OptiTrust: Producing Trustworthy High-Performance Code via Source-to-Source Transformations [DRAFT]

GUILLAUME BERTHOLON, ARTHUR CHARGUÉRAUD, THOMAS KŒHLER, BEGATIM BYTYQI, and DAMIEN ROUHLING, Inria & Université de Strasbourg, CNRS, ICube, France

Developments in hardware have delivered formidable computing power. Yet, the increased hardware complexity has made it a real challenge to develop software that exploits hardware to its full potential. Numerous approaches have been explored to help programmers turn naive code into high-performance code, finely tuned for the targeted hardware. However, these approaches have inherent limitations, and it remains common practice for programmers seeking maximal performance to follow the tedious and error-prone route of writing optimized code by hand.

This paper presents OptiTrust, an interactive source-to-source optimization framework. The programmer develops a script describing a series of code transformations. The framework provides continuous feedback in the form of human-readable *diff* s over conventional C code. OptiTrust supports advanced code transformations, including transformations exploited by the state-of-the-art DSL tools Halide and TVM, and transformations beyond the reach of existing tools. OptiTrust also supports user-defined transformations, as well as defining complex transformations by composition of simpler transformations.

Crucially, to check the validity of code transformations, OptiTrust leverages a *static resource analysis* in a simplified form of Separation Logic. Our analysis exploits user-provided annotations on functions and loops, and deduces precise resource usage throughout the code. Through three representative case studies, we demonstrate how OptiTrust can be employed to produce state-of-the-art, high-performance programs.

1 INTRODUCTION

1.1 Motivation

Performance matters in numerous fields of computer science, and in particular in applications from machine learning, computer graphics, and numerical simulation. Massive speedups can be achieved by fine-tuning the code to best exploit the available hardware [Kelefouras and Keramidas 2022]. Between a naive implementation and an optimized implementation, it is common to see a speedup of the order of 50×, on a single core. For many applications, the code can then be accelerated further by one or two orders of magnitude by exploiting multicore parallelism or GPUs.

Yet, producing high performance code is hard. Over the past decades, nontrivial mechanisms with subtle interactions were integrated into hardware architectures. Reasoning about performance requires reasoning about the effects of multiple levels of caches, the limitations of memory bandwidth, the intricate rules of atomic operations, and the diversity of vector instructions (SIMD). These aspects and their interactions make it challenging to build cost models. For example, the cost of a memory access can range from one CPU cycle to hundreds of CPU cycles, depending on whether the corresponding data is already in cache. In the general case, accurately modeling cache behavior requires a deep understanding of the algorithm and hardware at play.

Accurately predicting runtime behavior is challenging for expert programmers, and appears beyond the capabilities of automated tools. Therefore, compilers struggle to navigate the exponentially large search space of all possible code candidates [Vachharajani et al. 2003], resorting to best-effort heuristics, and often failing to produce competitive code [Barham and Isard 2019].

Today, it remains common practice in industry for programmers to write optimized code *by hand* [Amaral et al. 2020; Evans et al. 2022]. However, manual code optimization is unsatisfactory for at least three reasons. First, manually implementing optimized code is time-consuming. Second,

Authors' address: Guillaume Bertholon; Arthur Charguéraud, arthur.chargueraud@inria.fr; Thomas Kœhler; Begatim Bytyqi; Damien Rouhling, Inria & Université de Strasbourg, CNRS, ICube, France.

	Halide/TVM	Elevate+Rise	Exo	Clay/LoopOpt	ATL	Alpinist	Clava+LARA
Generality	◔	◐	◐	◐	◐	◕	●
Expressiveness	●	●	●	◐	◐	◔	◔
Control	◐	◐	◕	◐	◐	●	●
Feedback	◐	◐	●	●	◐	●	◐
Composability	○	●	◐	◐	●	○	●
Extensibility	○	●	●	○	●	●	●
Trustworthiness	◐	◐	◐	◐	●	●	○

Table 1. Overview of user-guided tools for high-performance code generation. Darker is better.

the optimized code is hard to maintain through hardware and software evolutions. Third, the rewriting process is error-prone: not only every manual code edition might introduce a bug, but the code complexity also increases, especially when introducing parallelism. These three factors are exacerbated by the fact that optimizations typically make code size grow by an order of magnitude (Section 2 contains examples).

In summary, neither fully automatic nor fully manual approaches are satisfying for generating high performance code. *Semi-automatic code optimization* aims at combining the benefits of machine automation with the strength of human insight. Before reviewing tools for semi-automatic code optimization, let us introduce a number of qualitative properties on which to evaluate these tools.

- **Generality**: How large is the domain of applicability of the tool? In particular, is it restricted to a domain-specific language (DSL)?
- **Expressiveness**: How advanced are the code transformations supported by the tool? Is it possible to express state-of-the-art code optimizations?
- **Control**: How much control over the final code is given to the user by the tool? In particular, is there a monolithic code generation stage?
- **Feedback**: Does the tool provide easily readable intermediate code after each transformation?
- **Composability**: Is it possible to define transformations as the composition of existing transformations? Can transformations be higher-order, i.e., parameterized by other transformations?
- **Extensibility** of transformations: Does the tool facilitate defining custom transformations that are not expressible as the composition of built-in ones?
- **Trustworthiness**: Does the tool ensure that user-requested transformations preserve the semantics of the code? Can it moreover provide mechanized proofs?

There exists other properties for optimization tools, such as the ease of integration in an existing code base, the maintainability of optimized code, or the steepness of the learning curve for new users. These are certainly important aspects, yet they are harder to evaluate objectively. Hence, we omit them from the discussion, and focus on the aforementioned technical properties.

1.2 Closely Related Work

Table 1 summarizes the properties of existing approaches, highlighting their diversity. For the tools considered, generality appears negatively correlated with expressiveness, i.e., with how advanced the supported transformations are. For each property considered, at least two tools show strengths on that property. However, even if we leave out the ambition of achieving mechanized proofs, each tool considered shows weaknesses on at least two properties. Hence, it appears that there remains a lot of room for improvement. Before presenting the contribution of the OptiTrust framework, we first describe the tools listed in the table.

Halide [Ragan-Kelley et al. 2013] is an industrial-strength domain-specific compiler for image processing, used e.g. to optimize code running in Photoshop and YouTube. Halide popularized the idea of separating an *algorithm* describing what to compute from a *schedule* describing how to optimize the computation. This separation makes it easy to try different schedules. TVM [Chen et al. 2018] is a tool directly inspired by Halide, but tuned for machine learning applications; it is used by most of the major CPU/GPU manufacturers. Other tools inspired by Halide include Fireiron [Hagedorn et al. 2020a], used at Nvidia, as well as PartIR [Alabed et al. 2024], used at Google. All these tools are inherently limited to the domains (DSLs) that they target. They do not support higher-order composition of transformations, and are not extensible [Barham and Isard 2019; Ragan-Kelley 2023]. Moreover, understanding their output is difficult as the applied transformations are not detailed to the user, even though interactive scheduling systems have been proposed to mitigate this difficulty [Ikarashi et al. 2021].

Elevate [Hagedorn et al. 2020b] is a functional language for describing *optimization strategies* as composition of simple *rewrite rules*. Advanced optimizations from TVM and Halide can be reproduced using Elevate. One key benefit is extensibility: adding rewrite rules is much easier than changing complex and monolithic compilation passes [Ragan-Kelley 2023]. Elevate strategies are applied on programs expressed in a functional array language named Rise, followed by compilation to imperative code. The use of a functional array language greatly simplifies rewriting, however it restricts applicability and makes controlling imperative aspects difficult (e.g. memory reuse).

Exo [Ikarashi et al. 2022] is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. The strength of Exo lies in externalizing target-specific code generation to user-level code instead of compilation backends. Exo programs can be optimized by applying a series of source-to-source transformations. These transformations are described in a Python script, with simple string-based patterns for targeting code points. The user can add custom transformations, possibly defined by composition; higher-order composition seems possible but has not yet been demonstrated. A major limitation of Exo is that it is restricted to static control programs with linear integer arithmetic.

Clay [Bagnères et al. 2016a] is a framework to assist in the optimization of loop nests that can be described in the *polyhedral model* [Feautrier 1992]. The polyhedral model only covers a specific class of loop transformations, with restriction over the code contained in the loop bodies, however it has proved extremely powerful for optimizing code falling in that fragment. Clay provides a decomposition of polyhedral optimizations as a sequence of basic transformations with integer arguments. The corresponding transformation script can then be customized by the programmer. Clint [Zinenko et al. 2018b] adds visual manipulation of polyhedral schedules through interactive 2D diagrams. LoopOpt [Chelini et al. 2021] provides an interactive interface that helps users design optimization sequences (featuring unrolling, tiling, interchange, and reverse of iteration order) that can be bound in a declarative fashion to loop nests satisfying specific patterns.

ATL [Liu et al. 2022] is a purely functional array language for expressing Halide-style programs. Its particularity is to be embedded into the Coq proof assistant. ATL programs can be transformed through the application of rewrite rules expressed as Coq theorems. With this approach, transformations are inherently accompanied by machine-checked proofs of correctness. The set of rules includes expressive transformations, some beyond the scope of Halide, and can be extended by the user. Once optimized, ATL programs are then compiled into imperative C code. Like Rise, generality and control are restricted by the functional array language nature of ATL.

Alpinist [Sakar et al. 2022] is a *pragma*-based tool for optimizing GPU-level, array-based code. It is able to apply basic transformations such as loop tiling, loop unrolling, data prefetching, matrix linearization, and kernel fusion. The key characteristic of Alpinist is that it operates over code formally verified using the VerCors framework [Blom et al. 2017]. Concretely, Alpinist transforms not only the code but also its formal annotations. If Alpinist were to leverage transformation scripts instead of pragmas, it might be possible to chain and compose transformations; yet, this possibility remains to be demonstrated.

Clava [Bispo and Cardoso 2020] is a general-purpose C++ source-to-source analysis and transformation framework implemented in Java. The framework has been instantiated mainly for code instrumentation purpose and auto-tuning of parameters. Clava can also be used in conjunction with a DSL called LARA [Silvano et al. 2019] for optimizing specific programs. LARA allows expressing user-guided transformations by combining declarative queries over the abstract syntax tree and imperative invocations of transformations, with the option to embed JavaScript code. The application paper on the Pegasus tool [Pinto et al. 2020] illustrates this approach on loop tiling and interchange operations.

1.3 Contribution

This paper introduces OptiTrust, the first interactive optimization framework that operates at the level of C syntax, and that supports and validates state-of-the-art optimizations. OptiTrust is open-source and available at the URL: https://github.com/charguer/optitrust.

Overview. In OptiTrust, the user starts from an unoptimized C code, and develops a *transformation script* describing a series of optimization steps. Each step consists of an invocation of a specific transformation at specified *targets*. OptiTrust provides an expressive target mechanism for describing, in a concise and robust manner, one or several code locations. On any step of the transformation script, the user can press a key shortcut to view the *diff* associated with that step, in the form of a comparison between two human-readable C programs. Concretely, a transformation script consists of an OCaml program linked against the OptiTrust library of transformations.

A central aspect of OptiTrust is that it guarantees that the code transformations requested by the programmer preserve the semantics of the program. To that end, OptiTrust leverages our *static resource analysis*, which concretely takes the form of a type checking algorithm in a type system featuring linear resources. This type system may be thought of as a variant of the Rust type system, or as a scaled down version of Separation Logic [Reynolds 2002].

For type-checking resources, functions and loops need to be equipped with *contracts* describing their resource usage. These contracts may be inserted either directly as no-op annotations in the C source code, or they may be inserted by dedicated commands as part of the transformation script. OptiTrust is able to automatically infer simple loop contracts, thus not all loops need to be annotated manually. Crucially, every OptiTrust transformation takes care of updating contracts in order to reflect changes in the code. In other words, a well-typed program must remain well-typed after a successful transformation. This property is essential to ensure that subsequent transformations in the optimization chain can be validated by exploiting information from our resource analysis.

The implementation of OptiTrust distinguishes between *basic* transformations and *combined* transformations. On the one hand, a basic transformation applies minimalistic changes to the abstract syntax tree (AST). The validity of a basic transformation is checked by leveraging the resource analysis. On the other hand, a combined transformation is implemented as a composition of basic transformations. Combined transformations aim to implement high-level strategies, that may trigger the execution of dozens of basic transformation. These more complex combined transformations need not be accompanied with code for checking validity: their validity is guaranteed by the validity checks performed by the basic transformations. This two-layer approach enables us to minimize the size of the trusted code base (TCB) of OptiTrust.

Internally, OptiTrust does not manipulate an abstract syntax tree (AST) for C syntax, but an AST following a presentation closer to an imperative λ-calculus. We have developed a bidirectional translation between the C language, which we parse using Clang, and the OptiTrust internal language. In particular, our translation is able to eliminate—and subsequently reintroduce—mutable variables and operations involving l-values. Considering a syntax and semantics simpler than that of C considerably helps to tame the complexity of the design and implementation of typing rules, code transformations, and correctness criteria associated with transformations.

Limitations. In the long term, our aim is for OptiTrust to perform full-score on all the aforementioned evaluation criteria. On the way towards this highly ambitious goal, we have considered three simplifications that apply to the work described in the present paper.

(1) We restrict ourselves to a subset of the C language. As our case studies show, this subset nevertheless suffices to express numerous practical, high-performance programs, in an idiomatic programming style both for the unoptimized and for the optimized code. Moreover, for simplicity, we ignore complications related to arithmetic overflows.
(2) We have already implemented more than a hundred transformations, among the most standard ones. These transformations suffice to assess the interest of the OptiTrust approach to code optimization. However, for production usage, we believe there remains a couple hundreds additional transformations to implement.
(3) We restrict ourselves to a subset of Separation Logic. Our resource-based type system is able to describe the ownership of arrays, matrices, or individual cells, however it does not allow specifying properties about the values stored in data structures. Nevertheless, as our case studies show, shape-based resources suffice to justify the correctness of many practical code optimization patterns.

In the long term, our resource-based system aims to be similar in spirit to RefinedC [Sammler et al. 2021], a Separation Logic-based type system for C code. The effectiveness of Separation Logic has been successfully demonstrated across a broad range of applications, both for low-level and high-level code [Charguéraud 2020a; O'Hearn 2019]. By building OptiTrust on Separation Logic, we are confident that our framework has the potential to be generally applicable.

In summary, we present a framework that can readily be exploited to optimize certain classes of programs, and acknowledge that future work remains necessary to achieve full generality. Note that we have taken great care in our design to anticipate for the extension to a richer programming language and to a richer Separation Logic.

1.4 Contents of the Paper

We first present the features of OptiTrust by means of example, in Section 2. Then, we present the construction of OptiTrust in five parts. In Section 3, we describe the abstract syntax tree used internally by OptiTrust. In Section 4, we explain the core of a resource-based typechecker. This part presents relatively standard Separation Logic concepts, but following an algorithmic rather than a declarative presentation of the reasoning rules. In Section 5, we explain our key addition to the typechecker, namely the computation of *usage information* for every resource and for every subterm. In Section 6, we present a set of representative code transformations, illustrating in particular how usage information is exploited to justify the correctness of these transformations. Finally, we discuss additional related work in Section 7.

2 OPTITRUST IN PRACTICE

Let us present the features of OptiTrust through three case studies. In Section 2.1, we reproduce a manually written code from OpenCV—a very popular, highly optimized computer vision library. In Section 2.2, we consider a physics simulation program featuring a kernel typical of particle

```
void rowSum(const int n, const int cn, const int kn, const T S[n+kn][cn], ST D[n][cn]) {
  for (int c = 0; c < cn; c++) { // for each channel (e.g., red, green, and blue)
    for (int i = 0; i < n; i++) { // for each target pixel in the row described by D
      ST s = 0;
      for (int j = i; j < i+kn; j++) // for each source pixel nearby to the right
        s += (ST) S[j][c];
      D[i][c] = s;
} } }
```

Fig. 1. Unoptimized C code for the OpenCV case study, using multi-dimensional arrays.

simulations; we demonstrate how to apply, using OptiTrust, several optimizations that are ubiquitous in this kind of applications. In Section 2.3, we reproduce an optimized implementation of matrix-multiply, similar to the one produced by TVM, the state-of-the-art specialized compiler for machine learning applications. Then, in Section 2.4, we evaluate OptiTrust against the desirable properties for semi-automatic code optimization frameworks.

2.1 The OpenCV Case Study

In image processing, *blur* is typically used to remove noise and smoothen images. Blur can be decomposed as a combination of *column-based blur*, *row-based blur*, and (optionally) the application of a normalization pass. Our case study focuses on the *row-based blur* function, as implemented in the state-of-the-art OpenCV library.

Unoptimized code. If performance was not a concern at all, the row-based blur function would be implemented as shown in Fig. 1. The output is a single-row image, stored in an array named `D`, made of `n` pixels. The input is a single-row image, stored in an array named `S`, made of `n+kn` pixels. The parameter `kn` is a parameter of the transformation—the width of the blur. Each pixel is encoded using `cn` integers. Practical values include `cn=1` for grayscale, `cn=3` for RGB, `cn=4` for RGBA. The input pixels are encoded on `cn` integers of type `T`, whereas the output pixels are encoded on `cn` integers of type `ST`. Typically, the type `ST` is represented on more bits than the type `S`. The output pixel `D[i]` is computed as the sum of the values of the input pixels in the range from `S[i]` to `S[i+kn-1]`. This sum is computed independently for each of the `cn` color channels.

Multi-dimensional vs flat arrays. The code from Fig. 1 is presented using C syntax for multi-dimensional arrays, for the sake of improved readability. However, the optimized code from Fig. 3 and our contract-annotated code from Fig. 2 instead use a flat array representation. The flat representation is idiomatic in high-performance code: it allows performing simplifications in array accesses, moreover it allows for compatibility with C++ parsers. For technical reasons, and to anticipate for extensions of OptiTrust, OptiTrust relies on a C++ parser. We leave to future work the parsing of multi-dimension arrays and their elimination via a source-to-source transformation.

Optimized code. The handwritten OpenCV library includes an implementation of row-sum blur structured like the code shows in Fig. 2. The original OpenCV code may be viewed online.[1] The code from Fig. 2 corresponds to the code that we produce using our OptiTrust. The only notable difference between the OpenCV code and ours is that OpenCV traverses certain arrays by incrementing pointers, whereas we use explicit array indexing everywhere. In general, for this type of code, array indexing gives equivalent or better performance than pointer shifting; we leave

[1] https://github.com/opencv/opencv/blob/4.10.0/modules/imgproc/src/box_filter.simd.hpp#L75: The OpenCV code is implemented as class with the types S and ST as template arguments, whereas for the moment our code refers to fixed yet unspecified integer types; we look forward to add support for template polymorphism in the future.

```c
void rowSum(const int n, const int cn,
            const int kn, const T* S, ST* D) {
  if (kn == 3) {
    for (int ic = 0; ic < n * cn; ic++) {
      D[ic] = (ST) S[ic]
            + (ST) S[cn + ic]
            + (ST) S[2 * cn + ic];
    }
  } else if (kn == 5) {
    for (int ic = 0; ic < n * cn; ic++) {
      D[ic] = (ST) S[ic]
            + (ST) S[cn + ic]
            + (ST) S[2 * cn + ic]
            + (ST) S[3 * cn + ic]
            + (ST) S[4 * cn + ic];
    }
  } else if (cn == 1) {
      ST s = (ST) 0;
      for (int i = 0; i < kn; i++) {
        s += (ST) S[i];
      }
      D[0] = s;
      for (int i = 1; i < n; i++) {
        s += (ST) S[-1 + i + kn]
          - (ST) S[-1 + i];
        D[i] = s;
      }
  } else if (cn == 3) {
    ST s0 = (ST) 0;
    ST s1 = (ST) 0;
    ST s2 = (ST) 0;
    for (int i = 0; i < 3*kn; i+=3) {
      s0 += (ST) S[i];
      s1 += (ST) S[1 + i];
      s2 += (ST) S[2 + i];
    }
    D[0] = s0;
    D[1] = s1;
    D[2] = s2;
    int p = 3 * kn;
    for (int i = 3; i < p; i += 3) {
      s0 += (ST) S[-3 + i + p] - (ST) S[-3 + i];
      s1 += (ST) S[-2 + i + p] - (ST) S[-2 + i];
      s2 += (ST) S[-1 + i + p] - (ST) S[-1 + i];
      D[i] = s0;
      D[i + 1] = s1;
      D[i + 2] = s2;
    }
  } else if (cn == 4) {
    // [...] similar to cn == 3, with one more variable
  } else {
    int p = cn * kn;
    for (int c = 0; c < cn; c++) {
      ST s = (ST) 0;
      for (int i = 0; i < p; i += cn) {
        s += (ST) S[i + c];
      }
      D[c] = s;
      for (int i = cn; i < p; i += cn) {
        s += (ST) S[-cn + i + p + c] - (ST) S[-cn + i + c];
        D[i + c] = s;
} } } }
```

Fig. 2. Our optimized C code for the OpenCV case study, showing the body of the `rowSum` function. This code exploits essentially the same optimizations as the original OpenCV code.

it to future work to extend OptiTrust with a transformation able to introduce pointer shifting over a loop.

