Reconciling Abstraction with High Performance: A MetaOCaml approach

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

O. Kiselyov. *Reconciling Abstraction with High Performance: A MetaOCaml approach*. Foundations and Trends® in Programming Languages, vol. 5, no. 1, pp. 1–101, 2018.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-436-9

Foundations and Trends® in Programming Languages
Volume 5, Issue 1, 2018

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2018, Volume 5, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Foundations and Trends® in Programming Languages
Vol. 5, No. 1 (2018) 1–101

DOI: 10.1561/2500000038

Reconciling Abstraction with High Performance: A MetaOCaml approach

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Contents

Abstract

A common application of generative programming is building high-performance computational kernels highly tuned to the problem at hand. A typical linear algebra kernel is specialized to the numerical domain (rational, float, double, etc.), loop unrolling factors, array layout and a priori knowledge (e.g., the matrix being positive definite). It is tedious and error prone to specialize by hand, writing numerous variations of the same algorithm.

The widely used generators such as ATLAS and SPIRAL reliably produce highly tuned specialized code but are difficult to extend. In ATLAS, which generates code using printf, even balancing parentheses is a challenge. According to the ATLAS creator, debugging is nightmare.

A typed staged programming language such as MetaOCaml lets us state a general, obviously correct algorithm and add layers of specializations in a modular way. By ensuring that the generated code always compiles and letting us quickly test it, MetaOCaml makes writing generators less daunting and more productive.

The readers will see it for themselves in this hands-on tutorial. Assuming no prior knowledge of MetaOCaml and only a basic familiarity with functional programming, we will eventually implement a simple domain-specific language (DSL) for linear algebra, with layers of optimizations for sparsity and memory layout of matrices and vectors, and their algebraic properties. We will generate optimal BLAS kernels. We shall get the taste of the "Abstraction without guilt".

O. Kiselyov. *Reconciling Abstraction with High Performance: A MetaOCaml approach*. Foundations and Trends® in Programming Languages, vol. 5, no. 1, pp. 1–101, 2018.

DOI: 10.1561/2500000038.

1

Introduction

1.1 Why metaprogramming?

Ever-present in all areas of programming is the agonizing trade-off between, on one hand, the maintainable, reusable, easy to read and understand, obviously correct, textbook code – and the code that performs well. The trade-off is exacerbated in high-performance computing (HPC). Coding the matrix-vector multiplication just as `a * v` is clear, portable, self-describing. On the other hand, the typical high-performance code that multiplies an integer-valued matrix to an integer-valued vector takes many, many lines and not at all self-evident. It looks nothing like `a * v`. It also looks nothing like the code that multiplies a single-precision floating-point matrix to a floating-point vector. Which, in turn, bears scarcely any resemblance to the high-performance code multiplying a sparse matrix to a vector.

Already at the end of the last century it was recognized that we can no longer rely on optimizing compilers to turn the high-level code to the high-performance code (see references in Cohen et al. (2006)): many profitable optimizations are domain specific and often narrowly applicable, and hence unlikely to be supported by a general-purpose compiler. Even the simplest replacement `0*e` with `0` is not generally sound: think of e that calls external functions or returns NaN[1]. A domain expert, knowing the input data and the entire algorithm, could tell that the side effects of e may be disregarded or NaNs do not occur – hence the optimization should be carried out, for particular multiplications in particular expressions.

It is cognitively and economically prohibitive for general-purpose compilers to give programmers such minute level of control over optimizations. It is very common therefore for experts to write the computational kernels by hand – and keep re-writing them to accommodate new architectures or new patterns in the input data.

Metaprogramming – code generation specifically – promises a way out: instead of a program we write a program generator, which incorporates domain-specific knowledge and outputs a number of low-level, specialized, high-performance programs. This is the approach taken by the widely known and used fast Fourier transformer generator FFTW (Frigo and Johnson, 2005), basic linear algebra (BLAS) generator ATLAS (Whaley and Petitet, 2005), DSP and linear algebra generator SPIRAL (Püschel et al., 2005), image filter generator Halide (Ragan-Kelley et al., 2013).

The above projects also showed that writing a good generator is still very difficult: it is worth a paper in a prestigious conference. For example, ATLAS – which uses C to generate C code as strings – has been notoriously difficult to write, debug and extend. We need help with code-generating chores – provided by MetaOCaml, Lightweight Modular Staging in Scala (Rompf and Odersky, 2012) or Template Haskell (Sheard and Peyton Jones, 2002). We need levels of abstractions.

