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 Signapore

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 Signapore

2014/2015

Project No: H018590

Advisor: Assoc. Prof. Chin Wei Ngan

Deliverables:

Report: 1 Volume

Program: 1 CD (TODO confirm this)

Abstract

(TODO write abstract)

Subject Descriptors:

(TODO find classification)

Keywords

(TODO find keywords)

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 implementation (Czarnecki, O'Donnell, Striegnitz, & Taha, 2004; Futamura, 1999; Herrmann & Langhammer, 2006).

The many opportunities provided by meta-programming 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 graph 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 concludes the report and points out areas of possible future work.

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 implement 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 code fragment below

```
- : 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 run.

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).

2.1.2. Escape `.~`

Escapes evaluate code pointed by the operator to produce code fragments which is then spliced into its surrounding bracket. This implies that escape can only be applied to values of type code or function that return values 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 add 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 in to 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 = 11
```

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 creates and manipulate other programs. Another important feature of MetaOCaml that's worth noting is the ability to use values and functions from the generator level in the generated code (cross-stage persistence). E.g.

```
# let plus2genLevel = (+) 2;;
val plus2genLevel : int -> int = <fun>
# let plus2 = .< plus2CSP 3 >.;;
val plus2 : int code = .< (* CSP plus2genLevel *) 3 >.
# !. plus2
- : int = 5
```

What's happening here is we have `plus2genLevel` function from the generator level used in the generated code level. As the generator level function is the form of 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 generator level and the code from generated level is linked accordingly and the `plus2genLevel` function is used. This however requires that the

generator code 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 to be run at a later time.

2.2. OCaml Extension Points

OCaml extension points (Frisch, 2013; Zotov, 2014), available from OCaml 4.02.1 onwards, are generic structures embedded in the OCaml abstract syntax tree (AST). These structures can be used to attach extra information which can be used to expand OCaml syntax by processing the information with a ppx preprocessor. There are three types of extension structures 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. Attributes have an id and a potentially empty payload which is written as

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

depending on whether the attribute is attached to an OCaml expression, structure item, or floats independently.

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.

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.

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 them with the preprocessor and transform them 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, 1995b). The library Ast_mapper (Inria, 2013b) 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 that default mapper at the parts where an annotation might have been placed, checks 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. ppx_toMeta

This section describes the implementation of ppx_toMeta, a 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 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 and specialized to the given static information. The produced code is thus optimized for the given static information.

3.1. Translation Scheme

We first need to come up with a translation scheme 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 always the same, for example if y is always 2, we can say that y is static and we may specialize this function like so

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

what we want is then 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 like so

```
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 plus with y to be a static value of 2. Also notice that the produce 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 functions

```
let <FunName> <Args> = <FunBody>
                                    =>
let <FunName>_staged <StaticArgs> =
  .< let <FunName> <DynArgs> = <FunBody> in <FunName> >.
```

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 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. 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 >.
```

From here we get the following translation rules:

```
let <FunName> <Args> =
  if <CondExp> then <ThenBody> else <ElseBody>
and CondExp can be evaluated statically
                                    =>
let <FunName>_staged <StaticArgs> =
  let aux <Args> =
    if <CondExp> then .< <ThenBody> >. else .< <ElseBody> >.
    in .< let <FunName> <DynArgs> =
             .~(aux <Args [DynArg -> .<DynArg>.]> )
          in <FunName> >.
```

Where [x -> y] means the substitution of every occurrence of x with y in the given structure. Note that this rule can be extended for other control structures (e.g. match…with…| …|…) and the implemented ppx_toMeta does cover more than just if…then...else... but as the extension is straightforward it is omitted in this report for brevity.