The optimized implementation is a *multi-versioned* code, with dedicated execution paths for handling specific values of the parameters. The branches `kn == 3` and `kn == 5` correspond to values of `kn`, the width that are commonly used by library users. For these small constant values of `kn`, the inner loop on `j` from Fig. 2 can be unfolded. Otherwise, when the loop on `j` is not unfolded, a standard algorithmic optimization called *sliding window* can be applied. Note that Halide, the state-of-the-art specialized compiler for image processing, does not support the introduction of sliding windows—and the developers of Halide do not plan to lift this limitation.[2] When using Halide, a programmer either needs to manually refine the code to introduce the sliding window before optimizing the code; or needs to exploit other transformation tools specialized for applying sliding window optimizations [Chaurasia et al. 2015; Kanetaka et al. 2024].

The branch of the code that uses the sliding window optimization is then further specialized with branches for commonly used parameters: `cn == 1` and `cn == 3` and `cn == 4`. For these small constant values of `cn`, the outer loop on `c` can be unfolded, then the multiple occurrences of the loop on `i` that result from this unfolding can be fused into a single loop. The final else-branch in the code from Fig. 2 corresponds to the generic implementation.

Annotated Unoptimized Code. Before we can start optimizing the code from Fig. 1 using OptiTrust, we need to annotate the code with *function contracts*, *loop contracts*, as well as *ghost instructions*, in

[2] https://github.com/halide/Halide/issues/180

```
void rowSum(const int kn, const T* S, ST* D, const int n, const int cn) {
  __requires("kn >= 0, n >= 1, cn >= 0");
  __reads("S ~> Matrix2(n+kn, cn)");
  __modifies("D ~> Matrix2(n, cn)");
  __ghost_begin("unswap1", swap_groups, "items := fun i, c -> &D[i][c] ~> Cell");
  for (int c = 0; c < cn; c++) { // for each channel
    __xmodifies("for i in 0..n -> &D[MINDEX2(n, cn, i, c)] ~> Cell");
    for (int i = 0; i < n; i++) { // for each pixel
      __xmodifies("&D[MINDEX2(n, cn, i, c)] ~> Cell");
      __ghost(assume, "is_subrange(i..i + kn, 0..n + kn)");
      ST s = 0;
      for (int j = i; j < i+kn; j++) {
        s += (ST) S[MINDEX2(n+kn, cn, j, c)];
      }
      D[MINDEX2(n, cn, i, c)] = s;
    }
  }
  __ghost_end("unswap1");
}
```

Fig. 3. Unoptimized C code for the OpenCV case study, using flat arrays and augmented with resource annotations.

order to guide OptiTrust's resource-based typechecker. Fig. 3 shows the same C code as in Fig. 1 augmented with a number of no-ops that provide the relevant annotations.[3]

The clause `__requires` contains assumptions about the input parameters. The clause `__reads` asserts that input array `S` can be accessed in read-only mode. The clause `__modifies` asserts that output array `D` can be modified in place. The syntax `__modifies("D")` is a shorthand for `__modifies("D ~> Matrix2(n,cn)")`. The latter explicit form would be useful to provide the dimensions if the argument of the function was declared as `ST* D` instead of `ST D[n][cn]`, as it is the case for example in the optimized code. The clause `__xmodifies` describes a *loop contract*: it indicates not only that the `i`-th iteration can modify the cell `D[i][c]`, but also that the other iterations do not access this cell. In other words, the `i`-th iteration has exclusive access to that cell. The "`x`" prefix in `__xmodifies` stands for "exclusive".

The lines introduced by `__ghost_begin`, `__ghost_end`, or sometimes just `__ghost` correspond to *ghost instructions*: no-ops whose purpose is to change the view on the resources. The need for ghost instructions is standard in Separation Logic frameworks. The specialized keywords `__ghost_begin` and `__ghost_end` materialize a pair of ghost instructions that are the reciprocal of one another. The `__ghost_begin` pseudo-instruction in Fig. 3 refines the view on the array `D` from $\bigstar_{i \in 0..n} \bigstar_{c \in 0..cn}$ `&D[i][c] ~> Cell` to $\bigstar_{c \in 0..cn} \bigstar_{i \in 0..n}$ `&D[i][c] ~> Cell`, where the star symbol corresponds to the *iterated separating conjunction* operator of Separation Logic. Intuitively, it swaps two "logical loops" that iterate over the sets of memory cells covered by the array D. The matching `__ghost_end` pseudo-instruction applies the symmetrical operation, reverting the view on the resources back to the original form.

Optimization script. Fig. 4 shows our optimization script for generating the optimized code of Fig. 2 starting from the annotated unoptimized code of Fig. 3. In OptiTrust, optimizations are dictated by means of a script written in the OCaml programming language. For the reader not familiar with OCaml, `f x y` denote the call of `f` on the arguments `x` and `y`; the symbol `~` is used

[3]Alternatively, one could use *attributes* to attach certain annotations to language constructs, e.g., contracts to loops and functions. We believe, however, that representing ghost instructions as no-op instructions is more natural, because a ghost instruction is part of a sequence without being attached to either the previous or the subsequent instruction.

```
Specialize.variable_multi ~mark_then:fst ~mark_else:"nokn"
  ["kn", int 3; "kn", int 5] [cFunBody "rowSum"; cFor "c"];
Loop.unroll [nbMulti; cMark "kn"; cFor "c"];
Instr.gather_targets [nbMulti; cMark "kn"; cArrayWrite "D"];
Loop.sliding_window [cMark "nokn"; cFor "j"];
Loop.swap [nbMulti; cMark "nokn"; cFor "c"];
Specialize.variable_multi ~mark_then ~mark_else:"nocn"
  ["cn" int 1; "cn", int 3; "cn", int 4] [cMark "nokn"];
Loop.unroll [nbMulti; cMark "cn"; cFor "c"];
Loop.collapse [nbMulti; cMark "cn"; cFor "i"];
Target.foreach [nbMulti; cMark "cn"] (fun c ->
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "i" ~body:[cArrayWrite "D"]];
  Instr.gather_targets [c; cStrict; cArrayWrite "D"];
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor ~stop:[cVar "kn"] "i"];
  Instr.gather_targets [c; cFor "i"; cArrayWrite "D"];
);
Cleanup.std [];
```

Fig. 4. Optimization script for the OpenCV case study. [TODO: finalize]

to provide optional (or named) arguments; `[x; y; z]` denotes a list; `(x, y, z)` denotes a tuple; `s1 ^ s2` denotes a string concatenation; and `let f x = e1 in e2` introduces a local function f.

A transformation script generally consists of a series of calls to functions from the OptiTrust library. By convention, the last argument of a transformation always denotes a *target*. A target provides a way to concisely and robustly refer to one or several code locations. A target consists of a list of constraints (prefixed by "c") that is satisfied by code paths that go through nodes satisfying each constraint, in the given order. For example, the constraint `cFunBody "rowSum"` requires visiting a function definition with the name `"rowSum"`. The constraint `cFor "c"` requires visiting a for loop over an index with the name c. The constraint `cMark "cn"` requires visiting an AST node that carries the mark `"cn"`. Such marks are introduced by transformations, on demand of the programmer.

Targets may also be given as arguments to constraints: for example, `cFor "i"~body:[cArrayWrite "D"]` requires visiting a for loop over an index with the name `"i"`, whose body also contains a write on the array `D`. Targets may also include special modifiers. The modifier `nbMulti` in a target indicates that the programmer expects to find not one but multiple AST nodes that match this target. For example, the modifier `tBefore`, which appears in the other two case studies, allows targeting the interstice before an instruction. Finally, the special command `Target.set_ctx` registers a partial target to be used as prefix for all subsequent targets. This command enables the subsequent instructions of the script to focus on a particular piece of the program.

[TODO: comment a little bit more on the contents of the script when it is finalized.]

Interactive Visualization. Each step of an evaluation script may be executed interactively: with the cursor on a line, the OptiTrust user can press a shortcut key in their code editor to visualize the *diff* associated with the transformation on that line. All intermediate versions of the code consist of human-readable, executable C code. [TODO: insert an example diff, for example showing loop fusion.] Additionally, OptiTrust can produce a complete *execution trace* in the form of an interactive tree. This tree reports the *diff* not only for every transformation visible in the script, but also for all the internal transformations that are leveraged in the process. We encourage the reader to navigate the traces associated with our case studies.[4]

[4]The traces associated with our case studies may be found at: https://www.chargueraud.org/softs/optitrust/traces/index.html

```
void simulate(const vect* fieldAtCorners,
 const int nbSteps, const double deltaT,
 const double pCharge, const double pMass,
 const int nbParticles, particle* particles) {
  __reads("fieldAtCorners ~> Array(8)");
  __modifies("particles ~> Array(nbParticles)");
  for (int idStep = 0; idStep < nbSteps; idStep++) {
    for (int idPart = 0; idPart < nbParticles; idPart++) {
      // Each particle is updated at each time step
      __xmodifies("&particles[idPart] ~> Cell");
      const particle p = particles[idPart];
      // Interpolate the field based on the position relative to the corners of the cell
      double const coeffs[8];
      compute_corner_interpolation_coeffs(p.pos, coeffs);
      const vect fieldAtPos = matrix_vect_mul(coeffs, fieldAtCorners);
      // Compute the acceleration: F = m*a and F = q*E gives a = q/m*E
      const vect accel = vect_mul(pCharge / pMass, fieldAtPos);
      // Compute the new speed and position for the particle
      const vect speed2 = vect_add(p.speed, vect_mul(deltaT, accel));
      const vect pos2 = vect_add(p.pos, vect_mul(deltaT, speed2));
      // Update the particle
      particles[idPart].speed = speed2;
      particles[idPart].pos = pos2;
} } }
```

Fig. 5. Unoptimized code for the particle simulation case study, with resource annotations.

Validity Checks. The evaluation of the transformation script from Fig. 4 triggers the application of a chain of basic transformations, which modify the AST of the program. For each such basic transformation, OptiTrust checks that the transformation preserves the semantics of the program, by leveraging resource typing information. Because the checks performed by OptiTrust depend on resource typing, every intermediate program must typecheck. In particular, if a transformation modifies the code, it may need to also modify the annotations, such as the loop contracts and the ghost instructions. All these aspects associated with correctness criteria and preservation of typing will be discussed in Section 6.

2.2 The Particle Simulation Case Study

Particle-In-Cell (PIC) is a technique commonly used to simulate plasma, where charged particles are in motion, by approximating the charge distribution using a grid.[5] In the present case study, we consider a simplified PIC simulation, focusing on the computations associated with one particular cell. Our goal is to illustrate how OptiTrust can be used to derive a certain number of transformations idiomatic of particle simulations—and also ubiquitous in other physics simulation code.

Unoptimized code. Fig. 5 presents the implementation of a kernel of a particle simulation. It is a simplified version of a realistic simulation code. A number of particles, all with the same mass and charge, move inside a large cube. We assume that the particles do not leave the cube during the simulation. The initial position and speed of every particle is given.

For simplicity, we here assume that the particles do not affect each other—we leave it to future work to optimize the code for the *charge deposit* as in the aforementioned publication. Instead, we

[5]Our case study is inspired by the work by Barsamian et al. [2018], who present a PIC implementation featuring state-of-the-art optimizations, targeting the Intel Xeon Platinum 8160 architecture.

```
void simulate(const vect* fieldAtCorners,
 const int nbSteps, const double deltaT,
 const double pCharge, const double pMass,
 const int nbParticles, particle* particles) {
  const double fieldFactor = deltaT * deltaT * pCharge / pMass;
  vect* const lFieldAtCorners = (vect*) malloc(nbCorners * sizeof(vect));
  for (int i = 0; i < nbCorners; i++) {
    lFieldAtCorners[i].x = fieldAtCorners[i].x * fieldFactor;
    lFieldAtCorners[i].y = fieldAtCorners[i].y * fieldFactor;
    lFieldAtCorners[i].z = fieldAtCorners[i].z * fieldFactor;
  }
  for (int i = 0; i < nbParticles; i++) {
    particles[i].speed.x *= deltaT;
    particles[i].speed.y *= deltaT;
    particles[i].speed.z *= deltaT;
  }
  double const coeffs[8];
  for (int idStep = 0; idStep < nbSteps; idStep++) {
    for (int idPart = 0; idPart < nbParticles; idPart++) {
      compute_corner_interpolation_coeffs(particles[idPart].pos, coeffs);
      double fieldAtPosX = 0.;
      double fieldAtPosY = 0.;
      double fieldAtPosZ = 0.;
      for (int k = 0; k < nbCorners; k++) {
        fieldAtPosX += coeffs[k] * lFieldAtCorners[k].x;
        fieldAtPosY += coeffs[k] * lFieldAtCorners[k].y;
        fieldAtPosZ += coeffs[k] * lFieldAtCorners[k].z;
      }
      const double speed2X = particles[idPart].speed.x + fieldAtPosX;
      const double speed2Y = particles[idPart].speed.y + fieldAtPosY;
      const double speed2Z = particles[idPart].speed.z + fieldAtPosZ;
      particles[idPart].pos.x += speed2X;
      particles[idPart].pos.y += speed2Y;
      particles[idPart].pos.z += speed2Z;
      particles[idPart].speed.x = speed2X;
      particles[idPart].speed.y = speed2Y;
      particles[idPart].speed.z = speed2Z;
    }
  }
  for (int i = 0; i < nbParticles; i++) {
    particles[i].speed.x /= deltaT;
    particles[i].speed.y /= deltaT;
    particles[i].speed.z /= deltaT;
  }
  free(lFieldAtCorners);
}
```

Fig. 6. Optimized code for the particle simulation case study.

assume an external electric field that affects the acceleration of the particles. This external electric field, is described by 8 vectors, which correspond to the field at the corners of the cube. The electric field that applies at a given position inside the cube is obtained by means of a linear interpolation with respect to the fields at the corners—a standard technique in particle-in-cell (PIC) simulations.

The simulation proceeds as follows. At each time step, all the particles are updated. For a given particle, its speed is first updated, based on the value of the acceleration at the position of this particle. Then, the position of the particle is updated, based on its speed. Observe how, in Fig. 5, these updates are described at a high-level of abstraction, using vector operations, as well as a matrix-vector product for computing the interpolation. The auxiliary function `compute_corner_interpolation_coeffs`, not shown, computes the interpolation coefficients associated with the position of the particle.

```
Target.set_prefix [cFunBody "simulate"];
Loop.move_out [cVarDef "coeffs"];
Function.inline [multi cFun ["matrix_vect_mul"; "vect_add"; "vect_mul"]];
Variable.def "fieldFactor" (expr "deltaT * deltaT * pCharge / pMass")
  [tBefore; cVarDef "lFieldAtCorners"];
Record.set_explicit [multi cWrite ~typ:["vect"; "particle"]];
Record.to_variables [cVarDefs ["fieldAtPos"; "pos2"; "speed2"; "accel"]];
Variable.inline [cVarDef "p"];
Accesses.scale_inplace ~factor:(var "fieldFactor") [nbMulti; cVarRe "fieldAtPos[XYZ]"];
Accesses.scale_inplace ~factor:(var "deltaT")
  [nbMulti; sExprRe "particles\\.speed\\.[xyz]"];
Accesses.scale_incopy ~into:"lFieldAtCorners" ~factor:(var "deltaT")
  [nbMulti; cOr [cVarRe "speed2[XYZ]"]; ];
Accesses.scale_immut ~factor:(var "deltaT") [nbMulti; cVarRe "speed2[XYZ]"];
Variable.inline [cVarDefsRe ["accel[XYZ]"; "pos2[XYZ]"]];
Arith.(simpls_rec [expand; gather_rec]) [];
Cleanup.std [];
```

Fig. 7. Optimization script for the particle simulation case study [TODO: finalize].

Optimized code. Fig. 6 shows our optimized code for the function `simulate`. Three key optimizations are applied. First, the vector and matrix operations are replaced with operations over individual fields (named `pos.x`, `pos.y`, `pos.z`, `speed.x`, `speed.y`, and `speed.z`). Moreover, local vector variables are replaced with families of variables (e.g., `fieldAtPosX`, `fieldAtPosY`, and `fieldAtPosZ`). Second, the array `coeffs`, that was used to store the interpolation coefficients computed by `compute_corner_interpolation_coeffs`, is now allocated once and for all outside the loop. Third, and most importantly, a *scaling* transformation is applied on the data in order to simplify the arithmetic computations that need to be performed at every time step. We next explain how this standard optimization technique works, focusing on the x-coordinate.

Consider a particle. At a given time step, its speed, written v, its the position, written x, are updated, according to the formulae: $a{=}qE/m$ and $v \mathrel{+}= a\Delta_t$ and $x \mathrel{+}= v\Delta_t$ where E denotes the electric field interpolated at the location of this particle. The constants q, m, and Δ_t corresponds to the program variables `pCharge`, `pMass`, and `deltaT`, respectively. The idea of scaling transformation is to store not the values of E and v, but instead the values E' and v' defined as: $E' = qE\Delta_t^2/m$ and $v' = \Delta_t v$. The interest is that the speed and position updates at a given time step are now described using much simpler formulae: $v' \mathrel{+}= E'$ and $x' \mathrel{+}= v'$. These formulae involve nothing more than simple additions. To implement this scaling transformation, the components of the field `speed` of the array `particles` are multiplied, in-place, by a factor Δ_t before starting the simulation; symmetrically, at the end of the simulation, the values are divided by Δ_t. For the electric field array, which is read-only, the scaling factor is applied on an auxiliary array named `lFieldAtCorners`, obtained by multiplying the values of `fieldAtCorners` by $q\Delta_t^2/m$. (An in-place update would also have been possible.) By linearity of the interpolation computations, this scaling propagates to the values computed for the electric field at the particle location (`fieldAtPosX`, `fieldAtPosY`, and `fieldAtPosZ`).

Optimization script. Fig. 7 shows our optimization script. The operator `multi` is a shorthand for describing a union of targets. For example, `multi cFun ["matrix_vect_mul"; "vect_add"; "vect_mul"]` is equivalent to `nbMulti; cOr [[cFun "matrix_vect_mul"]; [cFun "vect_add"]; [cFun"vect_mul"]]`, where `cOr` is the target disjunction combinator. The constraint such as `cVarRe` match variable names using regular expressions. The constraint `sExprRe` matches C expressions according to their string representation, again using regular expressions. The crux of the script is the evaluation of `Arith.(simpls_rec [expand; gather_rec])` in the script. This transformation performs the arithmetic

```
void mm(const int m, const int n, const int p,
        const float A[m][p], const float B[p][n], float C[m][n]) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[i][k] * B[k][j];
      C[i][j] = sum;
} } }
void mm1024(const float A[1024][1024], const float B[1024][1024], float C[1024][1024]) {
  mm(1024, 1024, 1024, A, B, C);
}
```

Fig. 8. Unoptimized C code for the matrix-multiply case study, using multi-dimensional arrays, and the specialization to input size 1024.

simplifications that make all the scaling factors cancel out, leaving exactly the trivial additions for updating the speed and the position values of each particle.