Ideally, the end user would write the matrix-vector multiplication *generator* just as a ∗ v. The (domain-specific) operation ∗ would be implemented (perhaps by another programmer, an algorithm designer) using the vocabulary of a different, 'MapReduce' domain:

```
let dot v1 v2 = reduce add zero (zip_with mul v1 v2)
let ( * ) a v = map (dot v) a
```

The generators reduce, add, etc. are to be provided by some other do-

[1] In fact, OCaml before version 4.05 incorrectly performed this optimization: https://github.com/ocaml/ocaml/pull/956.

main expert, a specialist in data layout. An expert in the domain over which matrices and vectors are taken would supply a library of algebraic laws, to invoke to simplify scalar expressions. Eventually it comes to MetaOCaml, to generate code in OCaml or (with offshoring) C or LLVM. This ideal is attainable! In fact, by the end of the tutorial, we shall implement exactly such layered domain-specific language for simple linear algebra. Rompf et al. (2013) and the FEniCS project (Markall et al., 2013) present more examples of such generator DSLs built by composing progressively more detailed abstractions – and their empirical evaluation.

All in all, we do let the end users write programs in the clearest to them form in terms of the familiar domain vocabulary – and yet obtain the high-performance code tuned to various domains. To use Ken Kennedy's phrase, metaprogramming gives us "Abstraction without guilt".

1.2 Why this tutorial?

The goal of the tutorial is to teach how to write typed code generators, how to make them modular, and how to gradually introduce domain-specific optimizations – with MetaOCaml. By the end of the tutorial we will implement a simple domain-specific language (DSL) for linear algebra, with layers of optimizations for the memory layout of matrices and vectors, their sparsity and algebraic properties. We will generate optimal Basic Linear Algebra (BLAS) kernels. Hopefully the readers will see that writing generators is not too complicated and that (staged) types are of great help.

The readers are not expected to know MetaOCaml but should be somewhat familiar with a modern functional language. Even a brief experience with a language in the ML family is a boon. However, Scala or Haskell, etc., programmers should not feel left out.

The present tutorial is by and large a written record of a live tutorial delivered on several occasions (first at CUFP – Commercial Users of Functional Programming 2013). It inherits the hands-on style of those tutorials, built around live coding, in interaction with the MetaOCaml and its type checker – and the audience. We will be developing code piece-by-piece, by submitting small fragments to the MetaOCaml interpreter; fixing the pointed out type problems; generating sample code and testing it; noting the points of improvement and adjusting the generator as needed. The tutorial includes many exercises and homework projects to work on alone or in groups.

1.3 Why MetaOCaml?

We will be using *BER MetaOCaml* (Kiselyov, 2017, 2014), which is a complete re-implementation of the no longer available original MetaOCaml by Walid Taha, Cristiano Calcagno and collaborators (Calcagno et al., 2003).

BER MetaOCaml is a conservative extension of OCaml for "writing programs that generate programs". BER MetaOCaml adds to OCaml the type of *code values* (denoting "program code", or future-stage computations), and two basic constructs to build them: quoting and splicing. The generated code can be printed, stored in a file – or compiled and linked-back to the running program, thus implementing run-time code optimization. MetaOCaml code without staging annotations, or with the annotations erased, is regular OCaml.

MetaOCaml has been successfully used for the most optimal stream fusion (Kiselyov et al., 2017), specializing numeric and dynamic programming algorithms, building FFT kernels, compilers for an image processing and database query DSLs, OCaml server pages, generating families of specialized basic linear algebra and Gaussian Elimination routines, and high-performance stencil computations (Aktemur et al., 2013). See Lengauer and Taha (2006) for a collection of MetaOCaml applications.

Writing code generators in a typed staged language like MetaOCaml benefits in several ways. First, the generated code will be well-formed, with all parentheses matching. Such a guarantee is a dear wish when writing C with printf (as done in ATLAS) or C++ with Matlab. MetaOCaml makes sure that the generated code is well-typed and shall compile without errors. There is no longer puzzling out a compilation error in the generated code, which is typically large, obfuscated and with unhelpful variable names. Mainly, code generation errors are reported in terms of the generator rather than the generated code. The tutorial will give many chances to see the importance of good error reporting.

