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

# Abstract

Meta prog is good but hard so automate

Subject Descriptors:

A

Keywords

Programming Languages & Systems, Program Analysis and Optimization

Implementation Software:

OCaml 4.02.1, BER MetaOCaml N102

# Table of Contents

[Abstract ii](#_Toc415094597)

[Table of Contents iii](#_Toc415094598)

[1. Introduction 1](#_Toc415094599)

[2. Tools 3](#_Toc415094600)

[2.1. MetaOCaml 3](#_Toc415094601)

[2.1.1. Bracket .< … >. 3](#_Toc415094602)

[2.1.2. Escape .~ 4](#_Toc415094603)

[2.1.3. Run !. (Runcode.run) 4](#_Toc415094604)

[2.2. OCaml Extension Points 6](#_Toc415094605)

[2.3. ppx Preprocessor 6](#_Toc415094606)

[3. ppx\_toMeta 7](#_Toc415094607)

[3.1. Goal 7](#_Toc415094608)

[3.2. Translation Scheme 7](#_Toc415094609)

[3.3. Implementation 7](#_Toc415094610)

[3.3.1. Source Code Annotation 7](#_Toc415094611)

[3.3.2. ppx Preprocessor 7](#_Toc415094612)

[3.3.2.1. Hooking to default mapper 7](#_Toc415094613)

[3.3.2.2. Extracting information from the annotations 7](#_Toc415094614)

[3.3.2.3. Building the staged function 7](#_Toc415094615)

[3.3.2.4. Combining the results 7](#_Toc415094616)

[3.3.3. Generating and prettyprinting the staged code 7](#_Toc415094617)

[4. Conclusions 8](#_Toc415094618)

[4.1. Summary 8](#_Toc415094619)

[4.2. Limitations 8](#_Toc415094620)

[4.2.1. Scope of translation scheme 8](#_Toc415094621)

[4.2.2. Scope of preprocessor 8](#_Toc415094622)

[4.3. Recommendations for Further Work 8](#_Toc415094623)

[4.3.1. Removing annotations from helper functions 8](#_Toc415094624)

[4.3.2. Static analysis and optimization 8](#_Toc415094625)

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; 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.

1. 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.

* 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*.

* + 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).

* + 1. 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.

* + 1. 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.

* 1. OCaml Extension Points

OCaml extension points (Frisch, 2013; Zotov, 2014) are generic structures embedded in the OCaml abstract syntax tree (AST). This feature isvailable from OCaml 4.02.1 onwards. 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

[%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 string literal and id can then be extracted and processed by a ppx preprocessor.

In this project, only attributes are used as what we need is only annotations on the original source code as will be explained in Section 3

On a side note, other tools for extending OCaml syntax is available such as

Camlp4 is an alternative but (Agarwal & Lesourd, 2014)

Camlp5 (de Rauglaudre, 2003)

Outdated

No documentation

Unsupported by metaocaml

* 1. ppx Preprocessor

As mentioned above, the generic extention point structures need to be preprocessed by a preprocessor to produce vanilla OCaml AST

To do this we need to access and modify the parsed AST

Compiler-libs

Asttypes Parsetree, the OCaml AST data structure

Ast\_mapper hook into it

Ast\_helper help write AST

1. ppx\_toMeta

The preproc inpml

* 1. Goal

Use a simple annotation scheme to cue the preprocessor to automatically stage functions

This is done by adding attributes to function definition and function calls to let the preprocessor know what to stage and how

* 1. Translation Scheme

A standard way to stage functions

Currently covers simple functions

Non recursive plus

Recursive pow (if then, match)

Also functions that uses previously staged function

* 1. Implementation
     1. Source Code Annotation
     2. ppx Preprocessor
        1. Hooking to default mapper
        2. Extracting information from the annotations
        3. Building the staged function
        4. Combining the results
     3. Generating and prettyprinting the staged code

1. Conclusions
   1. Summary
   2. Limitations
      1. Scope of translation scheme
      2. Scope of preprocessor
   3. Recommendations for Further Work
      1. Removing annotations from helper functions
      2. Static analysis and optimization

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.

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.

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.

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/>