[TODO: comment once finalized]

Benefits of using OptiTrust. Applied mathematicians commonly write optimized code such as that of Fig. 6 by hand. Revealing the x, y and z coordinates triples the size of the code, and applying a scaling transformation by hand is a highly error-prone task. The aim of OptiTrust is to provide them with an alternative route, more productive and more trustworthy.

2.3 The Matrix-Multiply Case Study

TVM [Chen et al. 2018] is the state-of-the-art, industrial-strength, semi-automatic compiler for machine learning code. The TVM tutorial presents an optimization script[6] (a.k.a. *schedule*) for optimizing a matrix multiplication function, specialized for square matrices of size 1024. This script has been carefully tuned to produce code optimized for specific Intel CPUs. On a 4-core Intel i7-8665U CPU with AVX2 support, the TVM experts thereby achieve a speedup of 150× over a totally naive, sequential implementation of matrix multiply.[7] The aim of this third case study is to demonstrate the ability of OptiTrust to produce code that matches the performance delivered by TVM. More precisely, we show that, using a transformation script reasonably short compared with the TVM script, we are able to generate code that features the exact same optimization patterns as in the TVM case study.

Unoptimized code. Fig. 8 shows a naive implementation of matrix-multiply. Like for the first case study, to help the reader we use multi-dimensional arrays as opposed to the flat arrays visible in our contract-annotated code (Fig. 10).

Optimized code. TVM output code is expressed not as C code, but directly in the intermediate representation of LLVM, known as LLVM IR. Hence, our first task has been to manually parse the LLVM IR output associated with TVM's matrix-multiply case study, and to infer the corresponding C code. This C code is essentially equivalent to that shown in Fig. 9, which contains the code that we produce using OptiTrust.

[6] https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html

[7] The 150× speed up achieved using TVM does not quite match the 204× speedup achieved by the proprietary Intel's MKL, a library manually optimized by Intel's experts. Yet, keep in mind that whereas the MKL provides optimized implementation for a fixed set of functions, the TVM compiler can be used to optimize entire classes of functions. We leave it to future work to investigate the extent to which OptiTrust could be used to derive code that matches the performance of MKL.

```c
void mm1024(const float* A, const float* B, float* C) {
  float* pB = (float*)malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++) {
    for (int bk = 0; bk < 256; bk++) {
      for (int k = 0; k < 4; k++) {
        for (int j = 0; j < 32; j++) {
          pB[32768 * bj + 128 * bk + 32 * k + j] = B[1024 * (4 * bk + k) + 32 * bj + j]; }}}}
  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* sum = (float*)malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          sum[32 * i + j] = 0.; }}
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[1024 * (32 * bi + i) + 4 * bk] * pB[32768 * bj + 128 * bk + j]; }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[1 + 1024 * (32 * bi + i) + 4 * bk] * pB[32 + 32768 * bj + 128 * bk + j]; }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[2 + 1024 * (32 * bi + i) + 4 * bk] * pB[64 + 32768 * bj + 128 * bk + j]; }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[3 + 1024 * (32 * bi + i) + 4 * bk] * pB[96 + 32768 * bj + 128 * bk + j]; }
          memcpy(&sum[32 * i], s, sizeof(float[32])); }}
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j]; }}
      free(sum);
  } }
  free(pB);
}
```

Fig. 9. Our optimized C code for the matrix-multiply case study. This code features the same optimization patterns and achieves similar performance as the reference output of TVM.

Remark: even though the indices for array accesses involved in this optimized code from Fig. 9 contain somewhat obfuscated arithmetic formulae, these accesses can be displayed until the last optimization step of the OptiTrust script, in the form of multi-dimensional accesses, e.g. `pB[MINDEX4(32, 256, 32, 4, bj, bk, 3, j)]`. Moreover, for the purpose of readability of generated programs, OptiTrust offers the option to print multi-dimensional the same array access in the shorter form `pB[bj; bk; 3; j]`. This form, purposely out of the syntax of C, remains non-ambiguous because the size information appears in the description of the resources at hand.

Compared with the naive code from Fig. 8, the optimized code from Fig. 9 integrates numerous optimizations.

```
void mm(float* C, float* A, float* B, int m, int n, int p) { // naive matrix-multiply
  __reads("A ~> Matrix2(m, p), B ~> Matrix2(p, n)");
  __modifies("C ~> Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xmodifies("for j in 0..n -> &C[MINDEX2(m, n, i, j)] ~> Cell");
    for (int j = 0; j < n; j++) {
      __xmodifies("&C[MINDEX2(m, n, i, j)] ~> Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++)
        sum += A[MINDEX2(m, p, i, k)] * B[MINDEX2(p, n, k, j)];
      C[MINDEX2(m, n, i, j)] = sum;
    }
  }
}
void mm1024(float* C, float* A, float* B) { // specialization to 1024x1024 matrices
  __reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)");
  __modifies("C ~> Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}
```

Fig. 10. Unoptimized C code for the matrix-multiply case study, using flat arrays and augmented with resource annotations.

- The body of the generic matrix multiply function `mm` is specialized to the size `1024`.
- An auxiliary matrix named `pB` is allocated to store the transpose of the matrix `B`. The introduction of this auxiliary matrix induces a cost for the initial copy, but then greatly improves the memory access patterns.
- The matrices are processed by blocks of size 32: each loop over a range of size 1024 is replaced with 2 loops each of range 32. Blocking improves locality in matrix-multiply.
- Results are not accumulated directly in the output matrix `C`. Instead, the results are first accumulated into a stack-allocated array named `s` of size 32. This array `s` can be mapped onto a few 256-bit registers.
- The results accumulated in `s` are then transferred into another stack-allocated array, named `sum`, which has the same size as a block but uses a different memory layout. For each row of the block being processed, two `memcpy` operations are used for synchronizing `s` with `sum`. After processing a block in full, the values from the array `sum` are copied into the output array `C`.
- The various loops are reordered in a particular manner, both to improve cache locality and to enable parallelization. The outermost loops are executed in parallel by several cores. The instructions of the inner loop are parallelized by means of SIMD operations.
- The 4 loops tagged as `#pragma omp simd` in Fig. 9 are very similar. However, if we attempt to factorize them into a loop with 4 iterations, then Intel's compiler (ICX) produces slower code. Hence, the loops need to be unfolded as shown.

Please keep in mind that we have not chosen this particular set of optimizations. We have simply inferred these optimizations from the output of the TVM case study. Our goal is to reproduce the exact same set of optimizations.

Annotated Unoptimized Code. Fig. 10 shows the annotations that we need to insert into the naive matrix-multiply implementation from Fig. 8 for typechecking the resources involved. The syntax of annotations has been explained earlier, except for the iteration construct, which we explain next.

```
Function.inline_def [cFunDef "mm"];
let tile (id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ id) ~bound:TileDivides [cFor id] in
List.iter tile [("i", 32); ("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"] [cPlusEq ()];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB" ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s" ~copy_dims:1 [cFor ~body:[cPlusEq ()] "k"];
Omp.simd [cFor ~body:[cPlusEq ()] "j"];
Omp.parallel_for [cFunBody "mm1024"; cStrict; cFor ""];
Loop.unroll [cFor ~body:[cPlusEq ()] "k"];
```

Fig. 11. Optimization script for the matrix-multiply case study.

```
k = tvm.reduce_axis((0, P))
A = tvm.placeholder((M, P))
B = tvm.placeholder((P, N))

pB = tvm.compute((N / 32, P, 32),
  lambda bj, k, j:
    B[k, bj * 32 + j])

C = tvm.compute((M, N),
  lambda i, j:
  sum(A[i, k] * pB[j // 32, k, j \% 32],
      axis=k))
```

```
CC = s.cache_write(C, "global")
bi, bj, i, j = s[C].tile(
  C.op.axis[0], C.op.axis[1], 32, 32)
s[CC].compute_at(s[C], bj)
i2, j2 = s[CC].op.axis
(kaxis,) = s[CC].op.reduce_axis
bk, k = s[CC].split(kaxis, factor=4)
s[CC].reorder(bk, i2, k, j2)
s[CC].vectorize(j2)
s[CC].unroll(k)
s[C].parallel(bi)
bj3, _, j3 = s[pB].op.axis
s[pB].vectorize(j3)
s[pB].parallel(bj3)
```

Fig. 12. TVM case study for matrix-multiply. On the left, input code in TVM's domain specific language. On the right, TVM optimization script (a.k.a. schedule) to the right. Both use Python syntax.

The outer loop on `i` is annotated with a clause of the form `__xmodifies("for j in 0..n -> Hj")`, where the logical **`for`** construct denotes an iteration over resources. This clause indicates that the `i`-th iteration requires exclusive access to the union of the resources described by `Hj`, for all `j` in the range `0..n`. The resource `C ⇝ Matrix2(m,n)` is equivalent to `"for i in 0..m -> for j in 0..n -> &C[i][j] ⇝ Cell"`, which covers all the `n*m` cells of the matrix `C`. The resource `"for j in 0..n -> &C[i][j] ⇝ Cell"` covers only the `i`-th row of that matrix. Further in the paper, this same resource will be written using LaTeX-style notation as: $\bigstar_{j \in 0..n} \big(\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}\big)$.

Optimization script. Fig. 11 shows our optimization script, which consists of only 10 lines. Internally, though, the high-level transformations mentioned in the script trigger the application of 55 basic transformations. These internal basic transformations may be visualized in the execution trace generated by OptiTrust.[4] An illustrative example is the call to `Loop.reorder_at` on Line 4 of Figure 11. This combined transformation takes as argument a specific instruction (referred to as "an instruction of the form `+=`") as well as a description of the desired order for the loops that surround this instruction (the list `["bi"; "bj"; "bk"; "i"; "k"; "j"]`). The `reorder` transformation iteratively "brings down" the loops that need to be swapped closer to the instruction, starting from the innermost loops, and processing the loops until the outermost one. The call to `reorder_at` in our script triggers a total of 4 *loop swaps*, 6 *loop fissions*, and 2 *loop hoist* operations. In particular, the effect of these 2 hoist operations is to turn local variable named `sum` in Fig. 8 into the 2D-array named `sum` in Fig. 9.

Comparison Against TVM. The TVM matrix-multiply case study appears in Fig. 12. We only comment on specific aspects and refer to TVM's tutorial[6] for further details. In TVM, input programs are written in a domain-specific language embedded in Python. Ideally, the matrix-multiply program shown on the left-hand side of Fig. 12 would be written more simply as:

```
k = tvm.reduce_axis((0, P))
```

```
A = tvm.placeholder((M, P))
B = tvm.placeholder((P, N))
C = tvm.compute((M, N), lambda i, j: sum(A[i, k] * B[k, j], axis=k))
```

Yet, TVM is unable to express the introduction of the transposed matrix of `B`, named `pB`, as a code transformation. The programmer therefore needs to introduce this auxiliary structure manually in the input source code. Likewise, the blocking strategy needs to be hardwired in the source code. In contrast, our input program for matrix multiply (Fig. 8) is expressed using standard C code, and needs not include any optimization. Starting from a totally unoptimized reference code improves readability, trustworthiness, and maintainability.

Besides, note that our although our source code for matrix-multiply is currently expressed using explicit loops, in the future we could alternatively express it using higher-order combinators similar to those available in TVM (using either C++ syntax or the syntax of another language).

The right-hand side of Fig. 12 shows TVM's optimization script. Our optimization script for that case study (Fig. 11) is not much longer than that of TVM. As mentioned earlier, we have carefully checked that the C code produced using OptiTrust features the same optimizations and delivers the same performance as the LLVM IR code produced using TVM. To the best of our knowledge, OptiTrust is the first general-purpose optimization framework to demonstrate the ability to reproduce a case study from a state-of-the-art, specialized compiler such as TVM.

Guided by all the contents from Fig. 12, TVM applies a monolithic compilation pass to produce optimized code. TVM does not provide interactive, easily-readable feedback for the transformations performed. In contrast, OptiTrust applies a series of local, source-to-source transformations, manipulating programs expressed in conventional C syntax. Moreover, it provides human-readable *diff* s for every transformation step—as well as for every substep—involved in the optimization process.

2.4 Evaluation of OptiTrust

Now that we have given a tour of the features of the OptiTrust framework, let us try to evaluate it against the set of desirable properties for semi-automatic code optimization from Section 1.1.

Generality. As pointed out in Section 1.3, this first release of OptiTrust has a number of limitations: it applies to a subset of the language, for a simplified version of Separation Logic, and there remains a fair number of transformations to implement. Thus, OptiTrust in its current certainly form does not yet demonstrate full generality. Yet, every aspect of OptiTrust has been designed towards that goal.

Expressiveness and Control. OptiTrust supports a number of basic transformations that, taken individually, may appear reasonably straightforward. However, by chaining such transformations in the desired manner, the OptiTrust user is able to achieve state-of-the-art high-performance code, which resembles code that an expert might have written by hand. Moreover, the many basic transformations involved need not be explicitly invoked by the user: the use of high-level combined transformations allows to achieve expressiveness via concise scripts—recall, e.g., the call to `Loop.reorder` in the matrix-multiply case study. A key feature of OptiTrust is that, at any stage in the optimization process, the user remains fully in control.

Expressiveness also depends on the generality of the correctness criteria associated with every transformation. In practice, there could be situations where the user may want to legitimately apply a basic transformation, yet OptiTrust's implementation is unable to recognize this application as correct. In the short term, one option is for the user to treat this particular step as "user-trusted", and to rely on human review of the *diff* associated with that step.[8]

[8]In the long term, by leveraging a full-blown Separation Logic, possibly combined with the use of interactive proofs, users could replace a piece of code with any other piece of code that provably satisfies the same specification.

Feedback. For each step in the transformation script, OptiTrust delivers feedback in the form of human-readable C code. The user usually only needs to read the *diff* against the previous code. Interestingly, OptiTrust also records a trace that allows investigating all the substeps triggered by a *combined* transformation. This information is critically useful when the result of a high-level transformation does not match the user's intention. Such traces can also be very useful for third-party reviewing of an optimization process. Besides, a key feature of OptiTrust is its fast feedback loop. The production of fast, human-readable feedback in a system with significant control is reminiscent of interactive proof assistants, and of the aforementioned ATL tool [Liu et al. 2022].

Composability. OptiTrust transformation scripts are expressed as OCaml programs, and each transformation from our library consists of an OCaml function. Because OCaml is a full-featured programming language, OptiTrust users may define additional transformations at will by combining existing transformations. User-defined transformations may query the abstract syntax tree (AST) that describes the C code, allowing to perform analyses before deciding what transformations to apply. Furthermore, because OCaml is a higher-order programming language, transformation can take other transformations as argument. We use this programming pattern for example to customize the arithmetic simplifications to be performed after certain transformations.

Extensibility. If the user needs a transformation that is not expressible as a combination of transformations from the OptiTrust library, a custom transformation can be devised. Because OptiTrust does not rely on heuristics, adding a new transformation to OptiTrust does not impact in any way the behavior of existing scripts. To define relatively simple custom transformations, OptiTrust provides a term-rewriting facility based on pattern matching. For more complicated transformations, one can follow the patterns employed in the OptiTrust's library. For all custom transformations, it is the programmer's responsibility to work out the criteria under which applying the transformation preserves the semantics of the code, and to adapt contracts if necessary in order to produce well-typed code.

Trustworthiness. Compilers are well-known to be incredibly hard to get 100% correct [Yang et al. 2011]. Like compilers, optimization tools are highly subject to bugs. OptiTrust mitigates the risks of producing incorrect code in two ways. First, the *diff* of every step can be thoroughly scrutinized. Secondly, as explained in Section 1.3, we have organized the OptiTrust code base so as to isolate the implementation of the *basic* transformations, which consists of transformations that directly modify the AST. Only basic transformations need to be trusted. We have been careful to systematically minimize the complexity of the interface and of the implementation of our basic transformations. All other transformations—the *combined* transformations—are *not* part of the trusted computing base (TCB).

This completes our high-level presentation of the OptiTrust framework. The remaining sections present the implementation of OptiTrust: its internal AST, its typechecking algorithm, and its transformations.

3 OPTITRUST'S INTERNAL AST

In OptiTrust, input C programs are encoded into an imperative λ-calculus. All code transformations are performed on that imperative λ-calculus. Then, programs are decoded back into C syntax. This approach enables OptiTrust to report the *diff* associated with a transformation in terms of a concise syntax familiar to the programmer.

The OptiTrust abstract syntax tree (AST) is represented as an immutable tree data structure. A program transformation takes as input such an immutable AST, and produces as output another

$$
\begin{array}{rcll}
r & := & \mid\ \mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}}) & \text{range for simple for-loops} \\
\pi & := & \mid\ \mathbf{par} \ \mid\ \cdot & \text{optional parallel flag on simple for-loops} \\
t & := & \mid\ x \ \mid\ \mathbf{res} & \text{variables, and the special variable } \mathbf{res} \\
 & & \mid\ b \ \mid\ n & \text{boolean values, and number values} \\
 & & \mid\ (t_1; \ldots; t_n) & \text{sequence} \\
 & & \mid\ \mathbf{let}\ x = t & \text{variable definition} \\
 & & \mid\ \mathbf{fun}(a_1, \ldots, a_n) \mapsto t & \text{function definition} \\
 & & \mid\ t_0(t_1, \ldots, t_n) & \text{function call} \\
 & & \mid\ \mathbf{for}^{\pi}(i \in r)\ t & \text{possibly parallel, simple for-loop} \\
 & & \mid\ \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 & \text{conditional} \\
 & & \mid\ \{f_1 = t_1; \ldots; f_n = t_n\} \ \mid\ [t_1; \ldots; t_n] & \text{structure and array as values} \\
 & & \mid\ t_1[t_2] \ \mid\ t.f & \text{projection from array/struct values} \\
 & & \mid\ t_1 \boxplus t_2 \ \mid\ t \boxdot f & \text{address computation}
\end{array}
$$

Fig. 13. Grammar of OptiTrust's internal λ-calculus. Type annotations on binders and operators are not shown.

AST, which may share subtrees with the input AST. There are two major benefits to following a purely functional programming style using immutable trees. First, this approach avoids numerous bugs typically associated with inadvertent sharing of subtrees when modifying data structures in-place. Second, this approach, by enabling sharing, improves the efficiency of the construction of a complete execution trace for reporting to the user all the intermediate ASTs constructed during the evaluation of the user's transformation script.

The purpose of this section is to present OptiTrust's internal language, whose constructs appear throughout the rest of the paper, from the statement of typing rules in Section 4 and usage analysis in Section 5, to the description of transformations in Section 6.

In Section 3.1, we present the grammar of our imperative λ-calculus. It is mostly standard, except perhaps for the special variable **res** used to bind return values, for the two primitive operations to implement stack-allocation, for the construct for (possibly parallel) simple for-loops, and for the operators that compute address arithmetic. These ingredients are not ubiquitous in presentations of λ-calculus, yet they have been employed by others in prior work.

In Section 3.2, we describe, at a high-level, the intuition behind OptiTrust's translation from C into our internal λ-calculus. Again, such a translation is relatively standard: numerous compilers include a phase that eliminates mutable variables and l-values. The specificity of our translation is that it introduces a few annotations on terms to allow implementing the reciprocal translation.

3.1 OptiTrust's Internal AST

Fig. 13 gives the grammar of OptiTrust's AST. In this language, variables are bound by let-bindings and function definitions, and they are always immutable. Immutable variables allow for a straightforward implementation of substitution: variables may be substituted with values without concern on whether occurrences appear as right- or left-values. We equip OptiTrust's language with a standard call-by-value semantics (not shown). We next describe the grammar, starting with the less common features.