MetaOCaml generators are hygienic, producing well-scoped code, with no unbound variables. Otherwise, hygiene violations are hard to detect in practice and may lead to the devious error of unintentionally bound variables. Although the unbound variables in the generated code stand out (when compiling it), determining what has caused them proved to be highly non-trivial in practice, as reported by Ofenbeck et al. (2016). The authors wrote a new compiler testing framework, to specifically detect unbound variable and other such problems introduced during refactoring of generators. MetaOCaml is designed to prevent the generation of the problematic code in the first place.

Most importantly, MetaOCaml is typed. Types, staged types in particular, really do help write the code. All throughout the tutorial we will be writing code in live interaction with the type checker – accepting type errors not as a punishment but as a valuable hint. We shall see on many occasions that once we fix the type signature, the generator practically writes itself. The type checker will tell us where to put a staging annotation.

MetaOCaml is purely generative: the generated code is treated as a black box and cannot be examined. One can put code together but cannot take it apart. Pure generativity significantly simplifies the type system and strengthens the static assurances. It may also seem that pure generativity precludes code optimizations. Fortunately, that is not the case, as shall soon see.

The staging annotations of MetaOCaml are like the "assembler" instructions of metaprogramming. We need higher-level abstractions. The final benefit of MetaOCaml – compared to the preprocessors like camlp4 or ppx – is that it is part of OCaml itself, and hence can take the full advantage of OCaml's abstraction and combination facilities, from higher-order functions to modules. Building optimization libraries and composing generators is the stress of the tutorial.

1.4 Overview

The tutorial is based on the progression of problems, which, except the introductory one, are all slightly simplified versions of real-life problems:

1. First steps in staging and MetaOCaml
2. Digital filters
3. Complex vector multiplication: varying data representation (structure of arrays vs. array of structures)
4. Systematic optimization of simple linear algebra: building extensively specialized general BLAS
5. From an interpreter to a compiler: DSL for image manipulation
6. Further challenges (Homework)

In fact, problems 3, 4 and 6 were suggested by HPC researchers as challenges to program generation community (Shonan challenges). The common theme is building high-performance computational kernels highly tuned to the problem at hand. Hence most problems revolve around simple linear algebra – a typical and most frequently executed part in HPC.

The stress on high-performance applications and on modular optimizations and generators sets this tutorial apart from Taha's very accessible, gentle introductions to the 'classical' partial evaluation and staging, focused on turning an interpreter of a generally higher-order language into a compiler (Taha, 2004, 2008). We also get to see this classical area in §6; however, we pay less attention to lambda-calculus and more to image processing. Furthermore, this tutorial mentions recent additions to MetaOCaml such as offshoring and let-insertion.

The source code for the tutorial is available as a supplement: §8.

1.5 Obtaining MetaOCaml

The tutorial needs at least BER MetaOCaml N104, which is available from OPAM

```
opam update
opam switch 4.04.0+BER  # or a later version
eval `opam config env`
```

The MetaOCaml web page http://okmij.org/ftp/ML/MetaOCaml.html talks in depth about the design, implementation and history of MetaOCaml. It also shows other ways of installing it.

References

Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shonan challenge for generative programming: Short position paper. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 147–154, New York, January 2013. ACM Press.

Kenichi Asai. Toward introducing binding-time analysis to MetaOCaml. In PEPM, pages 97–102.

Alan Bawden. Quasiquotation in Lisp. In *PEPM*, number NS-99-1 in Note, pages 4–12. BRICS, January 1999.

BLAS. BLAS: basic linear algebra subprograms. `http://www.netlib.org/blas/`, February 2017.

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *GPCE*, number 2830 in Lecture Notes in Computer Science, pages 57–76, 22–25 September 2003.

Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Science of Computer Programming*, 76(5):349–375, 2011. doi: `10.1016/j.scico.2008.09.008`.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

Albert Cohen, Sébastien Donadio, María Jesús Garzarán, Christoph Armin Herrmann, Oleg Kiselyov, and David A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, September 2006.

Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar N. Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. *New Generation Computing*, 25(3):305–336, 2007. doi: `10.1007/s00354-007-0020-x`.

Andrei P. Ershov. On the partial computation principle. *IPL: Information Processing Letters*, 6(2):38–41, 1977a.

A.P. Ershov. A theoretical principle of system programming. *Doklady AN SSSR (Soviet Mathematics Doklady)*, 18(2):312–315, 1977b.

Daniel Fontijne. Gaigen 2: a geometric algebra implementation generator. In *Proceedings of GPCE 2006: 5th International Conference on Generative Programming and Component Engineering*, pages 141–150, New York, October 2006. ACM Press. doi: `10.1145/1173706.1173728`.

