Source-to-Source Optimizations Validated using Separation Logic

Guillaume Bertholon, Arthur Charguéraud, Thomas Koehler

Source-to-Source Optimizations Validated using Separation Logic

Guillaume Bertholon[1,2], Arthur Charguéraud[2,1], and Thomas Kœhler[2,1]

[1]Université de Strasbourg, CNRS, ICube, France
[2]Inria, France

We present a demo of OptiTrust, an interactive framework for optimizing general-purpose programs via series of programmer-guided, source-to-source transformations. Optimization steps are described in transformation scripts, expressed as OCaml programs. At every step, the programmer may interactively visualize the effect of the transformation as the difference between two pieces of human-readable code. The framework currently applies to C code. That said, our internal AST is based on the imperative lambda-calculus, thus we expect it to be straightforward to extend OptiTrust for optimizing OCaml code.

A central question is how to verify that nontrivial transformations preserve the semantics of the code. To that end, we require the programmer to provide Separation Logic annotations, and use them to compute resource usage throughout the code. As we show, this information suffices to justify the correctness of numerous essential source-to-source transformations, such as swapping of instructions, swapping of loops, hoisting of instructions out of loops, etc.

Producing high-performance code is critical in many domains, in particular numerical simulation, image processing and machine learning. Yet, achieving high-performance on modern hardware is a very challenging task.

A common industrial practice is to optimize code *by hand* in languages such as C/C++ [TV14, MLP+17] or Fortran [VD18]. On the one hand, the process of manually rewriting code is highly time-consuming. In particular, several optimization paths must be empirically explored because code performance is hard to assess without benchmarking. On the other hand, the output of manual rewriting is very unsatisfying: the optimized code is much longer than the original, hard to read, hard to maintain, hard to adapt to other hardware, and—worst of all—is highly likely to contain bugs.

Another common industrial practice consists of using DSLs, such as Halide [RKBA+13] or TVM [CMJ+18]. A DSL consists of a programming language restricted to a particular application domain, accompanied with a compiler providing specific optimizations for that language. The main limitation of that approach is that it falls completely short when the programmer requires features that are just outside of the scope of DSL.

OptiTrust is an optimization framework that aims to support optimization of general-purpose code. OptiTrust operates by applying a series of source-to-source transformations, guided by the programmer. The programmer interactively develops a transformation script. At every transformation step, the programmer may visualize the corresponding *diff* between pieces of human-readable code.

Prior work on OptiTrust [CBRB22] has demonstrated its ability to reproduce a state-of-the-art optimized implementation of a numerical Particle-In-Cell simulation, as well as an

```
void mm(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  __modifies("C ⇝ Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __modifies("Group(range(0, n, 1), fun j -> &C[i][j] ⇝ Cell)");
    __seq_reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
    for (int j = 0; j < n; j++) {
      __modifies("&C[i][j] ⇝ Cell");
      __seq_reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __GHOST_BEGIN(focusA, matrix2_ro_focus, "A, i, k");
        __GHOST_BEGIN(focusB, matrix2_ro_focus, "B, k, j");
        sum += A[i][k] * B[k][j];
        __GHOST_END(focusA);
        __GHOST_END(focusB);
      }
      C[i][j] = sum;
    }
  }
}
void mm1024(float* C, float* A, float* B) {
  __reads("A ⇝ Matrix2(1024, 1024), B ⇝ Matrix2(1024, 1024)");
  __modifies("C ⇝ Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}
```

Figure 1. Unoptimized code for matrix multiplication: `A` and `B` denote the input, and `C` the output; and `mm1024` specializes input sizes to 1024.

optimized matrix multiplication kernel generated by the specialized compiler TVM [CMJ+18]. However, code transformations in that prior work were *unchecked*, in the sense that the user could request transformations that do not preserve the semantics of the code.