Sequences in OptiTrust. A sequence is a term that consists of a list of subterms or let-bindings, to be executed in order. We impose in the OptiTrust AST the invariant that every function body consists of a sequence block, even if the sequence contains a single instruction. Likewise, loop body and branches of conditionals systematically consist of sequences.

The Special Result Variable. A special variable, named **res**, is used to denote the result value of a sequence, typically the body of a function. A statement **return** t that appears in terminal position of a C function is translated as a binding "**let res** = t". More generally, in OptiTrust, all sequences with a non-void type are expected to contain exactly one **let res** construct.

The variable **res** also typically appears in function contracts, to specify the return value of a function. The use of a dedicated name such as **res** is common practice in program verification tools, such as ESC/Java [Flanagan et al. 2002] or Why3 [Filliâtre 2003]. Besides, treating **return** as an assignment instruction is a technique that appears, e.g., in the Viper program verification tool [Müller et al. 2017]. A key interest of the "**let res**" construct is that it allows placing instructions *after* the point at which the return value is computed. Doing so is specifically useful for *ghost instructions* that must appear at the very end of a sequence. Indeed, the purpose of ghost instructions is to refine the formulation of certain resources, and these resources may depend on the return value.

Manipulation of Heap and Stack Cells. To account for heap-allocated data, OptiTrust provides the following standard primitive functions: alloc for allocating one or several uninitialized cells, get for reading a cell, set for writing a cell, and free for freeing allocated cells. As usual, a read in an uninitialized memory cell is undefined behavior. Additionally, to account for stack-allocated variables, OptiTrust includes two special functions. The operation ref(t) allocates a memory cell initialized with a specific content t, and whose space is automatically reclaimed at the end of the surrounding sequence. The operation ref_uninit() is similar, but it allocates a memory cell without initializing its contents. These two special operations are meant to occur as part of a let-binding, for example **let** x = ref(3), occurring directly within a sequence. The two stack-allocation operators, apart from their implicit-free behavior, are treated like other primitive functions. Remark: we have considered the possibility of encoding ref and ref_uninit using alloc, set and free, but ultimately concluded that this approach was slightly less practical, leading to additional complexity in code transformations and in target resolution.

Possibly Parallel, Simple For Loops. The construct $\mathbf{for}^{\pi}\,(i \in \mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}}))\; t_{\text{body}}$ describes a *simple-for-loop*. In such a loop, the immutable variable i denotes the loop index, and the loop range, which consists of the loop bounds and the per-iteration step are evaluated only once before starting the loop. Following the convention used by Python and other languages, the index goes from the *start* value inclusive to the *stop* value exclusive. If the *step* value is negative, the loop index iterates downwards. Optionally, the loop may be tagged with a *parallel* flag (i.e., setting π to **par**), thereby asserting that the loop should be treated as a parallel loop by the compiler and the runtime. This flag corresponds to the directive: `#pragma openmp parallel`. The restrictions imposed by OpenMP on the ranges of parallel for-loops essentially constraint them to fit the format $\mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$, which is the format that we use for our simple-for-loops.

Structured Data. The constructs $\{f_1 = t_1; \ldots; f_n = t_n\}$ and $[t_1; \ldots; t_n]$ build records and arrays as constant values. Mutable record and arrays are allocated by means of a call to the alloc function. OptiTrust features 4 operations to manipulate structured data. If a corresponds to a constant array value, then the operation $a[i]$ reads the i-th cell of the array a. If, however, a corresponds to the address of a heap-allocated or a mutable stack-allocated array, then the memory address of i-th cell of the array a can be computed by the operation $t \boxplus_{\hat{\tau}} i$, where $\hat{\tau}$ denotes the type of the elements of t. This operation corresponds to the C pointer arithmetic operation `t+i`. The contents of that cell may be retrieved by evaluating $\mathrm{get}(t \boxplus_{\hat{\tau}} i)$. Likewise, reading the field f of a constant record r is described by the operation $r.f$, whereas the memory address of the field f of a record r allocated in memory is described by the operation $r \boxdot_{\hat{\tau}} f$, where $\hat{\tau}$ denotes the type of r. This operation corresponds to shifting the pointer r by the offset associated with the field f from the type $\hat{\tau}$.

Other Language Constructs. The other language constructs from Fig. 13 are standard. They include sequence, variable definition, function definitions, function calls, and conditionals. Our implementation accounts for a diversity of literal types. For simplicity, we consider in this paper only two kinds of literals: the metavariable b denotes a boolean literal, and the metavariable n denotes an integer literal.

Other Primitive Operations. Besides the aforementioned primitive operations for manipulating heap and stack cells, OptiTrust provides primitive functions that correspond to the arithmetic and boolean operators of the C languages. For the special operators `&&` and `||`, we encode them using conditionals whenever the second argument is not a simple expression (in particular, if it might fail or perform side effects).

For the pre/post-increment/decrement operators, such as `i++`, we encode them using conventional function calls, taking advantage of the fact that our type system guarantees the correctness of such an encoding. For example, our type system rules out problematic expressions such as `f(i++,i++)`. More precisely, provided that the input program does not exhibit undefined behavior, and provided that the encoding of the input program is well-typed in our system, then an expression `i++` appearing in that C program is equivalent to the expression `get_incr(&i)`, with the definition `int get_incr(int *p){ int r = *p; *p = r + 1; return r; }`.

Unsupported Language Features. As mentioned earlier, the present paper aims at demonstrating the interest of OptiTrust's approach to code optimization. It does not aim at covering all the features of the C language. Let us nevertheless comment on three features that we look forward to support in the near future.

For while-loops and general forms of for-loops, we plan to use an encoding into a single form of repeat-loop, following the approach of numerous compilers. Observe that, despite the absence of general loops, the language that we consider in this paper is Turing-complete thanks to our support for general recursive functions.

To handle abrupt termination, as triggered by **break**, **continue**, and non-final **return** statements, we need a generalization of our type system. The treatment of abrupt termination in Separation Logic is well-understood—they are handled, for example, in the VST program verification framework for C programs [Cao et al. 2018]. Yet, its support introduces a fair amount of additional complexity, explaining why we have not included them in the present paper.

The C language allows mutation of function arguments, whereas OptiTrust features only immutable arguments. Even though mutating function arguments in C is sometimes considered bad practice, we could support this pattern in OptiTrust by introducing an auxiliary fresh local mutable variable, and turning the mutated argument into a constant argument.

3.2 Bidirectional Translation between C and OptiTrust's Language

As mentioned earlier, OptiTrust *encodes* input C programs into OptiTrust's internal AST, for the purpose of facilitating typechecking and transformation of the code. Because we wish to typecheck terms after their encoding, the encoding must not depend on the result of our resource analysis. After an operation on the internal AST, OptiTrust *decodes* the program back into C syntax, for the purpose of proposing feedback in a familiar programming syntax. In particular, the *diff* associated with a transformation requested by the user is computed over the two pieces of C code. The parts of the input C program that are not altered by the requested transformation do not appear in the *diff*.

In the rest of this section, we explain how the initial program is normalized and how annotations in the OptiTrust AST are used in order to ensure that a *round-trip* property holds. Then, we present the key ideas of the encoding scheme by means of example—as pointed out earlier, the elimination of l-values is a standard compiler translation.

Initial Normalization. The current implementation of OptiTrust does not keep in the AST information about comments and spacing—we left this to future work. As a result, for the very first encoding-decoding round-trip, which our implementation automatically applies as preliminary step, some amount of semantically-irrelevant information may be lost. Then, after that initial normalization step, for all programs manipulated by OptiTrust, the *round-trip* property holds: encoding and decoding are exactly reciprocal of each other.

Annotations. To deal with the fact that several C expressions might admit the same encoding in the OptiTrust language, our translation attaches annotation on certain OptiTrust terms. For example, `(*r).f` and `r->f` are both encoded as $\mathrm{get}(r \boxdot f)$, therefore we instrument the encoding to attach a "dont-use-arrow" annotation on the get term when translating `(*r).f`. As another example, the two terms `e1 && e2` and `if (e1){ return e2; } else { return true; }` have the same encoding when `e2` is a nontrivial expression, therefore we attach a "use-`&&`" on the conditional when translating `e1 && e2`. If a transformation step modifies the instruction `return true` from the else-branch into something else, then the annotation "use-`&&`" is ignored by the decoding operation.

Encoding Scheme and Pure Variables. The core of OptiTrust's encoding consists of eliminating l-values. For a heap allocated piece of data, a read operation `*p` is encoded as the function call $\mathrm{get}(p)$, and an assignment `*p = v` is encoded as $\mathrm{set}(p, v)$. For stack-allocated C variables, the encoding distinguishes two cases, depending on whether the address of the variable needs to be manipulated. We say that a stack variable `x` is *pure* if there is no assignment operation on `x` and the address of `x` or one of its sub-components is never taken.[9][10] For such a pure variable, its definition, say `int x = 3`, is encoded simply as $\textbf{let } x = 3$. For a non-pure variable, its encoding involves a stack-allocation. For example, the definition `int x = 3` is encoded as $\textbf{let } x = \mathrm{ref}(3)$, the assignment `x = 4` is encoded as $\mathrm{set}(x, 4)$, and an occurrence of `&x` is encoded as x. Fig. 14 provides additional examples.

4 COMPUTING PROGRAM RESOURCES

Traditional typecheckers have a typing judgment of the form $\Gamma \vdash t : \tau$. Yet, the OptiTrust typechecker needs to account also for linear resources. Following the presentation of Separation Logic, OptiTrust's typing judgment is written as a *triple* of the form $\{\Gamma\}\ t\ \{\Gamma'\}$. The input context Γ decomposes as $\langle E \mid F \rangle$, where E consists of *pure resources* and F consists of *linear resources*. Symmetrically, the output context Γ' contains both pure and linear resources. The pure resources from Γ' typically correspond to ghost return values and to pure postconditions. Triples will be later generalized in Section 5 to the form $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, where Δ denotes a *usage map*, providing a summary explaining which resources are used by every subterm, and how they are used. This section presents the typing entities and the algorithmic typing rules, ignoring usage maps.

The section is organized as follows. Section 4.1 presents the grammar of *pure resources* and *linear resources*. Section 4.2 presents the grammar of *contexts*. Section 4.3 presents the grammar of *function contracts* and *loop contracts*. Section 4.4 presents the declarative entailment relation.

[9] For example, the variable `x` is *not pure* if there exists an occurrence of `&x`, or of the form `&x.f` or `&x[i]`. On the contrary, a pointer variable `x` may be *pure* despite occurring in an expression of the form `&(x->f)`. Indeed, if `x` is pure, then `&(x->f)` is encoded as $x \boxdot f$.

[10] Note that a variable declared as `const` is not necessarily pure, because its address might be taken. On the contrary, a variable that is pure may always be annotated as `const`.

`int x = 3;`	$\longleftrightarrow$ $\textbf{let}_{\texttt{int}}\ x = 3;$	where `x` pure
`f(x);`	$\longleftrightarrow$ $f(x);$	with `void f(int)`
`int* a = malloc(sizeof(int));`	$\longleftrightarrow$ $\textbf{let}_{(\texttt{int*})}\ a = \mathsf{alloc}_{\texttt{int}}(1);$	where `a` pure
`*a = *a + 2;`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int}}(a,\ \mathsf{get}_{\texttt{int}}(a) + 2);$	
`free(a);`	$\longleftrightarrow$ $\mathsf{free}(a);$	
`int z;`	$\longleftrightarrow$ $\textbf{let}_{(\texttt{int*})}\ z = \mathsf{ref\_uninit}_{\texttt{int}}();$	where `z` not pure
`z = 6;`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int}}(z, 6);$	
`int v = z;`	$\longleftrightarrow$ $\textbf{let}_{\texttt{int}}\ v = \mathsf{get}_{\texttt{int}}(z);$	where `v` pure
`int y = 5;`	$\longleftrightarrow$ $\textbf{let}_{(\texttt{int*})}\ y = \mathsf{ref}_{\texttt{int}}(5);$	where `y` not pure
`f(y);`	$\longleftrightarrow$ $f(\mathsf{get}_{\texttt{int}}(y));$	
`y = y + 2;`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int}}(y,\ \mathsf{get}_{\texttt{int}}(y) + 2);$	
`y += 4;`	$\longleftrightarrow$ $\mathsf{set\_add}_{\texttt{int}}(y, 4);$	
`y++;`	$\longleftrightarrow$ $\mathsf{get\_incr}_{\texttt{int}}(y);$	
`int* p = &y;`	$\longleftrightarrow$ $\textbf{let}_{(\texttt{int*})}\ p = y;$	where `p` pure
`*p = *p + 2`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int}}(p,\ \mathsf{get}_{\texttt{int}}(p) + 2);$	
`int* q = &y;`	$\longleftrightarrow$ $\textbf{let}_{(\texttt{int**})}\ q = \mathsf{ref}_{(\texttt{int*})}(y);$	where `q` not pure
`q = &z;`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int*}}(q, z);$	
`*q = *q + 2;`	$\longleftrightarrow$ $\mathsf{set}_{\texttt{int}}(\mathsf{get}_{\texttt{int*}}(q),\ \mathsf{get}_{\texttt{int}}(\mathsf{get}_{\texttt{int*}}(q)) + 2);$	

Fig. 14. Example translations from C code into the OptiTrust's internal AST, with type annotations made explicit. A variable `x` is *pure* if there is no assignment operation on `x` and no occurrence of `&x`.

Section 4.5 presents the subtraction procedure, which corresponds to an algorithmic implementation of the entailment relation. Section 4.6 presents the typing judgment for pure expressions. Finally, Section 4.7 presents our algorithmic typing rules, which define the judgment $\{\Gamma\}\ t\ \{\Gamma'\}$.

Throughout the section, we assume a substitution operator for every entity. Concretely, given a map σ associating variable names to values, we write $\mathrm{Subst}\{\sigma\}(X)$ the substitution of the bindings from σ throughout X.

4.1 Grammar of Resources

As mentioned earlier, a context Γ decomposes as $\langle E \mid F\rangle$, where E contains pure resources and F contains linear resources. We next describe these two kind of resources.

Pure Resources. The pure part of a typing context contains bindings of the form "$x : \tau$", where τ corresponds either to a C type or to a *mathematical type*. A C type is denoted by the meta-variable $\hat{\tau}$. A mathematical type (a.k.a. logical type) corresponds to a type from higher-order logic. Thus, intuitively, the pure part of a typing context Γ can be thought of as an interleaving of a traditional program typing context, which binds program variables to C types, and a Coq context, which binds ghost variables to Coq types, including propositions and specifications.

Mathematical types include propositions, of type Prop: for example "$p : n > 0$" describes a proof p establishing the proposition "$n > 0$". Mathematical types also include logical functions, whose type is written $\tau_1 \xrightarrow{\text{pure}} \tau_2$. In particular, a predicate over a type τ has type $\tau \xrightarrow{\text{pure}} \mathrm{Prop}$. *Function specifications* are predicates over function values. As we will see in the next section, such a

Syntax in C	Syntax in the theory	Description
`p ~> Cell`	$p \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$	permission to access the cell at address p of type $\hat{\tau}$
`p ~> Matrix1(n)`	$p \rightsquigarrow \mathsf{Matrix1}_{\hat{\tau}}(n)$	permission on an array of length n
`p ~> Matrix2(m, n)`	$p \rightsquigarrow \mathsf{Matrix2}_{\hat{\tau}}(m, n)$	permission on a $m \times n$ matrix
`for i in r -> H(i)`	$\bigstar_{i \in r} H(i)$	union of resources $H(i)$, for i in the range r
`_RO(α, H)`	αH	read-only permission on H with fraction α
`_Uninit(H)`	$\mathsf{Uninit}(H)$	permission on H that disallows reads before a write

Fig. 15. Grammar of heap predicates. User-defined representation predicates are left to future work.

hypothesis takes the form $\mathsf{Spec}(f, [a_1, \ldots, a_n], \gamma)$, asserting that the function f expects arguments named a_i and admits the *function contract* γ. Note that function contracts may appear in contexts, while contexts are involved in the statement function contracts. This form of impredicativity is standard in higher-order Separation Logic (e.g., [Charguéraud 2020b]). Another particular kind of predicates are heap predicates (e.g., $p \rightsquigarrow \mathsf{Cell}$), which admit the type Hprop. In formalizations of Separation Logic, Hprop is typically defined as state $\xrightarrow{\text{pure}}$ Prop, where state denotes the type of a memory state, however this definition needs not be revealed to the OptiTrust user.

Thereafter, to avoid confusion between the separating conjunction operation $\star$ from Separation Logic and the star-symbol that denotes a C pointer type, we use the alternative syntax ptr(`A`) to denote the C type `A*`.

Linear Resources. The linear part of a typing context contains bindings of the form "$y : H$", where H is a *heap predicate*, and where y is a name. This name y is used in particular to refer to resources in usage maps. A heap predicate H describes "ownership" of part of the memory. Fig. 15 summarizes the most common heap predicates, which have already been discussed in Section 2, but for which we here introduce LaTeX-style notation, moreover making the type annotations explicit. In particular, $p \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$ is a resource that corresponds to the ownership of a single cell of type $\hat{\tau}$, located at address p. The resource $p \rightsquigarrow \mathsf{Matrix1}_{\hat{\tau}}(n)$ is syntactic sugar for $\bigstar_{i \in 0..n}\, p[i] \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$. This resource corresponds to the ownership of the set of all the cells in the array. The big-star symbol corresponds to the *iterated separating conjunction* of Separation Logic. Likewise, $p \rightsquigarrow \mathsf{Matrix2}_{\hat{\tau}}(n, m)$ denotes $\bigstar_{i \in 0..n} \bigstar_{j \in 0..m}\, p[i][j] \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$. We leave it to future work to provide mechanisms allowing the user to define *representation predicates* for custom data types.

Read-Only Fractions. Following standard Separation Logic, we represent read-only resources using *fractional resources* [Boyland 2003; Jung et al. 2018a]. Intuitively, possessing a non-zero fraction of a linear resource gives read-only access to this resource. Possessing the full fraction (i.e., 1) of a resource gives read-write exclusive access to this resource. Possessing both αH and βH is equivalent to possessing the single resource $(\alpha + \beta)H$. As a result, if we have αH at hand in the context, we can *carve out* a subfraction βH, leaving as remainder $(\alpha - \beta)H$. This splitting operation can be performed for any fraction β such that $0 < \beta < \alpha$.

Every time our typechecker requires a read-only permission on H in a context containing αH, it carves out a subfraction βH out of αH. This strategy ensures that we always keep around a fraction of the read-only resources initially available. These fractions may be useful for typing subsequent terms. When a read-only permission is returned after being used, our typing algorithm eagerly merges back βH and $(\alpha - \beta)H$ into the original form αH. Interestingly, carve-out operations may be performed in cascade, and merge-back operations can be performed in any order. To support this general pattern, we introduce the operation CloseFracs, which appears in our typing rules. The operation CloseFracs repeatedly applies the following rewrite rule:

$$(\alpha - \beta_1 - \ldots - \beta_n)H \star (\beta_i - \gamma_1 - \ldots - \gamma_m)H \longrightarrow (\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H.$$

In general, if we start with a full permission H, that is $1H$, then whatever the order in which we carve out and merge back all the fractions of H, we ultimately recover H in full.

At this stage in the development of OptiTrust, we assign meaning to a fractional permission αH only when H is a heap predicate that denotes either a single cell $p \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$ or a group of cells (typically, an array or matrix predicate). Finding an appropriate generalization of this definition of fractional resources that applies on user-defined container data structures remains, for the most part, an open problem in Separation Logic.

Resources for Uninitialized Cells. Separation Logic can guarantee that a program never reads from an uninitialized memory cell. The traditional way to formalize this approach is as follows.