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 is use is 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 >.
```

From here, we can derive the translation rule for recursive functions as follows

```
let rec <FunName> <Args> =
    if <CondExp> then <ThenBody> else <ElseBody>
and CondExp can be evaluated statically
                                    =>
let <FunName>_staged <StaticArgs> =
  let aux <Args> =
    if <CondExp>
      then .< <ThenBody [<FunName> <Args> -> .~(aux <Args>)]> >.
      else .< <ElseBody [<FunName> <Args> -> .~(aux <Args>)]> >.
  in .< let <FunName> <DynArgs > =
          .~(aux <Args [DynArg -> .<DynArg>.]> )
        in <FunName> >.
```

and

```
let rec <FunName> <Args> = <FunBody>
                                    =>
let <FunName>_staged <StaticArgs> =
  .< let rec <FunName> <DynArgs> =
       <FunBody [<FunName> <Args> -> <FunName> <DynArgs>]>
     in <FunName> >.
```

3.1.4. Function that uses other staged function

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

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

if n here is static and we have already previously staged pow over n, we can stage ff as follows

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

What's happening here is first we produce the code for pow specialized to the static argument n, and we compile it so it can be used later on. Then, we stage the function ff as normal while making sure that we call 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 x =
           if x = 0 then 0 else ((* CSP pow *) x) + (ff (x-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 n =
  .< let pow = .~(pow_staged n) in
     let rec ff x =
       if x = 0 then 0 else (pow x) + (ff (x - 1)) in ff >.

# ffstaged 3;;
```

```
- : (int -> int) code =
      .< let pow = (x * (x * (x * 1))) in
         let rec ff x =
           if x = 0 then 0 else (pow  x) + (ff (x-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 x is static in ff

```
let ff_staged x =
  let rec aux x n =
    if x = 0 then .< 0 >. else .< (pow .~n) + .~(aux (x – 1) n) >.
  in .< let pow = .~(pow_staged x) in
        let ff n = .~(aux x .<n>.) 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, which means that we are actually trying to use something that does not exist yet. So, we should declare and create the function pow first before we use it in the auxiliary function

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

# ff_staged 2;;
- : (int -> int) code =
      .< let ff n = ((* CSP pow *) n) + ((* CSP pow *) n) + 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 generator environment. However, if we take it as ff_staged will always have to be invoked first to produce the specialized ff before ff is used, the generator environment will always be available to the specialized ff.

From this, we get the following translation rule:

```
let <FunName> <Args> = <FunBody>
and <FunBody> uses a staged function <StagedFun>
                                    =>
let <FunName> <StaticArgs> =
  let <StagedFun> = !. .< .~(<StageFun>_staged <StaticArgs>) >. in
  Stage[<FunBody>]
```

where Stage[x] means to stage x using the previously established translation rules

Using the above translation scheme, we now know how to transform OCaml code into staged MetaOCaml code given which information is static to the original OCaml code.

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 extract it out from which argument is static in ff (TODO PROBLEM what if we want to use the previously staged function over variables which

are not the argument of the bigger function?). 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.

The full source code of the implemented ppx preprocessor can be found in Appendix A. Also, 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, 1995b, 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, 2013b) 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 following

```
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, whether or not it is recursive, and whether or not the control expressions in the function (if any) can be computed statically.

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 defininitons are recursive or not and the value_binding _list which is the actual structure 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. For figuring out whether or not control structures in the function have conditional expressions which can be computed statically, we pattern match for `Pexp_ifthenelse` or `Pexp_match` and see if the conditional expression contained within them only uses static information.

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 the staged version of the function as follows:

The first thing to build is the main staged body. Following the translation rules, this will be the original body (potentially with static arguments for recursive calls and calls to previously staged function omitted) if we cannot compute branches statically or an escaped call to the auxiliary function if we can, enclosed by a MetaOCaml bracket.

Next, if we need to use an auxiliary function we will have to build the auxiliary function and attach is to the main staged body as a let-bound value. To create the auxiliary function we take the original function body and stage the branches of the control structure. We will also need to change any recursive calls to the original function into an escaped recursive call to the auxiliary function. If there is any call to a previously staged function we'll need to omit the static arguments as well.

Lastly, if any previously staged function was used we'll need to add it's declaration and generation of the specialized function from the previously staged function as a let-bound value and attach it on top of the auxiliary function (if any) and the main staged body.

The main work that needs to be done in building the various parts of the staged function above 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.

Substituting nodes (for substitution of recursive calls and previously staged function calls) basically requires us to pattern-match down the AST to locate the things we want to substitute. 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 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 do our substitutions.

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

(TODO PROBLEM how to explain this without ended up copy-pasting whole code? Add example?)

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. 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.

4.1. Summary

(TODO summarize the report. Almost the same like abstract?)

4.2. Limitations and Recommendation for Future Work

4.2.1. Scope of translation scheme

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

Some example of code patterns which the translation scheme currently cannot handle are as follows:

4.2.1.1. Nested control structures

The translation scheme currently does not handle the staging of functions of nested control structures. For example, if we have the source as below

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

We can actually stage this using the rule we have for programs with control structures if none of the arguments or only x is static but if we have y as static, the expected program is the following

```
let f_staged y =
  let rec 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 the expected program is as follows

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

where the then-branch of the outer control structure is not staged but computed in the generator-level.