The present work aims at equipping OptiTrust with means of validating the correctness of the transformations described in the user's scripts. This validation process leverages Separation Logic shape information. To compute this information at every program point, we require the user to provide logical annotations on functions and loops (as well as on function calls with nontrivial instantiation of *auxiliary variables*). These annotations may be viewed either as a weak form of Separation Logic (describing shapes of data structures but not specifying the values stored inside), or as a strong form of typing (more expressive than Rust). As we show, this information suffices to verify transformations such as reordering of instructions or loops, or loop parallelization. Beyond the validation of transformations, another central aspect of our work is to show how the loop annotations can be automatically maintained through a series of transformations affecting the loops. Through our demo, we will highlight the key components of OptiTrust:

1. An internal, simplified abstract syntax tree (AST) of an imperative lambda-calculus, from which readable C code can be recovered throughout transformations. The parsed source code is encoded into this OptiTrust AST, pushing all mutable variables behind a pointer. Then, the AST can be pretty-printed back to C code, using annotations to disambiguate between patterns that are encoded in the same way. A similar idea of a simplified AST has been employed in the Cetus project [DBM+09].
2. An annotation language for lightweight separation logic predicates [Cha20]. These annotations include *function contracts*, *loop contracts*, and *ghost code*. Function contracts specify the ownership and the shape of the data accessible from pointers. Ghost code are annotations that allow to reorganize the view on a data structure. Loops contracts are optional. They can be used in particular to check if loops are parallelizable.
3. A system to compute available separation logic resources at every program point and check specifications. This can be compared to the Rust type system, yet with a more explicit management of aliasing through fractional permissions, or with RefinedC [SLK+21] with less automation but more flexibility thanks to expressive ghost code annotations.

```
void mm1024(float* C, float* A, float* B) {
__modifies("C ~> Matrix2(1024, 1024)");
__reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
for (int i = 0; i < 1024; i++) {
  __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
  __modifies("Group(range(0, 1024, 1), fun j -> &C[i][j] ~> Cell)");
  for (int j = 0; j < 1024; j++) {
    __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
    __modifies("&C[i][j] ~> Cell");
    float sum = 0.f;
    for (int k = 0; k < 1024; k++) {
      __GHOST_BEGIN(focusA, matrix2_ro_focus,
            "M := A, i := i, j := k");
      __GHOST_BEGIN(focusB, matrix2_ro_focus,
            "M := B, i := k, j := j");
      sum += A[i][k] * B[k][j];
      __GHOST_END(focusB);
      __GHOST_END(focusA);
    }
    C[i][j] = sum;
  }
}}
```

```
void mm1024(float* C, float* A, float* B) {
__modifies("C ~> Matrix2(1024, 1024)");
__reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
for (int i = 0; i < 1024; i++) {
  __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
  __modifies("Group(range(0, 1024, 1), fun j -> &C[i][j] ~> Cell)");
  __GHOST_BEGIN(tile_divides,
      "tile_count := 32, tile_size := 32, n := 1024, to_item := "
      "fun j -> &C[i][j] ~> Cell, bound_check := checked");
  for (int bj = 0; bj < 32; bj++) {
    __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
    __modifies(
        "Group(range(0, 32, 1), fun j -> &C[i][bj * 32 + j] ~> Cell)");
    for (int j = 0; j < 32; j++) {
      __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
      __modifies("&C[i][bj * 32 + j] ~> Cell");
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        __GHOST_BEGIN(focusA, matrix2_ro_focus,
              "M := A, i := i, j := k");
        __GHOST_BEGIN(focusB, matrix2_ro_focus,
              "M := B, i := k, j := bj * 32 + j");
        sum += A[i][k] * B[k][bj * 32 + j];
        __GHOST_END(focusB);
        __GHOST_END(focusA);
      }
      C[i][bj * 32 + j] = sum;
    }
  }
  __GHOST_END(divides);
}}
```

Figure 2. Transformation: `Loop.tile (int 32)~index:"bj"~bound:TileDivides [cFor "j"]`. Replace the loop on `j` of 1024 iterations, with two nested loops on `bj` and `j` of 32 iterations each. The inserted operation `tile_divides` is a ghost instruction.