(A1) Allocation of a memory cell at address p is specified as producing the heap predicate $p \rightsquigarrow \bot$, where $\bot$ is a special token denoting uninitialized content.

(A2) The specification of the read operation requires not only a fraction of a permission of the form $p \rightsquigarrow v$, but also requires the property $v \neq \bot$.

OptiTrust operates not on predicates of the form $p \rightsquigarrow v$, but on less precise predicates of the form $p \rightsquigarrow \mathsf{Cell}$. Hence, we follow a slightly different approach for handling uninitialized cells.

(B1) Our heap predicate $p \rightsquigarrow \mathsf{Cell}$ denotes not only the ownership of the cell at location p but also the information that its contents is previously initialized.

(B2) Our heap predicate $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell})$ denotes the ownership of the cell p, yet without the permission to read its contents before it is initialized.

(B3) We specify a write operation on p as consuming $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell})$ and producing $p \rightsquigarrow \mathsf{Cell}$.

(B4) We allow a permission $p \rightsquigarrow \mathsf{Cell}$ to be downgraded into $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell})$ at any time.

The combination of (B3) and (B4) means that a write operation can also be typechecked as an operation that consumes and returns the permission $p \rightsquigarrow \mathsf{Cell}$. More generally, as detailed further on (in Section 4.5), when our typechecker encounters a term that requires $\mathsf{Uninit}(H)$ in a context where the plain resource H is available, it weakens H into $\mathsf{Uninit}(H)$ on-the-fly.

We generalize the predicate to the form $\mathsf{Uninit}(H)$ to describe uninitialized arrays and matrices. Concretely, for a matrix, $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Matrix2}(m, n))$ corresponds to $\bigstar_{i \in 0..n} \bigstar_{j \in 0..m} \mathsf{Uninit}(p[i][j] \rightsquigarrow \mathsf{Cell})$. We do not attempt to provide a definition of $\mathsf{Uninit}(H)$ for arbitrary H: like for read-only resources, we use uninitialized resources only for cells and groups of cells.

4.2 Construction and Operations on Typing Contexts

Construction of Contexts. A context Γ takes the form $\langle E \mid F \rangle$, where E consists of a list of *pure resources* and F consists of a set of *linear resources*. In its expanded form, a context is written $\langle x_0 : \tau_0, ..., x_n : \tau_n \mid y_0 : H_0, ..., y_n : H_n \rangle$, where x_i denotes a pure resource of type τ_i, and y_i denotes a linear resource with heap predicate H_i. The names x_i and y_i must all be distinct. The pure part E is a *telescope*: the variable x_i may occur in any τ_j where $i < j$. Moreover, all the pure variables x_i scope over the linear formulas H_j. The order of the linear resources is irrelevant.

The pure part E of a context Γ may contain bindings of a special form, called *alias bindings*. Such a binding takes the form "$x_i : \tau_i := v_i$". The intention is that, in presence of such an alias, our typechecker eagerly replaces x_i with v_i during internal unification operations. An alias binding corresponds exactly to a *local definition* in Coq. An alias binding "$x_i : \tau_i := v_i$" may also be interpreted as a conventional binding that associates x_i to a singleton type whose sole inhabitant is v_i.

Following standard practice in proof assistants, variable names that are nowhere mentioned may be hidden. For example the context $\langle p : \mathsf{ptr(int)},\ n : \mathsf{int},\ n > 0 \mid p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}} \rangle$ contains two anonymous resources: $n > 0$ and $p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}}$. Internally, though, all context items are identified by a variable name.

Bindings of the Special Result Variable. Recall from Section 3.1 that, in OptiTrust, if a term t produces a result value, then this result is bound to the special variable **res**. Therefore, if t has a non-void type τ then, in the triple $\{\Gamma\}\ t\ \{\Gamma'\}$, the output context Γ' binds a variable **res** of type τ. Moreover, if in t the binding on **res** takes the "**let res** $= v$" for a *pure expression* v, then **res** is bound in Γ' as an alias for v. In other words, we would have $(\mathbf{res} : \tau := v) \in \Gamma'$. (The grammar of pure expressions is formalized further in Section 4.6.)

Projection of Context Components. We define two projection functions. For a context $\Gamma = \langle E \mid F \rangle$, the projection "$\Gamma.\mathsf{pure}$" returns E, and the projection "$\Gamma.\mathsf{linear}$" returns F.

Syntax for Contexts with One Component. As syntactic sugar, we define $[x_0 : \tau_0, ..., x_n : \tau_n]$ as $\langle x_0 : \tau_0, ..., x_n : \tau_n \mid \varnothing \rangle$, for contexts that are entirely pure. Furthermore, we allow ourselves to write F to mean $\langle \varnothing \mid F \rangle$, where F denotes a set of linear resources.

Separated Conjunction of Two Contexts. We write $F_1 \star F_2$ the disjoint union of two sets of linear resources. Furthermore, for two contexts Γ_1 and Γ_2 with disjoint domains, we define $\Gamma_1 \circledast \Gamma_2$ as $\langle \Gamma_1.\mathsf{pure}, \Gamma_2.\mathsf{pure} \mid \Gamma_1.\mathsf{linear} \star \Gamma_2.\mathsf{linear} \rangle$, where the comma indicates list concatenation. Observe that $[E] \circledast F = \langle E \mid F \rangle$.

Pointwise Operators Over Linear Contexts. Consider a set of resources F of the form $(y_0 : H_0, ..., y_n : H_n)$. We define $\bigstar_{i \in r} F$ as $(y_0 : \bigstar_{i \in r} H_0, ..., y_n : \bigstar_{i \in r} H_n)$, that is, the iterated separating conjunction distributes pointwise over the set of linear resources. Similarly, we define αF as $(y_0 : \alpha H_0, ..., y_n : \alpha H_n)$.

Filtering on Contexts. We define a filtering operation, written $G \Vdash X$, where G is a set of resources (linear or pure) and X is a set of variable names. This operation computes a set of resources where only the entries from G whose name belongs to the set X are kept. Filtering also applies to contexts: $\langle E \mid F \rangle \Vdash X$ is defined as $\langle E \Vdash X \mid F \Vdash X \rangle$.

Specialization of Contexts. The specialization operation is used for example to specialize the contract of a function for a specific call to that functions. The contract is then specialized on the arguments, as well as on the ghost arguments, on which the function is applied. In case of a polymorphic function, type arguments are specialized as well. The specialization operation takes the form $\mathrm{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$. The definition of this operation is fairly technical, yet it is a direct generalization of the process of typechecking function applications in higher-order logics. Rather than showing the technical details, let us illustrate the specialization operation on an example.

Consider a function f whose input is described by a context $\Gamma \equiv \langle A : \mathsf{Type}, C : \mathsf{Type}, n : \mathsf{int}, p : \mathsf{ptr}(A), b : A, c : C \mid p \rightsquigarrow \mathsf{Matrix1}_A(n) \rangle$, where A and C are type arguments, where p and n denote physical arguments, and where b and c are ghost arguments. Consider a function call of the form $f(7, q)$, where q is a program variable of type $\mathsf{ptr}(\mathsf{int})$ in scope at the call site. This call specializes n to 7 and p to q, hence it is described by a substitution $\sigma \equiv (n := 7, p := q)$. Let Γ_0 be the context describing the pure variables bound at the call site. In particular, we have $(q : \mathsf{ptr}(\mathsf{int})) \in \Gamma_0$. For the example considered, the specialization operation yields the context: $\langle C : \mathsf{Type},\ b : \mathsf{int}, c : C \mid q \rightsquigarrow \mathsf{Matrix1}_A(7) \rangle$. Observe how the types and arguments being specialized (namely A, n and p) are eliminated from the pure part of the context, and the corresponding values (namely int, 7 and q) are substituted in the entities that remain.

Renaming on Contexts. A renaming operation is involved when the programmer explicitly specifies the names to assign to the ghost variables obtained as part of the result of a function call. The operation $\mathrm{Rename}\{\rho\}(\Gamma)$ renames certain keys from Γ. Here, ρ denotes a map that associates resource names to other resource names. The keys from ρ may or may not be bound in Γ. The values

$$\begin{array}{rcl}
\{\} & \mathsf{alloc}_{\hat{\tau}}() & \{[\mathbf{res} : \mathsf{ptr}(\hat{\tau})] \circledast \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\} \\
\{[a : \mathsf{ptr}(\hat{\tau}), \alpha : \mathsf{frac}] \circledast \alpha(a \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\} & \mathsf{get}_{\hat{\tau}}(a) & \{[\mathbf{res} : \hat{\tau}] \circledast \alpha(a \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\} \\
\{[a : \mathsf{ptr}(\hat{\tau}), b : \hat{\tau}] \circledast \mathsf{Uninit}(a \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\} & \mathsf{set}_{\hat{\tau}}(a, b) & \{a \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}\} \\
\{[a : \mathsf{ptr}(\hat{\tau})] \circledast \mathsf{Uninit}(a \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\} & \mathsf{free}_{\hat{\tau}}(a) & \{\} \\
\{[b : \hat{\tau}]\} & \mathsf{ref}_{\hat{\tau}}(b) & \{[\mathbf{res} : \mathsf{ptr}(\hat{\tau})] \circledast \mathbf{res} \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}\} \\
\{\} & \mathsf{ref_uninit}_{\hat{\tau}}() & \{[\mathbf{res} : \mathsf{ptr}(\hat{\tau})] \circledast \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_{\hat{\tau}})\}
\end{array}$$

Fig. 16. Contracts assigned to key primitive functions; $\hat{\tau}$ denotes a C type; a and b denote program variables.

from ρ must be fresh from Γ. For example, $\mathrm{Rename}\{x := x', y := y'\}(\langle E_1, x : \tau, E_2 \mid F\rangle)$, where y has no occurrence in E_1, E_2 or F, evaluates to $\langle E_1, x' : \tau, \mathrm{Subst}\{x := x'\}(E_2) \mid \mathrm{Subst}\{x := x'\}(F)\rangle$. As another example, $\mathrm{Rename}\{y := y'\}(\langle E \mid F_1, y : H, F_2\rangle)$ evaluates to $\langle E \mid F_1, y' : H, F_2\rangle$.

4.3 Grammar of Contracts

Every function and loop carries a contract to guide the typechecker. We next detail the grammar of contracts.

Function Contracts. A function definition annotated with a *function contract* γ takes the form $\mathbf{fun}(a_1, ..., a_n)_\gamma \mapsto t$. The contract γ consists of two contexts, one for the *precondition*, written $\gamma.\mathrm{pre}$, and one for the *postcondition*, written $\gamma.\mathrm{post}$. Intuitively, a function f with arguments named a_i and with contract γ satisfies the Separation Logic triple $\{\gamma.\mathrm{pre}\}\ f(a_1, ..., a_n)\ \{\gamma.\mathrm{post}\}$. This property is formally captured by the proposition $\mathrm{Spec}(f, [a_1, ..., a_n], \gamma)$, which may appear in contexts.

Technically, a function γ takes the form $\{\mathrm{pre} = \Gamma_{pre}\ ;\ \mathrm{post} = \Gamma_{post}\}$. The precondition Γ_{pre} must contain all the formal parameters a_i, and may refer to any of the free variables in scope. The postcondition Γ_{post} may also refer to all these variables, as well as to the pure variables bound in the precondition Γ_{pre}.

Contracts for Primitive Functions. Fig. 16 gives the contracts that we axiomatize for the operations on heap and stack cells—technically, we present not their contracts but the triples derived from their contracts, to improve readability. The C language does not feature polymorphism, hence we describe in the figure a family of contracts, indexed with the type $\hat{\tau}$ of the memory cell at hand. (For polymorphic operators, there would be a single contract, which would include a quantification over the type $\hat{\tau}$ in the input environment.)

These contracts illustrate, in particular, how the bindings on **res** appears in output contexts. An allocation produces an uninitialized permission. A write requires an uninitialized permission and returns a full permission. A read requires a read-only permission and returns it. A free operation requires a full permission and does not return it. Recall that a full permission can be split into read-only resources, and that it may be downgraded at any time into an uninitialized permission. The operation on stack cells are similar.

Contracts for arithmetic operations are described later on, in Section 4.6.

Contracts for Ghost Functions. In addition to contracts for primitive heap-manipulating functions, OptiTrust provides contracts for primitive ghost functions. For example, consider the ghost function swap_groups, which was involved in the OpenCV case study (Section 2.1) for swapping two iterators (iterated separating conjunctions), and which typically needs to be involved on every loop-swap operation (as explained further in Section 6.5). It is specified shown below, where H is a heap predicate that depends on the two indices i and j.

$$\{[r_i : \text{range}, r_j : \text{range}, H : (\text{int}, \text{int}) \xrightarrow{\text{pure}} \text{Hprop}] \circledast \big(\bigstar_{i \in r_i} \bigstar_{j \in r_j} H(i,j) \big)\}\ \text{swap\_groups}\ \{\bigstar_{j \in r_j} \bigstar_{i \in r_i} H(i,j)\}$$

The OptiTrust user can define custom ghost functions by composing existing ghost functions. Doing so can be useful to factorize repetitive resource-manipulation patterns. Ghost functions are written and typechecked like regular C functions. Their specificity is that they contain only calls to ghost functions and control-flow structures such as for-loops.

Loop Contracts. A for-loop annotated with a *loop contract* χ takes the form **for** $(i \in r)_\chi\{t\}$. The loop contract χ consists of a record structured as follows.

$$\begin{cases} \text{vars} = E & \text{Pure variables, scoping over the other contract components} \\ \text{excl} = \begin{cases} \text{pre} = F_{pre} & \text{Resources consumed exclusively by one iteration} \\ \text{post} = F_{post} & \text{Resources produced exclusively by one iteration} \end{cases} \\ \text{shrd} = \begin{cases} \text{reads} = F_{reads} & \text{Read only resources shared between iterations} \\ \text{inv} = F_{inv} & \text{Sequential invariant, threaded through iterations} \end{cases} \end{cases}$$

We call E the *loop ghost variables.* The variables from E scope over F_{pre}, F_{post}, F_{reads} and F_{inv}. We call F_{pre} the *consumed per-iteration resources* and F_{post} the *produced per-iteration resources.* Resources in F_{pre} and in F_{post} may (and typically do) refer to the loop index. We call F_{reads} the *shared reads.* In practice, this set of resources consists of read-only resources. Resources in F_{reads} cannot refer to the loop index. We call F_{inv} the *sequential invariant.* It corresponds to a standard loop invariant in sequential Separation Logic. In this paper, we consider for simplicity that F_{inv} does not depend on the loop index.

As we will see later in Section 4.7, when typechecking a loop of the form **for** $(i \in r)_\chi\{t\}$, the loop body t is typechecked in a context that binds an index i of type int, a hypothesis of type $i \in r$, the variables from E, the resources F_{pre}, (subfractions of) the resources in F_{reads}, and the resources in F_{inv}. The loop body needs to produce the resources F_{post}, and it needs to give back the resources that it had received from F_{reads} and from F_{inv}.

Parallel Loop Contracts. A loop is parallelizable if it can be typechecked with an empty sequential invariant F_{inv}. Hence, we say that a loop contract χ is *parallelizable*, and write parallelizable(χ), when χ.shrd.inv $= \varnothing$.

4.4 Entailment

We next introduce the *entailment* judgment, written $\Gamma \Rrightarrow \Gamma'$. The entailment judgment is used in particular to assert that the set of resources obtained at a given program point corresponds to the set of resources expected at that same point. For example, the set of resources at the end of a function body corresponds to the set described by the postcondition of this function.

The literature on Separation Logic includes two types of entailment: *linear* and *affine* entailment relations. OptiTrust is based on a linear entailment relation, disallowing resources to be silently "dropped". The benefits of using linear entailment is that it allows checking the absence of memory leaks—every piece of heap allocated data must eventually be freed.

The OptiTrust entailment between two contexts:

$$\langle x_0 : \tau_0, \ldots, x_n : \tau_n \mid y_0 : H_0, \ldots, y_m : H_m \rangle \Rrightarrow \langle x'_0 : \tau'_0, \ldots, x_{m'} : \tau_{m'} \mid y'_0 : H'_0, \ldots, y'_{m'} : H'_{m'} \rangle$$

is defined as

$$\forall x_0 : \tau_0,\ \ldots,\ \forall x_n : \tau_n, \qquad H_0 \star \ldots \star H_m \overset{SL}{\Rightarrow} (\exists x'_0 : \tau'_0,\ \ldots,\ \exists x'_m : \tau'_m,\ H'_0 \star \ldots \star H'_{m'})$$

where $\overset{SL}{\Rightarrow}$ denotes the standard Separation Logic entailment.

For example, the following entailments hold:

- $\langle x : \mathsf{loc} \mid x \rightsquigarrow \mathsf{Cell} \rangle \Rightarrow \langle \alpha : \mathsf{frac} \mid \alpha(x \rightsquigarrow \mathsf{Cell}), (1-\alpha)(x \rightsquigarrow \mathsf{Cell}) \rangle$
- $\langle y : \mathsf{loc} \mid y \rightsquigarrow \mathsf{Cell} \rangle \Rightarrow \langle \mid \mathsf{Uninit}(y \rightsquigarrow \mathsf{Cell}) \rangle$
- $\langle A : \mathsf{loc}, n : \mathsf{int}, m : \mathsf{int} \mid \bigstar_{i \in 0..n} \bigstar_{j \in 0..m} A[i][j] \rightsquigarrow \mathsf{Cell} \rangle \Rightarrow \langle \mid \bigstar_{j \in 0..m} \bigstar_{i \in 0..n} A[i][j] \rightsquigarrow \mathsf{Cell} \rangle$
- $\langle n : \mathsf{int}, n \text{ even} \mid \rangle \Rightarrow \langle m : \mathsf{int}, n = 2m \mid \rangle$

However, the entailment $\langle x : \mathsf{loc} \mid x \rightsquigarrow \mathsf{Cell} \rangle \Rightarrow \langle \mid \rangle$ does not hold because linear resources cannot be dropped.

As a shorthand, we write $\Gamma \Leftrightarrow \Gamma'$ to assert that entailment holds both ways, that is, to assert that the conjunction $(\Gamma \Rightarrow \Gamma') \wedge (\Gamma' \Rightarrow \Gamma)$ holds.

4.5 Subtraction

The *subtraction* operation provides a sound yet incomplete algorithmic implementation of the entailment judgment. The subtraction operation not only allows checking the validity of an entailment, it also enables a certain amount of inference. At a high level, given Γ and Γ', the subtraction operation computes the *frame*, written F, which denotes the set of linear resources such that $\Gamma \Rightarrow \Gamma' \star \langle \varnothing \mid F \rangle$. The subtraction operation also infers the instantiation map σ providing the witnesses for the instantiations of the variables that are bound (and therefore existentially quantified) in Γ'. Such a subtraction operator is found in most—if not all—practical verification frameworks based on Separation Logic.

The typing rules of OptiTrust actually make use of two variants of the subtraction operation. The *core subtraction operation*, written $\Gamma \boxminus \Gamma'$, is able to convert uninitialized resources into full resources on-the-fly, however it does not support splitting read-only resources on-the-fly. The *carving subtraction operation*, written $\Gamma \ominus \Gamma'$, extends the former with the feature of carving out a fraction of a read-only permission from Γ every time a corresponding read-only permission is requested in Γ'. (Carving was described in Section 4.1.)

The core subtraction operation $\Gamma \boxminus \Gamma'$ is formally specified as a partial operation. It may fail (that is, return $\bot$) if a resource in Γ' cannot be matched against a corresponding resource in Γ. Otherwise, the operation returns a result of the form (σ, F). When $\Gamma \boxminus \Gamma' = (\sigma, F)$, then the entailment $\Gamma \Rightarrow \mathsf{Specialize}_\Gamma\{\sigma\}(\Gamma') \star \langle \varnothing \mid F \rangle$ holds. In particular, the subtraction operation can be used to prove an entailment $\Gamma \Rightarrow \Gamma'$, by checking that $\Gamma \boxminus \Gamma'$ evaluates to $(\sigma, \varnothing)$ for some σ.