Both the cases above is not currently covered by the translation scheme, which will produce an error for the first case and completely delays the then-branch of the outer branching for the second case.

4.2.1.2. Mutating static variables

The current translation scheme completely eliminates static arguments from the 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) * … )`

The two examples above shows the inadequacy of the current translation scheme and a more general and comprehensive translation scheme that covers the complete range of code patterns is a subject for further research.

4.2.2. 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 and 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 be processed currently.

Argument naming is also a problem as currently the preprocessor assumes that the arguments of a previously staged function have the same variable name as the one used when they are called. (TODO PROBLEM this is a real bug. Shouldn't assume same name but instead use argument index. Also, argument of used functions should not be assumed to always be the argument of main function)

4.2.3. 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 annotation of the main function and generate the required staged version of the helper function “on the fly”.

4.2.4. Static analysis and optimization

The current generated code can contain some inefficiencies. For example one of the examples had this as the final produced code

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

We can see that the call (* CSP pow *) n 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 saged 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.

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

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. (1995b, 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. (2013b, 4 October 2014). ast_mapper.mli. from http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/parsing/ast_mapper.mli

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 & 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: ppx_toMeta.ml

```
open Ast_mapper
open Ast_helper
open Asttypes
open Parsetree
open Longident

let msg_syntaxErrorDecl = "Syntax error for toMeta annotation. Annotation is of the form [@@static <statVars>]
where statVars = [] | sv::statVars"
let msg_syntaxErrorUse = "Syntax error for toMeta annotation. Annotation is of the form [@static,use <statVars>]
where statVars = [] | sv::statVars"

let applyFun = fun funTargets funName loc ->
  Exp.apply ~loc ~attrs:[]
    (Exp.ident ~loc ~attrs:[] {loc = loc; txt = Lident funName})
    (List.map (fun ft ->  ("", ft)) funTargets)

let applyMetaOCamlConstr = fun constrTarget constrName loc ->
  Exp.mk ~loc
    ~attrs:([({txt = constrName; loc = loc}, PStr [])] @ constrTarget.pexp_attributes)
    constrTarget.pexp_desc

let applyLift = fun liftTarget loc ->
  applyMetaOCamlConstr liftTarget "metaocaml.bracket" loc

let applyEsc = fun escTarget loc ->
  applyMetaOCamlConstr escTarget "metaocaml.escape" loc

let applyRun = fun runTarget loc ->
  applyFun [runTarget] "Runcode.run" loc

let getStagedName = fun fname statVars ->
  fname^"_"^(List.fold_left (fun acc v -> acc^v) "" statVars)

let isRecursive = fun funDef ->
  match funDef with
    {pstr_desc = Pstr_value (Nonrecursive, _)} -> false
    | {pstr_desc = Pstr_value (Recursive, _)} -> true
    | _ -> failwith "not a function definition"

let getVars = fun funDef ->
  let pvb_expr =
    match funDef with
      {pstr_desc = Pstr_value (_, value_binding_list)} ->
          (List.hd value_binding_list).pvb_expr
      | _ -> failwith "not a function definition"
  in let rec aux pvb_expr =
       match pvb_expr with
         {pexp_desc = Pexp_fun (_, _, {ppat_desc = Ppat_var {txt = v}}, pvb_expr)} ->
```

```
            v::(aux pvb_expr)
        | _ -> []
  in aux pvb_expr

let getAttrList = fun funDef ->
  match funDef with
    {pstr_desc = Pstr_value (_, value_binding_list)} ->
      (* value binding is a list for let .. and .. and .. *)
      (* TODO: handle ands *)
      (List.hd value_binding_list).pvb_attributes