4. A system for targeting program points, somewhat similar to XPath [CD+99] for XML, but specialized for an AST. This system allows to describe, in a concise and robust manner, one or several program points to transform.
5. A library of general-purpose transformations that can leverage resources computed at each program point to justify their correctness, including: core data layout transformations [SSH10], instruction-level transformations [AK02], control flow transformations [Wol95]. Transformation also preserve or adapt annotations on functions and loops, and insert ghost operations if needed.
6. A scripting language, embedded in OCaml, for describing transformations. Transformation scripts are a classic technique for programmer-guided optimization frameworks [BHRS08, BZHB16, NS16, ZCG18, CCH08, RKH+11, BC20, YQ08, YWC14]. Transformation scripts provide fine-grained control, and they allow to chain large numbers of transformation steps.
7. An interactive interface allowing to visualize code *diffs* associated with the transformation at a given line of the script, via a key shortcut in the code editor.

Our running example will consist of an optimization script for a matrix multiplication function. Figure 1 shows the input code, including its annotations. In particular, each function and each loop has a contract (`seq_reads(H)` indicates that every loop iteration reads the full resource `H`, whereas `reads(Hi)` indicates that only the `i`-th iteration reads the resource `Hi`.) We will explain how to annotate the C code with our lightweight Separation Logic predicates, and explain how OptiTrust type-checks the code to compute resources at every program point. We will then present the transformation script, and explain how transformations are implemented. Figure 2 shows the example of a *tiling transformation*. The transformation does not improve performance by itself but allows for efficient parallelization schemes in a subsequent transformation (like using SIMD instructions for the inner loop).

We hope to engage with the JFLA community, in particular on the benefits of exploiting lightweight Separation Logic, as well as on potential applications of source-to-source transformations on C and OCaml programs.

References

[AK02] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. 2002.

[BC20] João Bispo and João MP Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, 2020.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, 2008.

[BZHB16] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening Polyhedral Compiler's Black Box. In *IEEE/ACM International Symp. on Code Generation and Optimization*, March 2016.

[CBRB22] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. working paper, September 2022.

[CCH08] Chun Chen, Jacqueline Chame, and Mary W. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Jun 2008.

[CD+99] James Clark, Steve DeRose, et al. XML path language (xpath), 1999.

[Cha20] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[MLP+17] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6, 2017.

[NS16] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *Static Analysis - 23rd International Symposium, SAS*, volume 9837 of *LNCS*, 2016.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Conf. on Programming Language Design and Implementation*, 2013.

[RKH+11] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Languages and Compilers for Parallel Computing*, 2011.

[SLK+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: Automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conf. on Programming Language Design and Implementation*, pages 158–174, 2021.

[SSH10] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. PACT '10, pages 513–522, 2010.

[TV14] Ashkan Tousimojarad and Wim Vanderbauwhede. Comparison of three popular parallel programming models on the intel xeon phi. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20*, pages 314–325. Springer, 2014.

[VD18] Wim Vanderbauwhede and Gavin Davidson. Domain-specific acceleration and auto-parallelization of legacy scientific code in fortran 77 using source-to-source compilation. *Computers & Fluids*, 173:1–5, 2018.

[Wol95] M. Wolfe. *High performance compilers for parallel computing*. 1995.

[YQ08] Qing Yi and Apan Qasem. Exploring the optimization space of dense linear algebra kernels. In *LCPC*, 2008.

[YWC14] Qing Yi, Qian Wang, and Huimin Cui. Specializing compiler optimizations through programmable composition for dense matrix computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 596–608, USA, 2014.

[ZCG18] Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. Declarative Transformations in the Polyhedral Model. Research Report RR-9243, December 2018.