The subtraction operation is implemented following a standard scheme.

(1) The substitution map σ is initialized with bindings that associates each of the pure variables of Γ' to a fresh unification variable.
(2) Each of the linear resources from Γ' are matched against a corresponding resource from Γ. This process may trigger unifications, resulting in partial or total resolution of certain unification variables.
(3) If Γ' requests a linear resource of the form $\mathsf{Uninit}(H)$, and if Γ contains the resource H, then our algorithm applies an on-the-fly weakening from H to $\mathsf{Uninit}(H)$.
(4) The items from Γ that remains at the end are assigned to the frame F.

The carving subtraction operation $\Gamma \ominus \Gamma'$ behaves almost like $\Gamma \boxminus \Gamma'$ but outputs a triple (E_{frac}, σ, F) where σ and F are the same as in core subtraction and E_{frac} is a pure context for generated fractions containing only bindings of the form $\alpha : \mathsf{frac}$ and initially empty. Compared to the core subtraction, the carving subtraction refines step (2) as follows. If Γ' requests a fractional resource αH, if α is an unconstrained unification variable that denotes a fraction, and if Γ contains a fractional resource $\beta H'$ for some fraction β and where H unifies with H', then our algorithm applies an on-the-fly splitting operation to convert $\beta H'$ into the conjunction of $\alpha' H'$ and $(\beta - \alpha')H'$ for a

$$
\begin{array}{llll}
v & ::= & x & \text{Variable} \\
 & | & n \quad | \quad b & \text{Integer or boolean literal} \\
 & | & P \quad | \quad H \quad | \quad \hat{\tau} & \text{Mathematical proposition, heap predicate, or C type} \\
 & | & \mathbf{fun}(x_1 : \tau_1, ..., x_n : \tau_n) \mapsto v & \text{Pure function definition} \\
 & | & v_0(v_1, ..., v_n) & \text{Pure function application}
\end{array}
$$

$$
\textsc{Var}\ \frac{(v : \tau) \in E}{E \vdash v : \tau}
\qquad
\textsc{Int}\ \frac{}{E \vdash n : \mathsf{int}}
\qquad
\textsc{Bool}\ \frac{}{E \vdash b : \mathsf{bool}}
\qquad
\textsc{IntType}\ \frac{}{E \vdash \mathsf{int} : \mathsf{Type}}
\qquad
\textsc{BoolType}\ \frac{}{E \vdash \mathsf{int} : \mathsf{Type}}
$$

$$
\textsc{Prop}\ \frac{P \text{ is a mathematical proposition} \atop \text{with free variables in } E}{E \vdash P : \mathsf{Prop}}
\qquad
\textsc{Hprop}\ \frac{H \text{ is a heap predicate} \atop \text{with free variables in } E}{E \vdash H : \mathsf{Hprop}}
\qquad
\textsc{PtrType}\ \frac{E \vdash A : \mathsf{Type}}{E \vdash \mathsf{ptr}(A) : \mathsf{Type}}
$$

$$
\textsc{PureFun}\ \frac{(E, x_1 : \tau_1, ..., x_n : \tau_n) \vdash v : \tau}{E \ \vdash\ \mathbf{fun}(x_1 : \tau_1, ..., x_n : \tau_n) \mapsto v \ :\ (\tau_1, ..., \tau_n) \xrightarrow{\mathsf{pure}} \tau}
\qquad
\textsc{PureApp}\ \frac{E \ \vdash\ v_0 \ :\ (\tau_1, ...\tau_n) \xrightarrow{\mathsf{pure}} \tau \qquad \forall i \in [1..n],\ \ E \vdash v_i : \tau_i}{E \vdash v_0(v_1, ..., v_n) : \tau}
$$

Fig. 17. Grammar of pure expressions, written v, and associated typing judgment, written $E \vdash v : \tau$. Arithmetic operations such as $v_1 + v_2$ are viewed as functions calls and are therefore handled by the rule PureApp.

fresh α' added to E_{frac}. Then it adds the binding $\alpha := \alpha'$ in σ. Doing so ensures that a fraction of a piece of βH remains available in Γ, allowing to eliminate other resources of the form $\alpha'' H$ that might appear in the remaining elements from Γ'.

4.6 Typechecking of Pure Expressions

A *pure expression* (a.k.a. *logical expression*), written v, is an expression whose evaluation terminates and whose value does not depend on the memory state. Pure expressions, unlike stateful expressions, may appear in specifications and invariants. Pure expressions include program variables (which are always immutable in the OptiTrust AST), constant literals, mathematical propositions, heap predicates, C types, pure functions definitions, and pure function calls. Figure 17 formalizes the grammar of pure expressions, and defines their typing judgment, written $E \vdash v : \tau$, where E is a pure context—that is, the pure part of a context, in the sense of Section 4.2.

An arithmetic expression $v_1 + v_2$ can be considered as a pure expression if its two arguments are pure. The contract for addition is: $\{[a : \mathsf{int}, b : \mathsf{int}]\}\ (a + b)\ \{[\mathbf{res} := a \mathbin{\hat{+}} b : \mathsf{int}]\}$, where $+$ denotes the addition operator from the programming language, and where $\hat{+}$ denotes the corresponding addition operator from the logic. Partial functions may also be treated as pure expressions, simply with an additional precondition. The contract for division is: $\{[a : \mathsf{int}, b : \mathsf{int}, b \neq 0]\}\ (a/b)\ \{[\mathbf{res} := a \mathbin{\hat{/}} b : \mathsf{int}]\}$. where $\hat{/}$ denotes the logical integer division operator. Following standard practice in proof assistants, the operator $\hat{/}$ is defined in the logic as a total function that returns unspecified results when dividing by zero.

4.7 Typechecking of Terms

Our typing judgment takes the form $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, capturing the fact that, in context Γ, the term t is well typed and produces a context Γ' with a *usage map* Δ. We are interested in describing the *algorithmic* typing rules exploited by OptiTrust. Our typing algorithm takes Γ and t as input, and produces Γ' and Δ as output. The refinement with usage maps will be discussed further in Section 5.3. For now, we focus on describing typing rules for the judgment $\{\Gamma\}\ t\ \{\Gamma'\}$.

In general, in a valid triple $\{\Gamma\}\ t\ \{\Gamma'\}$, variables from the postcondition Γ' may refer to variables from the precondition Γ. For the purpose of the algorithmic typechecking, however, we design the typing rules in such a way that Γ' is always *closed*, meaning that variable occurrences in Γ' refer to variables previously bound in Γ'. The purpose of this design decision is to maximize the amount of information that is propagated forward during the typechecking.

In particular, in the algorithmic typechecking, all the logical bindings (ghost variables and pure facts) from Γ are reproduced in Γ'. The pure bindings that appear in Γ' but not in Γ correspond either (1) to the binding for **res**, which denotes the result value produced by t, as explained in Section 4.3; or (2) to logical bindings (ghost variables and pure facts) that correspond to existentially quantified variables and pure postconditions.

Unlike pure bindings, the linear bindings of Γ' may be arbitrarily modified compared with those in Γ. The modifications reflect the side effects performed by t. Linear resources that are bound with the same name in Γ' as in Γ necessarily correspond to resources that have not been modified by t.

Figure 18 presents our typing rules. The typing rule for applications handles the particular case where the subterms are program variables (i.e., functions calls in A-normal form)—the processing of effectful subterms depends on resource usage, and is explained further in Section 5.5. The soundness of these rules stems from the fact that they correspond to an algorithmic reformulation of the standard reasoning rules from Separation Logic. We next describe the rules individually.

Program Values. Consider a program value v. In its triple, of the form $\{\Gamma\}\ v\ \{\Gamma'\}$, the output context Γ' is obtained by extending Γ with an alias binding from **res** to v. Alias bindings were defined in Section 4.2. The type of v is computed by means of the typing judgment for pure expressions, defined in Section 4.6.

Let-Bindings. Consider an instruction of the form **let** $x = t$. Recall from Section 3.1 that such instructions only appear in sequences. The subexpression t produces a value, hence the output context Γ_1 associated with t binds the special variable **res**. The expression **let** $x = t$ itself does not produce a value, hence its output context Γ_2 does not bind **res**. However, the output context Γ_2 is extended with a binding on x. Concretely, Γ_2 is obtained by replacing in Γ_1 the bound name **res** with the bound name x.

Sequence of Instructions. We decompose the treatment of sequences in two rules: a first rule named Seq for handling the sequence of instructions per se, and a second rule named Block for handling the disposal of stack-allocated variables. The rest of this paragraph describes the Seq rule.

Consider a sequence of instructions t_i. Starting from an input context Γ_0, each subterm t_i makes the context evolves from Γ_{i-1} to Γ_i. There are two particular cases to handle. First, if a t_i is a non-void expression, then the output context of t_i features a **res** binding. This binding is materialized as a binding over a fresh ghost variable x_i in the final output context Γ_r. Second, if one of the t_i instruction is of the form **let res** $= t_i'$, then the final context Γ_r is patched to include a corresponding **res** binding, instead of a binding on a ghost variable x_i.

Scope Blocks. The typing rule Block is responsible for collecting the resources that corresponds to stack-allocated variables, when reaching the end of a sequence, that is, the end of their scope. Recall from Section 3.1 that stack allocation takes the form **let** $x = \mathsf{ref}(T)$ or **let** $x = \mathsf{ref\_uninit}()$, with such instructions occurring directly within a sequence. The auxiliary function $\mathsf{StackAllocCells}(t_1, \ldots, t_n)$ synthesizes, based on the syntax of the terms t_i that appear in the sequence at hand, a conjunction of resources, each of the form $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell}_\tau)$. These resources are subtracted from the set of resources available at the end of the sequence. Crucially, the subtraction operation checks

$$\dfrac{\Gamma.\mathsf{pure} \vdash v : \tau}{\{\Gamma\}\ v\ \{\Gamma \circledast [\mathbf{res} : \tau := v]\}}\ \textsc{Val} \qquad \dfrac{\{\Gamma_0\}\ t\ \{\Gamma_1\} \qquad \Gamma_2 = \mathsf{Rename}\{\mathbf{res} := x\}(\Gamma_1)}{\{\Gamma_0\}\ \mathbf{let}\ x = t\ \{\Gamma_2\}}\ \textsc{Let}$$

$$\dfrac{\begin{array}{c}\forall i \in [1, n].\quad x_i\ \mathsf{fresh} \quad\wedge\quad \{\Gamma_{i-1}\}\ t_i\ \{\Gamma_i'\} \quad\wedge\quad \Gamma_i = \mathsf{Rename}\{\mathbf{res} := x_i\}(\Gamma_i') \\ \Gamma_r = \begin{cases}\mathsf{Rename}\{x_i := \mathbf{res}\}(\Gamma_n) & \text{if } t_i \text{ is of the form “}\mathbf{let\ res} = t_i'\text{”} \\ \Gamma_n & \text{otherwise}\end{cases}\end{array}}{\{\Gamma_0\}\ \left(t_1; \ldots; t_n\right)\ \{\Gamma_r\}}\ \textsc{Seq}$$

$$\dfrac{\{\Gamma_0\}\ \left(t_1; \ldots; t_n\right)\ \{\Gamma_r\} \qquad (\varnothing, F) = \Gamma_r \boxminus \mathsf{StackAllocCells}(t_1, \ldots, t_n)}{\{\Gamma_0\}\ \left\{t_1; \ldots; t_n\right\}\ \{\langle \Gamma_r.\mathsf{pure} \mid F\rangle\}}\ \textsc{Block}$$

$$\dfrac{\begin{array}{c}\{[\Gamma_0.\mathsf{pure}] \circledast \gamma.\mathsf{pre}\}\ t\ \{\Gamma_1\} \qquad (\_, \varnothing) = \Gamma_1 \boxminus \gamma.\mathsf{post} \\ (\mathbf{res} : \hat{\tau}_r) \in \gamma.\mathsf{post} \qquad \hat{\tau}_f\ =\ (\hat{\tau}_1, \ldots, \hat{\tau}_n) \to \hat{\tau}_r\end{array}}{\{\Gamma_0\}\ \left(\mathbf{fun}(a_1 : \hat{\tau}_1, \ldots, a_n : \hat{\tau}_n)_\gamma \mapsto t\right)\ \{\Gamma_0 \circledast [\mathbf{res} : \hat{\tau}_f, \mathsf{Spec}(\mathbf{res}, [a_1, \ldots, a_n], \gamma)]\}}\ \textsc{Fun}$$

$$\dfrac{\begin{array}{c}\mathsf{Spec}(x_0, [a_1, \ldots, a_n], \gamma) \in \Gamma_0 \\ (E_{frac}, \sigma', F) = \Gamma_0 \ominus \mathsf{Specialize}_{\Gamma_0}\{\overline{a_i := x_i}^{i \in [1,n]}, \sigma\}(\gamma.\mathsf{pre}) \\ \mathsf{dom}(\rho) = \mathsf{dom}(\gamma.\mathsf{post}) \qquad \mathsf{im}(\rho) \cap \mathsf{dom}(\Gamma_0) = \varnothing \\ \Gamma_q = \mathsf{Rename}\{\rho\}(\mathsf{Subst}\{\overline{a_i := x_i}^{i \in [1,n]}, \sigma, \sigma'\}(\gamma.\mathsf{post})) \\ \Gamma_r = \mathsf{CloseFracs}([\Gamma_0.\mathsf{pure}, E_{frac}] \circledast F \circledast \Gamma_q)\end{array}}{\{\Gamma_0\}\ x_0(x_1, \ldots, x_n)_{\sigma,\rho}\ \{\Gamma_r\}}\ \textsc{App}$$

$$\dfrac{\begin{array}{c}\Gamma_p = [\chi.\mathsf{vars}] \circledast (\bigstar_{i \in r}\, \chi.\mathsf{excl.pre}) \circledast \chi.\mathsf{shrd.reads} \circledast \chi.\mathsf{shrd.inv} \\ (E_frac, \sigma', F) = \Gamma_0 \ominus \Gamma_p \\ \Gamma_p' = [i : \mathsf{int}, i \in r] \circledast [\chi.\mathsf{vars}] \circledast \chi.\mathsf{excl.pre} \circledast \frac{1}{r.\mathsf{len}}\chi.\mathsf{shrd.reads} \circledast \chi.\mathsf{shrd.inv} \\ \{[\Gamma_0.\mathsf{pure}] \circledast \Gamma_p'\}\ t\ \{\Gamma_q'\} \\ (\_, \varnothing) = \Gamma_q' \boxminus \chi.\mathsf{excl.post} \circledast \frac{1}{r.\mathsf{len}}\chi.\mathsf{shrd.reads} \circledast \chi.\mathsf{shrd.inv} \\ \Gamma_q = \mathsf{Subst}\{\sigma'\}((\bigstar_{i \in r}\, \chi.\mathsf{excl.post}) \circledast \chi.\mathsf{shrd.reads} \circledast \chi.\mathsf{shrd.inv}) \\ \Gamma_r = \mathsf{CloseFracs}([\Gamma_0.\mathsf{pure}, E_{frac}] \circledast F \circledast \Gamma_q) \\ \mathsf{parallelizable}(\chi) \ \vee\ \pi \neq \mathsf{parallel}\end{array}}{\{\Gamma_0\}\ \mathbf{for}^{\,\pi}(i \in r)_\chi\ t\ \{\Gamma_r\}}\ \textsc{For}$$

$$\dfrac{\begin{array}{c}\{\Gamma_0\}\ t_0\ \{\Gamma_0'\} \qquad \{\mathsf{Learn}\{\mathbf{res} = \mathsf{true}\}(\Gamma_0')\}\ t_1\ \{\Gamma_1\} \qquad \{\mathsf{Learn}\{\mathbf{res} = \mathsf{false}\}(\Gamma_0')\}\ t_2\ \{\Gamma_2\} \\ (\_, \varnothing) = \Gamma_1 \boxminus \Gamma_3 \qquad (\_, \varnothing) = \Gamma_2 \boxminus \Gamma_3\end{array}}{\{\Gamma_0\}\ \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \{\Gamma_3\}}\ \textsc{If}$$

Fig. 18. Algorithmic typing rules for establishing triples of the form $\{\Gamma\}\ t\ \{\Gamma'\}$. These rules are generalized in Section 5.3 to derive triples the form $\{\Gamma\}\ t^\Delta\ \{\Gamma'\}$, where Δ describes the resource usage.

that the resources indeed appear in the current resource set. Doing so ensures, in particular, that the address of a stack-allocated piece of data was not subject to a prior call to free.

Function Definition. Consider a function definition $\mathbf{fun}(a_1 : \hat{\tau}_1, \ldots, a_n : \hat{\tau}_n)_\gamma \mapsto t$, with arguments a_i of type $\hat{\tau}_i$, with body t, and with contract γ. Recall from Section 4.3 that the function contract consists of a precondition $\gamma.\mathsf{pre}$ and a postcondition $\gamma.\mathsf{post}$, both described as contexts. The function closure being built may capture free variables from the current context. In the rule, the pure variables from the current context are described as $\Gamma_0.\mathsf{pure}$. Note, however, that a function closure is not allowed to capture linear resources. Hence, the body of the function is typechecked in an environment that consists of the conjunction of Γ_0.pure and γ.pre. Ultimately, the body of the function must produce a set of resources Γ_r that entails the postcondition γ.post. The postcondition includes on **res**, which admits a function type, and a hypothesis Spec(**res**, $[a_1, ..., a_n]$, γ) that describes the specification of the function closure being created. As explained earlier in Section 4.3, this hypothesis captures $\{\gamma.\mathrm{pre}\}$ **res**$(a_1, ..., a_n)$ $\{\gamma.\mathrm{post}\}$, which is indeed the triple intended for the closure named **res**.

Function Applications. Consider a function application of the form $x_0(x_1, ..., x_n)$, where the x_i are program variables. (The general form will be discussed in section 5.5.) To typecheck it, the input context Γ_0 must contain an entry of the form Spec(x_0, $[a_1, ..., a_n]$, γ) for the function x_0. This same context Γ_0 must entail the precondition γ.pre, specialized for the arguments x_i by means of the Specialize operations defined in Section 4.2. This entailment is checked by means of the carving subtraction operation defined in Section 4.5. The subtraction produces a frame F that contains the resources from Γ_0 that are not used by the function call, and produces a substitution named σ' that describes the instantiation of the ghost arguments and resources. The final postcondition Γ_q is obtained by considering the postcondition γ.post, adding the frame F and Γ_0.pure, then invoking the CloseFracs operation described in Section 4.1 for eagerly recombininig carved-out fractions.

Two additional technicalities are involved in the statement of the App rule. They correspond to the handling of optional user-provided annotations, name σ and ρ, that may guide the type-checking of an application. Such annotations are commonly found both in proof assistants and in program verification frameworks. The map σ corresponds to a for instantiating a subset of the ghost arguments. Indeed, there could be situations where the subtraction operation would fail to infer a unique possible instantiation, by the only means of the unification process. Hence, user annotations are required to resolve the instantiation. The map ρ corresponds to a renaming map. Its purpose is to allow the user to specify custom names for the ghost bindings and for the linear resources that are produced by the postcondition. In all our case studies, we did not need ρ, and σ was only used on ghost calls, however we anticipate for the general case. Typically, a ghost result produced by a function call is explicitly named using the ρ annotation, in order to provide this same ghost result as part of a σ annotation for a subsequent function call.

