic(CondExp)
                    <ThenBody> => (ThenAuxList, <ThenBodyS>)
                    <ElseBody> => (ElseAuxList, <ElseBodyS>)
    --------------------------------------------------------------------------
                  if <CondExp> then <ThenBody> else <ElseBody>
                                      =>
                         ( ThenAuxList @ ElseAuxList @
                 [if <CondExp>
                    then .< .~(<ThenBodyS[DynArg->.~DynArg]>) >.
                    else .< .~(<ElseBodyS[DynArg->.~DynArg]>) >.]
                                       ,
                   .< .~(aux <Args[DynArg->.<DynArg>.]>) >. )
```

```
                              !isStatic(CondExp)
                         <CondExp> => (CondAuxList, <CondExpS>)
                        <ThenBody> => (ThenAuxList, <ThenBodyS>)
                        <ElseBody> => (ElseAuxList, <ElseBodyS>)
    ____________________________________________________________________
                    if <CondExp> then <ThenBody> else <ElseBody>
                                       =>
                      ( CondAuxList @ ThenAuxList @ ElseAuxList
                                       ,
       .< if .~(<CondExpS>) then .~(<ThenBodyS>) else .~(<ElseBodyS>) >. )
```

Match Control Structure

```
                              isStatic(CondExp)
                 <BranchBody_1> => (BranchAuxList_1, <BranchBodyS_1>),
                                      …,
                 <BranchBody_n> => (BranchAuxList_n, <BranchBodyS_n>)
  ______________________________________________________________________________
                 match <CondExp> with
                   <BranchPatt_1> -> <BranchBody_1>
                   | …
                   | <BranchPatt_n> -> <BranchBody_n>
                                       =>
              ( BranchAuxList_1 @ … @ BranchAuxList_n @
         [match <CondExp> with
            <BranchPatt_1> -> .< .~(<BranchBodyS_1[DynArg->.~DynArg]>) >.
            | …
            | <BranchPatt_n> -> .< .~(<BranchBodyS_n[DynArg->.~DynArg]>) >.]
                                       ,
                .< .~(aux <Args[DynArg->.<DynArg>.]>) >. )
```

```
                    !isStatic(CondExp)
              <CondExp> => (CondAuxList, <CondExpS>)
         <BranchBody1> => (BranchAuxList1, <BranchBodyS1>),
                                  …,
         <BranchBodyn> => (BranchAuxListn, <BranchBodySn>)
_________________________________________________________________________
         match <CondExp> with
           <BranchPatt1> -> <BranchBody1>
           | …
           | <BranchPattn> -> <BranchBodyn>
                                  =>
  ( CondAuxList @ BranchAuxList1 @ … @ BranchAuxListn
                                  ,
      .< match .~(<CondExpS>) with
           <BranchPatt1> -> .~(<BranchBodyS1>)
           | …
           | <BranchPattn> -> .~(<BranchBodySn>) >. )
```

Function Application

```
               isRecursiveCall()   isInAux()
              <Args> => (ArgsAuxList, <ArgsS>)
          <ArgsS'> = <ArgsS[DynArgS->.<DynArgS>.]>
________________________________________________________
                    <FunName> <Args>
                           =>
     ( ArgsAuxList, .< <FirstAuxName> .~(<ArgsS'>) >. )
```

```
               isRecursiveCall()   !isInAux()
          <DynArgs> => (DynArgsAuxList, <DynArgsS>)
_______________________________________________
                    <FunName> <Args>
                           =>
     ( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

```
                  isPrevStagedFun()
     <DynArgs> => (DynArgsAuxList, <DynArgsS>)
_________________________________________________
                  <FunName> <Args>
                         =>
( DynArgsAuxList, .< <FunName> .~(<DynArgsS>) >. )
```

```
         <Args> => (ArgsAuxList, <ArgsS>)
______________________________________________
                <FunName> <Args>
                       =>
   ( ArgsAuxList, .< <FunName> .~(<ArgsS>) >. )
```

Default Rule

```
__________________________________
<FunBody> => ([], .< <FunBody> >.)
```

# Appendix B: Sample OCaml AST

```
let rec pow x n = if n = 0 then 1 else x * (pow x (n - 1)) [@@static [n]]
==>
[{pstr_desc =
   Pstr_value (Recursive,
    [{pvb_pat = {ppat_desc = Ppat_var {txt = "pow"}};
      pvb_expr =
       {pexp_desc =
         Pexp_fun ("", None, {ppat_desc = Ppat_var {txt = "x"}},
          {pexp_desc =
            Pexp_fun ("", None, {ppat_desc = Ppat_var {txt = "n"}},
             {pexp_desc =
               Pexp_ifthenelse
                ({pexp_desc =
                   Pexp_apply ({pexp_desc = Pexp_ident {txt = Lident "="}},
                    [("", {pexp_desc = Pexp_ident {txt = Lident "n"}});
                     ("", {pexp_desc = Pexp_constant (Const_int 0)})])},
                {pexp_desc = Pexp_constant (Const_int 1)},
                Some {pexp_desc =
                   Pexp_apply ({pexp_desc = Pexp_ident {txt = Lident "*"}},
                    [("", {pexp_desc = Pexp_ident {txt = Lident "x"}});
                     ("", {pexp_desc =
                        Pexp_apply
                         ({pexp_desc = Pexp_ident {txt = Lident "pow"}},
                         [("", {pexp_desc = Pexp_ident {txt = Lident "x"}});
                          ("", {pexp_desc =
                             Pexp_apply
                              ({pexp_desc = Pexp_ident {txt = Lident "-"}},
                              [("", {pexp_desc = Pexp_ident {txt = Lident "n"}});
                               ("", {pexp_desc = Pexp_constant (Const_int 1)})])})])})])})})})};
      pvb_attributes =
       [({txt = "static"},
         PStr
          [{pstr_desc =
             Pstr_eval
              ({pexp_desc =
                 Pexp_construct ({txt = Lident "::"},
                  Some {pexp_desc =
                     Pexp_tuple
                      [{pexp_desc = Pexp_ident {txt = Lident "n"}};
                       {pexp_desc =
                         Pexp_construct ({txt = Lident "[]"}, None)}]})},
              ...)}])]}])}]
```