    | _ -> failwith "not a function definition"

let hasToMetaAnnot = fun funDef ->
  let attrList = getAttrList funDef in
  let rec aux attrs =
    match attrs with
      [] -> false
      | ({txt = "static"}, _)::_ -> true
      | _::attrs -> aux attrs
  in aux attrList

let getStatVars = fun funDef ->
  let attrList = getAttrList funDef in
  let rec aux attrs =
    match attrs with
      [] -> []
      | ({txt = "static"}, PStr [{pstr_desc = Pstr_eval (construct, _)}])::attrs ->
          let rec aux2 construct =
            begin match construct with
              | Pexp_construct ({txt = Lident "[]"}, None) -> []
              | Pexp_construct ({txt = Lident "::"},
                  Some {pexp_desc = Pexp_tuple [{pexp_desc = Pexp_ident {txt = Lident v}}; construct]}) ->
                      v::(aux2 construct.pexp_desc)
              | _ -> failwith msg_syntaxErrorDecl
            end in
          let statVars = aux2 construct.pexp_desc in
          statVars::(aux attrs)
      | ({txt = "static"}, _)::attrs -> failwith msg_syntaxErrorDecl
      | _::attrs -> aux attrs
  in aux attrList

let getUsedStagedFun = fun funBody ->
  let rec getUsf funBody =
    match funBody with
      {pexp_desc = Pexp_ifthenelse (cond, thenExp, elseExpOpt)} ->
          begin match elseExpOpt with
            None -> (getUsf cond)@(getUsf thenExp)
            | Some elseExp ->  (getUsf cond)@(getUsf thenExp)@(getUsf elseExp)
          end
      | {pexp_desc = Pexp_match (condExp, pattExpList)} ->
```

```
          (getUsf condExp)@(List.flatten (List.map (fun {pc_rhs = rhs} -> getUsf rhs) pattExpList))
      | {pexp_desc = Pexp_apply (fn, argList); pexp_attributes = attrs} ->
          let fname =
            begin match fn with
              {pexp_desc = Pexp_ident {txt = Lident fname}} -> fname
              | _ -> "_AnonFun"
            end in
          let usedHere =
            let rec aux attrs =
              match attrs with
                [] -> []
                | ({txt = "static.use"}, _)::_ -> [fname]
                | _::attrs -> aux attrs
            in aux attrs in
          let usedInArgs = List.flatten (List.map (fun (_,exp) -> getUsf exp) argList) in
          usedHere@usedInArgs
      | {pexp_desc = Pexp_construct (lid, constrExpOpt)} ->
          begin match constrExpOpt with
            None -> []
            | Some exp -> getUsf exp
          end
      | {pexp_desc = Pexp_tuple es} ->
          List.flatten (List.map (fun e -> getUsf e) es)
      | exp -> []
  in getUsf funBody

let rec removeArguments funBody n =
  if n = 0
    then funBody
    else
      match funBody with
        {pexp_desc = Pexp_fun (_, _, _, funBody)} -> removeArguments funBody (n-1)
        | _ -> failwith "arguments missing?"

let rec expContainsVar = fun exp var ->
  match exp with
    {pexp_desc = Pexp_apply (op, es)} ->
        List.exists (fun (_, e) -> expContainsVar e var) es
    | {pexp_desc = Pexp_ident {txt = Lident v}} ->
        v = var
    | _ -> false

let isControlVarStatic = fun statVars actualBody ->
  let rec aux body =
    match body with
      {pexp_desc = Pexp_ifthenelse (condExp, thenExp, elseExpOpt)} ->
          (List.exists (fun v -> expContainsVar condExp v) statVars)
            || aux thenExp
            || begin match elseExpOpt with Some e -> aux e | None -> false end
      | {pexp_desc = Pexp_match (condExp, pattExpList)} ->
```

```
          (List.exists (fun v -> expContainsVar condExp v) statVars)
            || List.fold_left (||) false (List.map (fun pattExp -> aux pattExp.pc_rhs) pattExpList)
      | {pexp_desc = Pexp_apply (op, es)} ->
          List.exists (fun (_, e) -> aux e) es
      | _ -> false
  in aux actualBody

let subAuxBody = fun funBody funName statVars dynVars loc ->
  let rec sub funBody inEsc =
    match funBody with
      {pexp_desc = Pexp_ifthenelse (condExp, thenExp, elseExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          let condExp' = sub condExp inEsc in
          let liftBranch = (List.exists (fun v -> expContainsVar condExp v) statVars) in
          let thenExp' = if liftBranch