Simple for-loops. Consider a possibly parallel, simple for-loop of the form $\mathbf{for}^{\pi}(i \in r)_{\chi}\, t$ The typechecking of such a loop, driven by the loop contract annotation χ, was already discussed at a high level at the end of Section 4.3. Let us comment on the three remaining aspects. First, the shared-read resources, described by χ.shrd.reads, are split into $\frac{1}{r.\mathrm{len}}$ subfractions, where r.len denotes the number of iterations associated with the range r. Note that, when typecking the body of the loop for a particular iteration $i \in r$, the denominator r.len can be assumed to be nonzero—indeed, $i \in r$ is equivalent to $0 \le i < r.\mathrm{len}$. Second, like for function calls, the instantiation of the contract using the resources from the input environment Γ_0 is computed using a subtraction, involving a frame F as well as an instantiation map σ'. Also, like for function calls, the output context is obtained by invoking the CloseFracs operation. Third, loops, like functions calls, feature optional annotations σ and ρ, which we have omitted from the statement of the rule, for simplicity. The map σ guides how the contract is instantiated in the input environment Γ_0. The map ρ can be used to explicit the names associated with the resources produced by the loop. The two maps are handled in a similar way as in the App rule.

Conditionals. Consider a conditional of the form **if** t_0 **then** t_1 **else** t_2. The condition t_0 is evaluated in the input context Γ_0 and produces a context Γ_0'. Then, both branches t_1 and t_2 need to typecheck in the context Γ_0'. This context needs to be patched to reflect the knowledge that t_0 evaluated to either true or false, depending on the branch. The patch is implemented by means of the operation Learn$\{\mathbf{res} = b\}(\Gamma)$. This operation applies the following three steps.

(1) If an aliasing binding of the form $\mathbf{res} := v : \text{bool}$ appears in Γ, then the operation replaces this binding with a conventional binding $\mathbf{res} : \text{bool}$, and extends Γ with an equality $[\mathbf{res} = v]$.
(2) It specializes the variable **res** with b, that is, it removes the binding $\mathbf{res} : \text{bool}$, and replaces all occurrences of **res** with the boolean value b.
(3) It applies basic simplifications on the expressions in which **res** has been substituted with b.

For example, assume t_0 is a test of the form `x == y`, and consider the evaluation of Learn$\{\mathbf{res} = \text{true}\}(\Gamma_0')$. The output context of t_0 contains the alias binding $\mathbf{res} := (\texttt{x==y}) : \text{bool}$. At step (1), this binding is replaced with an equality $\mathbf{res} = (\texttt{x==y})$. At step (2), **res** is replaced with true, hence the equality becomes $\text{true} = (\texttt{x==y})$. At step (3), this hypothesis is rewritten as the logical equality $\texttt{x} = \texttt{y}$.

The then-branch t_1 produces an output context Γ_1, and likewise the else-branch t_2 produce an output context Γ_2. The challenge, typical in program logics, is to find a *join* context Γ_3 that both Γ_1 and Γ_2 entail. In the general case, this context Γ_3 cannot be automatically inferred. It needs to be provided, at least in part, via a *join contract*. In practice, numerous heuristics can be devised to synthesis a join contract, to save the need for users to provide explicit join contract. In particular, whenever the if-statement is the last instruction of a function body or a loop body, the join contract is provided by the expected postcondition coming from the function contract or the loop contract. All our case studies followed this pattern. We leave it to future work to experiment with heuristics for join contracts.

5 COMPUTING PROGRAM RESOURCES

The first goal of this section is to formalize the usage maps, written Δ, and to generalize triples from the form $\{\Gamma\}\ t\ \{\Gamma'\}$ to the form $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$. Section 5.1 presents the grammar of usage maps. Section 5.2 presents operations on usage maps. Section 5.3 explains how usage maps are computed by our typing algorithm.

The second goal of this section is to formalize the *triple minimization* operations, which plays a central role in the typechecking of function calls involving effectful subexpressions. Triple minimization will also be useful later on to minimize the loop contracts produced by transformations. Section 5.4 presents the triple minimization procedure. Section 5.5 presents the typing rule for subexpressions—this typing rule applies as a preprocessing before the App rule presented earlier.

5.1 Grammar of Usage Maps

A *usage map*, written Δ, is an association map that binds resource names to *usage kinds*. For a *pure* resource name, there are 2 possible usage kinds: required and ensured. For a *linear* resource name, there are 5 possible usage kinds: full, uninit, splittedFrac, joinedFrac and produced. In a triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, the usage map Δ binds names of resources that can be bound in Γ or Γ', or possibly in both. The usage map Δ only binds names of resources that are effectively manipulated by t. (In other words, the *framed* resources are omitted from usage maps.) Let us now explain the meaning of each possible binding in a usage map Δ associated with the triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$.

- "x : required" means that x is a pure resource in Γ that was used during the typing of t.
- "x : ensured" means that x is a pure resource added to the context Γ' during the typing of t. In such a situation, x is not bound in Γ.
- "y : full" can arise when Γ contains a linear resource "$y : H$", for some predicate H. The usage "y : full" means that this resource is consumed during the typing of t. As a result y is not bound in Γ'. Even if t produces a linear resource with the same predicate H, this new occurrence of H is assigned a fresh name, distinct from y.

- "y : uninit" is similar to "y : full" but moreover captures the information that t needs not read the original contents of the memory cells associated with the resource named y. In particular, if t performs a write operation in a cell y before any read operation on y, then the usage of y is uninit.
- "y : splittedFrac" can arise when Γ contains a splittable linear resource "$y : H$", for some predicate H. The usage "y : splittedFrac" means that t uses an unspecified subfraction of this resource. In such a situation, the name y is bound both in Γ and in Γ'. It may be the case, however, that the resource named y carries different fractions in Γ and Γ'.
- "y : joinedFrac" can arise when Γ contains a linear resource of the form "$y : (\alpha - \beta_1 - ... - \beta_n)H$". The usage "$y$: joinedFrac" means that: (1) the linear resource named y is not used by t, and (2) t produced a resource of the form $(\beta_i - \gamma_1 - ... - \gamma_m)H$, and (3) these two resources are merged and the result appears in Γ' under the name y. If a single merge operation is applied, then the resulting resource is $y : (\alpha - \beta_1 - ... - \beta_{i-1} - \gamma_1 - ... - \gamma_m - \beta_{i+1} - ... - \beta_n)H$. (Recall Section 4.1)
- "y : produced" means that the linear resource y has been produced by t. In this case, y is the name of a linear resource in Γ', and does not occur in Γ.
- If a resource name is bound in Γ but not in Δ, then its absence indicates that the corresponding resource is not touched by t. Such a resource is bound under the same name in Γ and Γ'.

5.2 Operations on Usage Maps

Projections of Usage Maps. We define Δ.full as the set of names y such that "y : full" appears in Δ. Likewise, we define Δ.required, Δ.ensured, Δ.uninit, Δ.splittedFrac, Δ.joinedFrac and Δ.produced. In addition, we define the following operations.

$$
\begin{aligned}
\Delta.\mathsf{consumed} &= \Delta.\mathsf{full} \cup \Delta.\mathsf{uninit} \\
\Delta.\mathsf{usedRO} &= \Delta.\mathsf{splittedFrac} \cup \Delta.\mathsf{joinedFrac} \\
\Delta.\mathsf{notRO} &= \mathrm{dom}(\Delta) \setminus \Delta.\mathsf{usedRO}
\end{aligned}
$$

Intersection and Filtering. We define:

$$
\begin{aligned}
\Delta_1 \Cap \Delta_2 &= \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) \\
\Gamma \Vdash \Delta &= \Gamma \Vdash \mathrm{dom}(\Delta)
\end{aligned}
$$

Sequential Composition of Usage Maps. This section defines the *usage composition operator*, written $\Delta_1; \Delta_2$. This operator plays a central role in computing the usage of a sequence of terms. Let us begin with an example.

Consider the sequence "$(\textbf{let}\ r = \mathsf{alloc}(v))^{\Delta_1}; (\textbf{let}\ k = \mathsf{get}(r))^{\Delta_2}; \mathsf{free}(r)^{\Delta_3}$". In Δ_1, we have a binding "y : produced" because the first instruction produces the resource "$y : r \rightsquigarrow \mathsf{Cell}$". In Δ_2, we have a binding "y : splittedFrac" because the instruction only reads with y (thus it accepts any subfraction). In Δ_3, we have a binding "y : uninit" because the third instruction destroys the resource y without caring about the value of the Cell.

Let us give three examples of compositions. First, the usage map $\Delta_1; \Delta_2$ contains a binding "y : produced" because, taken together, the sequential composition of the those two instructions still creates the resource y. Second, the usage map $\Delta_2; \Delta_3$ contains a binding y : full because, taken together, the second and the third instruction consume the Cell, and they read the value that was contained inside. Third, the usage map $\Delta_1; \Delta_2; \Delta_3$ contains no binding for y because the Cell cannot be seen from outside the sequence of instruction.

Formally, the usage composition operation $\Delta_1; \Delta_2$ is defined by merging the two usage maps pointwise by resource name, using the table shown below to compute the "combined usage" in case a same resource name is bound both in Δ_1 and Δ_2. For example, Δ_1 contains a binding y : splittedFrac and Δ_2 contains a binding y : full, then $\Delta_1; \Delta_2$ contains a binding y : full. The $\varnothing$ and $\perp$ entries in the table are explained next.

$\Delta_1; \Delta_2$	$\varnothing$	required	ensured
$\varnothing$	$\varnothing$	required	ensured
required	required	required	$\perp$
ensured	ensured	ensured	$\perp$

$\Delta_1; \Delta_2$	$\varnothing$	full	uninit	splittedFrac	joinedFrac	produced
$\varnothing$	$\varnothing$	full	uninit	splittedFrac	joinedFrac	produced
full	full	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
uninit	uninit	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
splittedFrac	splittedFrac	full	full	splittedFrac	splittedFrac	$\perp$
joinedFrac	joinedFrac	full	uninit	splittedFrac	joinedFrac	$\perp$
produced	produced	$\varnothing$	$\varnothing$	produced	produced	$\perp$

The input or output $\varnothing$ corresponds to cases where there is no binding for a resource name in the usage map. Note that a resource produced in Δ_1 and then fully used in Δ_2 will be absent from $\Delta_1; \Delta_2$. As illustrated in the earlier example, a usage map abstract away intermediate resources not present in the final triple.

The output $\perp$ corresponds to cases that cannot arise. For example, it is not possible to have a linear resource used as full and used again afterwards, since usage full corresponds to a removal from the context. Similarly, the same resource name cannot be produced or ensured twice.

Finally, let us comment on the naming policy. If a linear resource is entirely consumed, its name disappears. If a resource $y : \beta H$ is split as αH and $(\beta - \alpha)H$, the $(\beta - \alpha)H$ part keeps the initial resource name y (and αH takes a fresh resource name). If CloseFracs merges the fractions $y : (\beta - \alpha)H$ and $y' : \alpha H$, it produces a resource βH with the name y (and the name y' disappears).

Let us illustrate how these rules play out on a concrete example. Assume a term t_1 uses a full resource named y to only perform a read operation, and subsequently a term t_2 uses the same resource to perform a write operation. Then, thanks to the fact that the name y was preserved during the carve-out and subsequent CloseFracs operation, the usage map of the sequence $t_1; t_2$ contains, as one would naturally expect, the binding y : full.

5.3 Computing Usage Maps

Usage of a context subtraction. Each time a typing rule performs a subtraction, we add entries to the usage map of the term invoking this rule. This paragraph explains the usage map associated with a subtraction. The usage map of a subtraction $(\sigma, F) = \Gamma_1 \boxminus \Gamma_2$ contains:

- One entry required for each pure variable of Γ_1 mentioned in σ.
- One entry uninit or full for each linear resource of Γ_1 that was unified with a resource of Γ_2. The entry is uninit if the resource in Γ_2 is of the form $\mathsf{Uninit}(H)$. Otherwise, it is a full.

For a subtraction performing read-only carving $\Gamma_1 \ominus \Gamma_2$, the usage map is defined in the same way as $\Gamma_1 \boxminus \Gamma_2$ except that if a linear resource from Γ_2 is found by carving a resource of Γ_1, the entry for that resource from Γ_1 has kind splittedFrac, and we also add an ensured entry for each newly generated fraction.

Usage of a CloseFracs. When closing fractions, we need to add entries to the usage map to account for the modifications on the context. We try to do so in a way that preserves as much information as possible.

When CloseFracs finds a possible reduction on two resources $y_1 : (\alpha - \beta_1 - \ldots - \beta_n)H$ and $y_2 : (\beta_i - \gamma_1 - \ldots - \gamma_m)H$ it keeps the name y_1 from the carved part for the generated closed resource $(\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H$. On the one hand, the resource y_2 disappears from the context (and it is not an uninitialized usage). Therefore, we have to put the usage $t_2 : \mathsf{full}$ in the usage map. On the other hand, the resource y_1 remains in the context. Since the absence of y_1 would not have blocked the typechecking, it gets the usage $t_1 : \mathsf{joinedFrac}$. Note this is currently the only way joinedFrac usage are generated. Note also that the order of reduction does not matter for the final usage map (all the fractions that disappear will have a usage full, and all the fractions that got bigger will have a usage joinedFrac).

Computing Usage During Term Typing. In order to produce triples of the form $\{\Gamma\}\ t\ \{\Gamma'\}$, we need to patch our typing rules to record the usage information.

Here is the full version of the rules Val and Let described earlier:

$$\frac{\Gamma.\mathsf{pure} \vdash v : \tau \qquad \Delta = \{\mathbf{res} : \mathsf{ensured}\}}{\{\Gamma\}\ v^{\Delta}\ \{\Gamma \circledast [\mathbf{res} : \tau := v]\}}\ \textsc{Val}$$

$$\frac{\{\Gamma_0\}\ t^{\Delta}\ \{\Gamma_1\} \qquad \Gamma_2 = \mathrm{Rename}\{\mathbf{res} := x\}(\Gamma_1) \qquad \Delta' = \mathrm{Rename}\{\mathbf{res} := x\}(\Delta)}{\{\Gamma_0\}\ (\mathbf{let}\ x = t)^{\Delta'}\ \{\Gamma_2\}}\ \textsc{Let}$$

For the rule Val, the usage map contains a single binding $\mathbf{res} : \mathsf{ensured}$ to account for the alias added in the context. For the rule Let, the typechecker uses the operator $\mathrm{Rename}\{x := x'\}(\Delta)$, that renames the key x by x' in Δ. This renaming is applied on the usage map of the body to follow the renaming in the context.

Since it would be too verbose to explicitly put usage map in each typing rule, the rest of this section will explain how these usage maps are computed in practice without rewriting the rules. We reuse the variables names of the rules described in figure 18.

- For the rule Seq, if each instruction t_i has a usage map Δ_i, we build Δ'_i to follow renamings in Γ_r.

$$\Delta'_i = \begin{cases} \Delta_i & \text{if } t_i \text{ is of the form "}\mathbf{let\ res} := t'_i\text{"} \\ \mathrm{Rename}\{\mathbf{res} := x_i\}(\Delta_i) & \text{otherwise} \end{cases}$$

  Then the usage map of the whole sequence is $(\Delta'_1; \ldots; \Delta'_n)$.
- For the rule Block, if we name Δ_r the usage map of the sequence, and Δ_c the usage map of the subtraction of StackAllocCells, the usage map of the whole block is $(\Delta_r; \Delta_c)$.
- For the rule Fun, if we name Δ_1 the usage map of the function body, Δ_2 the usage map of the subtraction, and S the generated specification hypothesis, then the usage map of the function definition is $((\Delta_1; \Delta_2) \Vdash \Gamma_0) \cup \{\mathbf{res} : \mathsf{ensured}, S : \mathsf{ensured}\}$. Indeed, viewed from outside the only dependencies of the function definition are the pure resources captured from the surrounding context.
- For the rule App, if Δ_σ is a usage map containing an entry required for each x_i and each pure resource from Γ_0 mentioned in σ, Δ_p is the usage map of the subtraction on Γ_0, Δ_q is a usage map containing one produced (resp. ensured) for each linear (resp. pure) resource in Γ_q, and Δ_f the usage map of the CloseFracs operation, the usage map of the application is $(\Delta_\sigma; \Delta_p; \Delta_q; \Delta_f)$.
- For the rule For, only the outer contract instantiation and the required pure variables needed to typecheck the loop body are considered for computing the usage map. If Δ_p is the usage map of the subtraction $\Gamma_0 \ominus \Gamma_p$, Δ_b is the usage of the body of the loop, Δ'_q is the usage of the subtraction on Γ'_q, Γ_q is a usage map containing one produced (resp. ensured) for

each linear (resp. pure) resource in Γ_q, and Γ_r is the usage of the CloseFracs operation, then $(\Delta_p; ((\Delta_b; \Delta'_q) \Vdash \Gamma_0); \Delta_q; \Delta_r)$ is the usage map of the for-loop. Note that the $((\Delta_b; \Delta'_q) \Vdash \Gamma_0)$ part of this usage map correspond to the usage of pure resources from outside the loop in the body of the loop (they all have a required usage kind).
- For the rule If, the exact usage depends on how the oracle finds a join point between the branch, and if this is made in a way that preserves the invariants of usage map. If the oracle is fully opaque, it is always sound yet imprecise to combine the usage map Δ_0 of the condition expression t_0 to another usage map Δ'_0 that gives a full usage to each linear resource in Γ'_0 and a usage map Δ_3 that contains a produced usage for each linear resource of Γ_3. For the usage of pure resources, we name Δ_1 (resp. Δ_2) the required usage from t_1 (resp. t_2). Then we take all the pure facts from Γ_3 that are not in Γ'_0 as ensured in a usage map Δ'_3. This gives $(\Delta_0; \Delta'_0; \Delta_1; \Delta_2; \Delta_3; \Delta'_3)$ as a potential usage map for the if instruction. We leave better usage computation of conditionals for future work.

5.4 Minimization of Triples

The *triple minimization operation* is used for typing function calls with effectful arguments and for minimizing loop contracts produced by transformations. The operation Minimize$(\Gamma, \Gamma', \Delta)$ is defined when its input corresponds to a valid triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$. The output of such operation is a quadruplet $(\hat{E}, \hat{F}, \hat{F}', \bar{F})$ where intuitively:

- $\hat{F}$ is the *minimized linear precondition*: a linear context containing resources from Γ.linear that are needed to typecheck t.
- $\hat{F}'$ is the *minimized linear postcondition*: a linear context produced after typechecking t if we give only $\hat{F}$ as the linear precondition.
- $\bar{F}$ is the *maximal frame*: a linear context of resources from Γ.linear that were useless in the typechecking of t. It means resources in $\bar{F}$ can be framed during the typechecking of t. Since these resources are not touched by t, they must also occur in Γ'.linear.
- $\hat{E}$ is the *generated fraction set*: a set of pure fractions that are created by the Minimize algorithm to give only an arbitrary subfraction of the resource in Γ.linear in $\hat{F}$ when such a fractional resource suffices to typecheck t.

From the perspective of soundness, the quadruplet $(\hat{E}, \hat{F}, \hat{F}', \bar{F})$ must satisfy the following properties:

- $\{\langle \Gamma.\text{pure}, \hat{E} \mid \hat{F}\rangle\}\ t\ \{\langle \Gamma'.\text{pure}, \hat{E} \mid \hat{F}'\rangle\}$
- $\Gamma \Leftrightarrow \langle \Gamma.\text{pure}, \hat{E} \mid \hat{F} \star \bar{F}\rangle$
- $\Gamma' \Leftrightarrow \langle \Gamma'.\text{pure}, \hat{E} \mid \hat{F}' \star \bar{F}\rangle$

From the perspective of completeness, it is desirable for the frame $\bar{F}$ to somehow corresponds to the "maximal" frame. We have not found a way to formally capture the *maximality* aspect of $\bar{F}$—or, symmetrically, the *minimality* of $\hat{F}$. It is not clear to us whether these notions could be defined in the general case. What appear clearly is a definition of Minimize ought to at least satisfy the following properties, which capture a partial form of completeness.