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

Richard W. Hamming. *Digital Filters*. Prentice-Hall, Englewood Cliffs, NJ, 1997, 1983, 1989.

Christoph A. Herrmann and Tobias Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Science of Computer Programming*, 62(1):47–65, September 2006. doi: `10.1016/j.scico.2006.02.002`.

Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. Staged parser combinators for efficient data processing. In *OOPSLA*, pages 637–653. ACM, 2014.

Oleg Kiselyov. Typed tagless final interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming*, SSGIP'10, pages 130–174. Springer-Verlag, Berlin, Heidelberg, 2012. doi: `10.1007/978-3-642-32202-0_3`.

Oleg Kiselyov. The design and implementation of BER MetaOCaml - system description. In *FLOPS*, number 8475 in Lecture Notes in Computer Science, pages 86–102. Springer, 2014. doi: `10.1007/978-3-319-07151-0_6`.

Oleg Kiselyov. BER MetaOCaml N104. `http://okmij.org/ftp/ML/MetaOCaml.html`, January 2017.

Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT*, pages 249–258, 27–29 September 2004.

Oleg Kiselyov, Yukiyoshi Kameyama, and Yuto Sudo. Refined environment classifiers - type- and scope-safe code generation with mutable cells. In Atsushi Igarashi, editor, *APLAS*, volume 10017 of *Lecture Notes in Computer Science*, pages 271–291. Springer-Verlag, 2016. doi: 10.1007/978-3-319-47958-3_15.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 285–299, New York, January 2017. ACM Press. doi: 10.1145/3009837.

Yannis Klonatos, Christoph Koch 0001, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10): 853–864, 2014.

Christian Lengauer and Walid Taha, editors. *MetaOCaml Workshop 2004*, volume 62(1) of *Science of Computer Programming*, September 2006.

Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Loriant, Carlo Bertolli, David A. Ham, and Paul H. J. Kelly. Performance-portable finite element assembly using pyOP2 and FEniCS. In Julian M. Kunkel, Thomas Ludwig 0002, and Hans Werner Meuer, editors, *Supercomputing - 28th International Supercomputing Conference, ISC 2013*, volume 7905 of *Lecture Notes in Computer Science*, pages 279–289. Springer, June 2013.

Mark-Jan Nederhof and Giorgio Satta. Tabular parsing. *Formal Languages and Applications, Studies in Fuzziness and Soft Computing*, 148:529–549, April 05 2004. URL http://arxiv.org/abs/cs/0404009.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. RandIR: differential testing for embedded compilers. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016*, pages 21–30. ACM, October 30 - November 4 2016. doi: 10.1145/2998392.

PEPM. *Proceedings of the 2016 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New York, January 2016. ACM Press.

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93 (2):232–275, 2005.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 519–530. ACM, June 2013.

Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012. doi: 10.1145/2184319.2184345.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL '13: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 497–510, New York, January 2013. ACM Press.

Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. In Manuel M. T. Chakravarty, editor, *Haskell Workshop*, pages 1–16, 3 October 2002. doi: 10.1145/581690.581693.

Julius O. III Smith. Introduction to digital filters with audio applications, September 2007. URL https://ccrma.stanford.edu/~jos/filters/filters.html.

Walid Taha. A gentle introduction to multi-stage programming. In *DSPG 2003*, number 3016 in Lecture Notes in Computer Science, pages 30–50, 2004.

Walid Taha. A gentle introduction to multi-stage programming, part II. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Revised Papers from GTTSE 2007: International Summer School on Generative and Transformational Techniques in Software Engineering II*, number 5235 in Lecture Notes in Computer Science, pages 260–290, Berlin, 2008. Springer-Verlag.

Naoki Takashima, Hiroki Sakamoto, and Yukiyoshi Kameyama. Generate and offshore: type-safe and modular code generation for low-level optimization. In *Proc. ACM SIGPLAN Workshop on Functional High-Performance Computing, FHPC@ICFP 2015, Vancouver, BC, Canada, September 3, 2015*, pages 45–53. ACM, 2015. doi: 10.1145/2808091.

R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software—Practice and Experience*, 35(2):101–121, February 2005.

Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *ML/OCaml*, volume 198 of *EPTCS*, pages 22–63, 2014. URL `http://arxiv.org/abs/1512.01438`.

Jeremy Yallop. Staging generic programming. In PEPM, pages 85–96.