                         then (applyLift (sub thenExp inEsc) loc)
                         else sub thenExp inEsc
          in let elseExpOpt' =
            begin match elseExpOpt with
              None -> None
              | Some elseExp ->
                  if liftBranch
                    then Some (applyLift (sub elseExp inEsc) loc)
                    else Some (sub elseExp inEsc)
            end in
          Exp.ifthenelse ~loc ~attrs condExp' thenExp' elseExpOpt'
      | {pexp_desc = Pexp_match (condExp, pattExpList); pexp_attributes = attrs; pexp_loc = loc} ->
          let condExp' = sub condExp inEsc in
          let liftBranch = (List.exists (fun v -> expContainsVar condExp v) statVars) in
          Exp.match_ ~loc ~attrs condExp'
            (List.map
               (fun {pc_lhs = lhs; pc_guard = guard; pc_rhs = rhs} ->
                  {pc_lhs = lhs; pc_guard = guard; pc_rhs = (if liftBranch
                                                             then applyLift (sub rhs inEsc) loc
                                                             else sub rhs inEsc)})
               pattExpList)
      | {pexp_desc = Pexp_apply (fn, argList); pexp_attributes = attrs; pexp_loc = loc} ->
          let fname =
            begin match fn with
              {pexp_desc = Pexp_ident {txt = Lident fname}} -> fname
              | _ -> "_AnonFun"
            end in
          let fn' =
            if fname = funName
              then Exp.ident ~loc ~attrs:[] {loc = loc; txt = Lident "aux"}
              else fn
          in let argList' =
            let rec aux args =
              match args with
                [] -> []
                | (lbl, exp)::args -> (lbl, sub exp (fname = funName))::(aux args)
```

```
            in aux argList
          in let e = Exp.apply ~loc ~attrs fn' argList' in
          if fname = funName then applyEsc e loc else e
      | {pexp_desc = Pexp_construct (lid, constrExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          begin match constrExpOpt with
            None -> Exp.construct ~loc ~attrs lid None
            | Some exp -> Exp.construct ~loc ~attrs lid (Some (sub exp inEsc))
          end
      | {pexp_desc = Pexp_tuple es; pexp_attributes = attrs; pexp_loc = loc} ->
          Exp.tuple ~loc ~attrs (List.map (fun e -> sub e inEsc) es)
      | {pexp_desc = Pexp_ident {txt = Lident v; loc = loc}} ->
          if not inEsc && (List.exists (fun dv -> v=dv) dynVars)
            then applyEsc funBody loc
            else funBody
      | exp -> exp
  in sub funBody false

let buildAuxCall = fun vars statVars dynVars loc ->
  let newBodyArgs =
    List.map
      (fun v ->
           let identExp = Exp.ident ~loc ~attrs:[] {loc = loc; txt = Lident v} in
           if List.exists (fun dv -> v=dv) dynVars
             then ("", applyLift identExp loc)
             else ("", identExp))
      vars
  in let newBody =
    Exp.apply ~loc ~attrs:[]
      (Exp.ident ~loc ~attrs:[] {loc = loc; txt = Lident "aux"})
      newBodyArgs
  in applyEsc newBody loc

let subUsedStagedFun = fun funBody usedStagedFun statVars ->
  let rec sub funBody =
    match funBody with
      {pexp_desc = Pexp_ifthenelse (cond, thenExp, elseExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          begin match elseExpOpt with
            None -> Exp.ifthenelse ~loc ~attrs cond (sub thenExp) None
            | Some elseExp -> Exp.ifthenelse ~loc ~attrs:[] cond (sub thenExp) (Some (sub elseExp))
          end
      | {pexp_desc = Pexp_match (condExp, pattExpList); pexp_attributes = attrs; pexp_loc = loc} ->
          Exp.match_ ~loc ~attrs condExp
            (List.map
               (fun {pc_lhs = lhs; pc_guard = guard; pc_rhs = rhs} ->
                  {pc_lhs = lhs; pc_guard = guard; pc_rhs = sub rhs})
               pattExpList)
      | {pexp_desc = Pexp_apply (fn, argList); pexp_attributes = attrs; pexp_loc = loc} ->
          let fname =
            begin match fn with
              {pexp_desc = Pexp_ident {txt = Lident fname}} -> fname
```

```
              | _ -> "_AnonFun"
            end in
          let isStagedFun =
            let rec aux sfs =
              match sfs with
                [] -> false
                | sfn::sfs -> if fname = sfn then true else aux sfs
            in aux usedStagedFun in
          let argList' =
            if isStagedFun
              then
                let rec aux args =
                  match args with
                    [] -> []
                    | (lbl, exp)::args ->
                        match exp with
                          {pexp_desc = Pexp_ident {txt = Lident v}} ->
                            if List.exists (fun sv -> sv=v) statVars
                              then aux args
                              else (lbl, exp)::(aux args)
                    | exp -> (lbl, sub exp)::(aux args)
                in aux argList
              else List.map (fun (lbl, exp) -> (lbl, sub exp)) argList
          in let attrs' =
            let rec aux attrs =
              match attrs with
                [] -> []
                | ({txt = "static.use"}, _)::attrs -> attrs
                | attr::attrs -> attr::(aux attrs)
            in aux attrs in
          Exp.apply ~loc ~attrs:attrs' fn argList'
      | {pexp_desc = Pexp_construct (lid, constrExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          begin match constrExpOpt with
            None -> Exp.construct ~loc ~attrs lid None
            | Some exp -> Exp.construct ~loc ~attrs lid (Some (sub exp))
          end
      | {pexp_desc = Pexp_tuple es; pexp_attributes = attrs; pexp_loc = loc} ->
          Exp.tuple ~loc ~attrs (List.map (fun e -> sub e) es)
      | exp -> exp
  in sub funBody

let subRecCall = fun funBody funName statVars ->
  let rec sub funBody =
    match funBody with
      {pexp_desc = Pexp_ifthenelse (cond, thenExp, elseExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          begin match elseExpOpt with
            None -> Exp.ifthenelse ~loc ~attrs cond (sub thenExp) None
            | Some elseExp -> Exp.ifthenelse ~loc ~attrs:[] cond (sub thenExp) (Some (sub elseExp))
          end
      | {pexp_desc = Pexp_match (condExp, pattExpList); pexp_attributes = attrs; pexp_loc = loc} ->
```

```
          Exp.match_ ~loc ~attrs condExp
            (List.map
               (fun {pc_lhs = lhs; pc_guard = guard; pc_rhs = rhs} ->
                  {pc_lhs = lhs; pc_guard = guard; pc_rhs = sub rhs})
               pattExpList)
      | {pexp_desc = Pexp_apply (fn, argList); pexp_attributes = attrs; pexp_loc = loc} ->
          let fname =
            begin match fn with
              {pexp_desc = Pexp_ident {txt = Lident fname}} -> fname
              | _ -> "_AnonFun"
            end in
          let argList' =
            let rec aux args =
              match args with
                [] -> []
                | (lbl, exp)::args ->
                    match exp with
                      {pexp_desc = Pexp_ident {txt = Lident v}} ->
                          if fname = funName && List.exists (fun sv -> sv=v) statVars
                            then aux args
                            else (lbl, exp)::(aux args)
                      | exp -> (lbl, sub exp)::(aux args)
            in aux argList
          in Exp.apply ~loc ~attrs fn argList'
      | {pexp_desc = Pexp_construct (lid, constrExpOpt); pexp_attributes = attrs; pexp_loc = loc} ->
          begin match constrExpOpt with
            None -> Exp.construct ~loc ~attrs lid None
            | Some exp -> Exp.construct ~loc ~attrs lid (Some (sub exp))
          end
      | {pexp_desc = Pexp_tuple es; pexp_attributes = attrs; pexp_loc = loc} ->
          Exp.tuple ~loc ~attrs (List.map (fun e -> sub e) es)
      | exp -> exp
  in sub funBody

let buildArgList args body loc =
  let rec aux args =
    match args with
      [] -> body
      | arg::args ->
          Exp.fun_ ~loc ~attrs:[] "" None
            (Pat.var ~loc ~attrs:[] {loc = loc; txt = arg})
            (aux args)
  in aux args

let buildStagedBody = fun funRec statVars dynVars funBody funName loc ->
  let recFlag = if funRec then Recursive else Nonrecursive in
  let letBody =
    Exp.let_ ~loc ~attrs:[] recFlag
      [Vb.mk ~loc ~attrs:[]
         (Pat.var ~loc ~attrs:[] {loc = loc; txt = funName})
```

```
        (buildArgList dynVars funBody loc)]
      (Exp.ident ~loc ~attrs:[] {loc=loc;txt=Lident funName})
  in applyLift letBody loc

let getStagedBody = fun funBody loc vars statVars dynVars funRec funName usedStagedFun ->
  let useAux = isControlVarStatic statVars funBody in
  let toStageBody =
    if useAux
      then buildAuxCall vars statVars dynVars loc
      else subRecCall funBody funName statVars
  in let toStageBody' =
    if (List.length usedStagedFun) > 0
      then subUsedStagedFun toStageBody usedStagedFun statVars
      else toStageBody
  in let stagedBody = buildStagedBody (funRec && (not useAux))
                       statVars dynVars toStageBody' funName loc
  in let fullBody =
    if useAux
      then let auxBody = subAuxBody funBody funName statVars dynVars loc in
           let auxBody' =
             if (List.length usedStagedFun) > 0
               then subUsedStagedFun auxBody usedStagedFun statVars
               else auxBody
           in Exp.let_ ~loc ~attrs:[] Recursive
             [Vb.mk ~loc ~attrs:[]
                (Pat.var ~loc ~attrs:[] {loc = loc; txt = "aux"})
                (buildArgList vars auxBody' loc)]
           stagedBody
      else stagedBody
  in let fullBodyWithDecl =
    let rec aux usfs =
      match usfs with
        [] -> fullBody
        | usfn::usfs ->
            let declBody =
              applyRun (applyLift (applyEsc (applyFun
                (List.map (fun v -> Exp.ident ~loc ~attrs:[] {txt = Lident v; loc = loc}) statVars)
                (getStagedName usfn statVars) loc) loc) loc) loc
            in Exp.let_ ~loc ~attrs:[] Nonrecursive
                 [Vb.mk ~loc ~attrs:[]
                   (Pat.var ~loc ~attrs:[] {loc = loc; txt = usfn})
                   declBody]
                 (aux usfs)
    in aux usedStagedFun
  in buildArgList statVars fullBodyWithDecl loc

let buildMeta = fun funRec funDef vars statVars dynVars ->
  let strLoc = funDef.pstr_loc in
  let f =
    match funDef.pstr_desc with
```

```
        Pstr_value (_, f) -> f
        | _ -> failwith "not a function definition"
  in let vbLoc = (List.hd f).pvb_loc in
  let funName =
    match (List.hd f).pvb_pat with
      {ppat_desc = Ppat_var {txt = fn}} -> fn
      | _ -> failwith "not a valid fun name pattern"
  in let stagedName = Pat.var ~loc:vbLoc ~attrs:[] {loc = vbLoc; txt = getStagedName funName statVars} in
  let funBody = removeArguments (List.hd f).pvb_expr (List.length vars) in
  let usedStagedFun = getUsedStagedFun funBody in
  let stagedBody = getStagedBody funBody vbLoc vars statVars dynVars funRec funName usedStagedFun in
  Str.value ~loc:strLoc Nonrecursive [Vb.mk ~loc:vbLoc ~attrs:[] stagedName stagedBody]

let toMeta_mapper argv =
  { default_mapper with
    structure = fun mapper structure_item_list ->
      let structure_item_mapper = fun mapper structure_item ->
        match structure_item with
          {pstr_desc = Pstr_value (_, _)} ->
              if hasToMetaAnnot structure_item
                then
                  let r = isRecursive structure_item in
                  let vars = getVars structure_item in
                  let statVarVariants = getStatVars structure_item in
                  let rec aux svvs =
                    match svvs with
                      [] -> []
                      | statVars::svvs ->
                          let dynVars = List.filter (fun v -> not (List.exists (fun sv -> v=sv) statVars)) vars in
                          (buildMeta r structure_item vars statVars dynVars)::(aux svvs)
                  in (default_mapper.structure_item mapper structure_item)::(aux statVarVariants)
                else [default_mapper.structure_item mapper structure_item]
          | _ -> [default_mapper.structure_item mapper structure_item]
      in List.flatten (List.map (structure_item_mapper mapper) structure_item_list)
  }

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

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)}])},
               ...)}])])}];
```