- If t can typecheck without a linear resource H, then H should be put in the frame $\bar{F}$.
- If t can typecheck with only the uninitialized version of H (because, for instance, it starts by overwriting the data accessible through H), then Uninit(H) should be placed in $\hat{F}$.
- If t can typecheck with only an arbitrary subfraction of H (because, for instance, t only reads using H), then a fresh fraction α should be created and placed in $\hat{E}$, the resource αH should be placed in $\hat{F}$, and $(1 - \alpha)H$ should remain in $\bar{F}$.

Our implementation of Minimize, whose behavior is guided by the entries in the usage map Δ computed when typechecking t, satisfy all the aforementioned soundness and partial-completeness properties.

5.5 Typechecking of Order-Irrelevant Subexpressions

We next explain how to leverage the minimization procedure for typechecking functions calls that are not in A-normal form, but possibly include effectful subexpressions. In C, the arguments of a function call may be evaluated in an arbitrary order. Our goal is to ensure that, for well-typed OptiTrust programs, the order of evaluation is irrelevant. To that end, we consider a sufficient condition: that the arguments can be evaluated in parallel. This condition provides additional flexibility for optimizations.

There do exist valid C programs that fail to typecheck in OptiTrust because our condition is slightly too restrictive. However, such programs may be easily rewritten by binding variables to arguments before the function call. We believe that such a situation would be relatively rare in idiomatic C code.

The rule Subexpr reduces the typechecking of a term with possibly effectful subexpressions to the typechecking of a term whose subexpressions are program variables. In particular, the rule may be used to compute the output context associated with a call of the form $f(e_1, ..., e_n)$ in an input context Γ_0, by reducing the problem to the typechecking of a call of the form $f(x_1, ..., x_n)$, in an input context Γ_p that binds the fresh variables x_i.

The rule Subexpr, shown below, applies to a term of the form $e[t_0, ..., t_n]$, where e denotes a multi-evaluation context and where the t_i variables denote the subterms in evaluation position. The goal of the rule is to distribute the linear resources from the input context Γ_0 between the subterms t_i. If several subterms read the same resource, then this resource needs to be split. If one subterm reads a resource and another subterm modifies that same resource, the rule must fail to apply. The key idea is to typecheck the subterms one after the other, taking advantage of the Minimize operation to remove the minimal amount of resources from the input context, thereby leaving as many resources as possible for the remaining subterms.

$$\frac{\begin{array}{c}\forall i \in [0,n].\quad \{\Gamma_i\}\ t_i^{\Delta_i}\ \{\Gamma_i'\} \quad \wedge \quad (\hat{E}_i, \hat{F}_i, \hat{F}_i', \bar{F}_i) = \mathsf{Minimize}(\Gamma_i, \Gamma_i', \Delta_i) \quad \wedge \quad x_i \text{ fresh} \\ \forall i \in [0,n].\quad \Gamma_{i+1} = \langle\, \Gamma_i.\mathsf{pure}, \hat{E}_i \mid \bar{F}_i \,\rangle \quad \wedge \quad \hat{\Gamma}_i' = \langle\, \Gamma_i'.\mathsf{pure} \Vdash \Delta_i.\mathsf{ensured} \mid \hat{F}_i' \,\rangle \\ \Gamma_p = \mathsf{CloseFracs}^{\Delta_p}(\Gamma_{n+1} \circledast \bigstar_{i\in[0,n]} \mathsf{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}_i')) \\ \{\Gamma_p\}\ e[x_0, ..., x_n]^{\Delta_q}\ \{\Gamma_q\} \\ \Delta = \mathsf{Rename}\{\mathbf{res} := x_0\}(\Delta_0); ...; \mathsf{Rename}\{\mathbf{res} := x_n\}(\Delta_n); \Delta_p; \Delta_q \end{array}}{\{\Gamma_0\}\ e[t_0, ..., t_n]^{\Delta}\ \{\Gamma_q\}}\ \textsc{Subexpr}_\ell$$

Example application of the rule. Figure 19 shows the application of the Subexpr on an example. The columns describe the 4 steps corresponding to the 4 subterms. The rows explain how the metavariables from the rule Subexpr are instantiated. In particular, observe how the two subexpressions get(p) both have read-only access to the same resource H. As the details in the Figure show, the first get(p), according to its minimized precondition, only needs a fraction of H. This fraction is carved out, obtaining a subfraction αH and leaving $(1 - \alpha)H$ for the second get(p).

6 JUSTIFYING TRANSFORMATION CORRECTNESS

In this section, we explain how OptiTrust leverages resource typing information to check the correctness of the transformations requested by the programmer. The aim of this section is not to cover all the transformations implemented in OptiTrust, but to present a representative subset thereof. We focus in particular on transformations that leverage the resource information in an

i	0	1	2	3
Γ_i.pure	$p, q, c : \mathsf{ptr}(\mathsf{int})$	$p, q, c : \mathsf{ptr}(\mathsf{int})$	$p, q, c : \mathsf{ptr}(\mathsf{int})$	$p, q, c : \mathsf{ptr}(\mathsf{int})$ $\alpha : \mathsf{frac}$
Γ_i.linear	$Hp : p \leadsto \mathsf{Cell}$ $Hq : q \leadsto \mathsf{Cell}$ $Hc : c \leadsto \mathsf{Cell}$	$Hp : p \leadsto \mathsf{Cell}$ $Hq : q \leadsto \mathsf{Cell}$ $Hc : c \leadsto \mathsf{Cell}$	$Hp : p \leadsto \mathsf{Cell}$ $Hq : q \leadsto \mathsf{Cell}$	$Hp : (1-\alpha)(p \leadsto \mathsf{Cell})$ $Hq : q \leadsto \mathsf{Cell}$
t_i	q	$\mathsf{get\_incr}(c)$	$\mathsf{get}(p)$	$\mathsf{get}(p)$
Δ_i	q : required **res** : ensured	c : required Hc : full Hc' : produced **res** : ensured	p : required Hp : splittedFrac **res** : ensured	p : required Hp' : splittedFrac **res** : ensured
$\hat{E}_i$	$\varnothing$	$\varnothing$	$\alpha : \mathsf{frac}$	$\beta : \mathsf{frac}$
$\hat{F}_i$	$\varnothing$	$Hc : c \leadsto \mathsf{Cell}$	$\alpha(p \leadsto \mathsf{Cell})$	$\beta(p \leadsto \mathsf{Cell})$
$\hat{F}'_i$	$\varnothing$	$Hc' : c \leadsto \mathsf{Cell}$	$\alpha(p \leadsto \mathsf{Cell})$	$\beta(p \leadsto \mathsf{Cell})$
$\bar{F}_i$	$Hp : p \leadsto \mathsf{Cell}$ $Hq : q \leadsto \mathsf{Cell}$ $Hc : c \leadsto \mathsf{Cell}$	$Hp : p \leadsto \mathsf{Cell}$ $Hq : q \leadsto \mathsf{Cell}$	$Hp : (1-\alpha)(p \leadsto \mathsf{Cell})$ $Hq : q \leadsto \mathsf{Cell}$	$Hp :$ $(1-\alpha-\beta)(p \leadsto \mathsf{Cell})$ $Hq : q \leadsto \mathsf{Cell}$
$\hat{\Gamma}'_i$.pure	**res** $:= q : \mathsf{ptr}(\mathsf{int})$	**res** : int	**res** : int	**res** : int
Γ_p.pure	$p, q, c : \mathsf{ptr}(\mathsf{int}),\ x_0 := q : \mathsf{ptr}(\mathsf{int}),\ x_1, x_2, x_3 : \mathsf{int}$			
Γ_p.linear	$Hp : p \leadsto \mathsf{Cell},\ Hq : q \leadsto \mathsf{Cell},\ Hc' : c \leadsto \mathsf{Cell}$			

Fig. 19. Example of an application of the Subexpr rule on an expression $e[q, \mathsf{get\_incr(c)}, \mathsf{get}(p), \mathsf{get}(p)]$, in a context $\langle p, q, c : \mathsf{ptr}(\mathsf{int}) \mid Hp : p \leadsto \mathsf{Cell}, Hq : q \leadsto \mathsf{Cell}, Hc : c \leadsto \mathsf{Cell}\rangle$.

interesting way. All the transformations presented in this section are invoked multiple times in our case studies from Section 2.

Recall that we only need to check the correctness of *basic* transformations, because *combined* transformations are defined as composition of basic transformations. For every basic transformation, we present a generally applicable, *sufficient condition* for the transformation to be correct.

On the one hand, a number of *basic* transformation might seem "trivial" to the reader. This simplicity is precisely a strength of OptiTrust. As explained in the introduction, we aim to minimize the trusted code base, by considering the simplest possible *basic* transformations and by implementing as many transformations as possible as composition of *basic* transformations.

On the other hand, certain loop transformations might seem "complex" to the reader. We believe this complexity is inherent to advanced loop transformations. In fact, there are even situations where the sufficient conditions that we have considered are *simplified* conditions, that we could further generalize in future work.

Before presenting the key aspects of specific transformations, we introduce notation for describing transformations. Transformations apply to instructions or group of instructions; they depend on context and usage information; and they produce code with possibly updated loop contracts, and possibly including ghost instructions. Hence, we need a convenient way to visualize all these entities.

Notation for Well-Typed Programs. Transformations leverage typing information, not only for checking correctness, but also for guiding the generation of the output code. Recall from the previous section that our typechecking algorithm computes, for every subterm t, its input context Γ_1, its output context Γ_2, and its usage maps Δ, establishing triples of the form $\{\Gamma_1\}\ t^{\Delta}\ \{\Gamma_2\}$. In this section, we use an alternative syntax, better-suited for describing the input of transformations. If t denotes an instruction, we write $\boxed{\Gamma_1\ t; \Delta\ \Gamma_2}$ as straight-line syntax for $\{\Gamma_1\}\ t^{\Delta}\ \{\Gamma_2\}$.

Groups of Instructions. Certain transformations operate on groups of consecutive instructions. We let the meta-variable T range over a (possibly empty) group of instructions. We generalize our alternative syntax by writing $\boxed{\Gamma_1\ T;\Delta\ \Gamma_2}$, where Γ_1 denotes the initial set of resource, Γ_2 denotes the final set of resources, and Δ denotes the *composition* of the usage of the instructions from the group, as defined in Section 5.3. Formally, if Δ_i denotes the usage of the term t_i, then:

$$\boxed{\Gamma_0\ T;\Delta\ \Gamma_n} \quad \equiv \quad \boxed{\Gamma_0\ t_1;\Delta_1\Gamma_1\ t_2;\Delta_2\Gamma_2\ ...t_n;\Delta_n\ \Gamma_n} \qquad \begin{array}{ll} \text{where} & T \equiv t_1;t_2;...;t_n \\ \text{and} & \Delta \equiv \Delta_1;\Delta_2;...;\Delta_n \end{array}$$

Program Contexts. Transformations generally apply to a program subterm, that is, apply under a *program context*. We let the meta-variable $\mathcal{E}$ range over such contexts. For example, evaluating a subexpression $1+1$ that appears in a context $\mathcal{E}$ is described as the transition from $\mathcal{E}[1+1]$ to $\mathcal{E}[2]$. We also allow program contexts to denote a hole in the middle of a sequence. For example, swapping two instructions that appear inside a sequence is described as the transition from $\mathcal{E}[t_1;t_2]$ to $\mathcal{E}[t_2;t_1]$, to be interpreted as a transition from $\mathcal{E}'[\{T_0;t_1;t_2;T_3\}]$ to $\mathcal{E}'[\{T_0;t_2;t_1;T_3\}]$, where $\mathcal{E}'$ denotes the program context associated with the outer sequence that contains $t_1;t_2$. For the first few transformations, we will mention a context $\mathcal{E}$ explicitly, however afterwards we will leave it implicit.

Simple Program Contexts. Certain transformations operate on subexpressions that appear inside an instruction. For those, we may need to restrict the form of the contexts in which the subexpression may appear (e.g., to avoid nontrivial control-flow arising from the && operator or from the C ternary operator). We therefore introduce the notion of *simple program context*, written $\hat{\mathcal{E}}$, to denote a program context that solely consists of possibly-nested functions calls. An example simple context is $f(g(\square,2),g(3,a+4))$, where $\square$ denotes the hole. One key property for any simple context $\hat{\mathcal{E}}$ is that the following rewrite is correct: $\boxed{\hat{\mathcal{E}}[e]w} \longmapsto \boxed{\textbf{let}\ x=e;\hat{\mathcal{E}}[x]}$.

Note that the validity of this binding rule, and more generally the interest of simple contexts, crucially relies on the hypothesis that the input code typechecks against our typing rules. Indeed, our rules ensure that, if a function has multiple arguments, then the available resources are distributed across the arguments—only read-only resources can be distributed onto several arguments. For example, $f(g_1(),g_2(),g_3())$ is equivalent to $\textbf{let}\ x = g_2();f(g_1(),x,g_3())$ because, if the former term is well-typed, then the effects of $g_2()$ do commute with the effects of $g_1()$ and $g_3()$.

Notation for Introducing Ghost Calls. Recall that a call to a ghost function is an instruction that semantically behaves as a no-op, yet updates the set of resources available. In the output of transformations, we write $\textbf{ghost}(\Gamma \longrightarrow \Gamma')$ to mean the insertion of an appropriate ghost function call $g()$, such that g admits Γ as precondition and Γ' as postcondition. Concretely, the effect of $\textbf{ghost}(\Gamma \longrightarrow \Gamma')$ is to consume the resources Γ then to produce the resources Γ'.

We are now ready to present transformations. We begin with transformations on instructions and variable bindings, then move on to transformations on storage, and transformations on loops.

6.1 Transformations on Sequences of Instructions

Moving Instructions. The basic transformation `Instr.move` allows to move a group of instructions to a given destination within the same sequence. Doing so amounts to swapping a group of instructions T_1 with an adjacent group of instructions T_2. The *move* transformation turns a program of the form $\mathcal{E}[T_1;T_2]$ into $\mathcal{E}[T_2;T_1]$, where $\mathcal{E}$ denotes a context. The transformation is formalized as shown below. The variables Δ_1 and Δ_2 denote the usage associated with T_1 and T_2. The correctness criteria, stated on the right-hand-side, is explained next.

$$\boxed{\mathcal{E}\,[T_1;\Delta_1;\,T_2;\Delta_2]} \;\longmapsto\; \boxed{\mathcal{E}\,[T_2;\,T_1]} \qquad \text{correct if: } \begin{cases} \Delta_1.\mathsf{notRO} \;\dot\cap\; \Delta_2 = \varnothing \\ \Delta_2.\mathsf{notRO} \;\dot\cap\; \Delta_1 = \varnothing \end{cases}$$

The expression $\Delta_1.\mathsf{notRO}$ essentially denotes the resources that T_1 modifies. (Technically, this expression denotes the resources that T_1 does not *only* read.) The property $\Delta_1.\mathsf{notRO} \dot\cap \Delta_2 = \varnothing$ captures the idea that if a resource is modified by T_1, then T_2 must not use it, otherwise swapping T_1 and T_2 might not be correct. (The resource intersection operator $\dot\cap$ was defined in Section 5.2.) The second property, namely $\Delta_2.\mathsf{notRO} \dot\cap \Delta_1 = \varnothing$, captures the symmetrical property: if a resource is modified by T_2, then T_1 must not use it. When both conditions are met, the only resources that both T_1 and T_2 depend on are accessed in read-only mode. In such a situation, the groups of instructions T_1 and T_2 may be safely swapped, without impact on the result of their evaluation.

Deleting Instructions. The basic transformation `Instr.delete` allows deleting a group of instructions T from a sequence. It therefore maps a program $\mathcal{E}'[\{T_0;T;T_2\}]$ to a program $\mathcal{E}'[\{T_0;T_2\}]$, for a context $\mathcal{E}'$. Following our convention that program contexts may describe subsequences, we may also describe the transformation as mapping $\mathcal{E}[T]$ to $\mathcal{E}[\varnothing]$, for a context $\mathcal{E}$.

Intuitively, the deletion operation preserves the semantics of the program if the contents of the resources modified by T are not observed by the rest of the program. If T has been typechecked as "$\Gamma\ T;\Delta$", then the set of resources modified by T, written Γ_m, is computed by the filtering operation $\Gamma \mathbin{|\!\cdot} \Delta.\mathsf{notRO}$. (Filtering was defined in Section 5.2.) To test the hypothesis that Γ_m is not observed by the rest of the program, we build an auxiliary program, written $\mathcal{E}[G]$, in which the instructions T are replaced with a ghost instruction G that casts the resources from Γ_m into their corresponding "uninitialized form", written $\mathsf{IntoUninit}(\Gamma_m)$. In other words, if a resource H is modified by T, then the ghost operation G consumes H and produces $\mathsf{Uninit}(H)$.

The transformation can therefore be formalized as follows.

$$\boxed{\mathcal{E}\,[\Gamma\ T;\Delta]} \;\longmapsto\; \boxed{\mathcal{E}\,[\varnothing]} \qquad \begin{array}{l}\text{correct if } \mathcal{E}\,[\mathbf{ghost}(\Gamma_m \longrightarrow \mathsf{IntoUninit}(\Gamma_m))] \text{ typechecks,} \\ \text{where } \Gamma_m \equiv \Gamma \mathbin{|\!\cdot} \Delta.\mathsf{notRO}.\end{array}$$

If the auxiliary program $\mathcal{E}[G]$ typechecks, then we can discard this program, and safely replace the original program $\mathcal{E}[T]$ with $\mathcal{E}[\varnothing]$. Note that this pattern of introducing an auxiliary program for the purpose of evaluating a correctness criteria will appear again for other transformations.

Inserting Instructions. The transformation `Instr.insert` refines a program from $\mathcal{E}[\varnothing]$ to $\mathcal{E}[T]$, where T denotes the group of inserted instructions. The correctness criteria, described below, is essentially the same as that for instruction deletion. Indeed, for $\mathcal{E}[T]$ to admit the same semantics as $\mathcal{E}[\varnothing]$, it suffices that $\mathcal{E}[\varnothing]$ admits the same semantics as $\mathcal{E}[T]$.

$$\boxed{\mathcal{E}\,[\varnothing]} \;\longmapsto\; \boxed{\mathcal{E}\,[T]} \qquad \begin{array}{l}\text{correct if:} \\ \text{(1) the program } \mathcal{E}[T] \text{ typechecks as } \mathcal{E}[\Gamma\ T;\Delta] \text{ for some } \Gamma \text{ and } \Delta; \\ \text{(2) the program } \mathcal{E}\,[\mathbf{ghost}(\Gamma_m \longrightarrow \mathsf{IntoUninit}(\Gamma_m))] \text{ type-} \\ \text{checks, where } \Gamma_m \equiv \Gamma \mathbin{|\!\cdot} \Delta.\mathsf{notRO}, \text{ for the above values of } \Gamma \text{ and} \\ \Delta.\end{array}$$

Idempotent Terms. A number of transformations depend on the notion of idempotence. In the C23 standard, an expression is said to be "idempotent" if, intuitively, evaluating this expression multiple times in immediate sequence produces the same results. In OptiTrust, we leverage our resource analysis to capture a practical over-approximation of idempotence.[11] A term can be considered

[11] The C23 standard defines a number of related notions. In particular, an expression is said to be "effectless" iff "if any store operation that is sequenced during the execution is the modification of an object that synchronizes with the call". An expression is said to be "reproducible" iff it is both effectless and idempotent. Reproducibility is essentially equivalent to the notion of pure expression in GCC's terminology [ale 2022]. Due to our resource typing discipline, all OptiTrust terms can be considered "effectless". Hence, in the context of OptiTrust, "idempotent" and "reproducible" are equivalent.

idempotent if the resources that this term produces correspond: (1) either to uninitialized resources that were consumed by this term; or (2) to read-only resources that the term consumes and returns with the exact same fraction. These criteria may be formalized as follows.

$$\text{A term } T \text{ that appears in a program } \mathcal{E}\,[\Gamma_1\ T;\ \Delta\ \Gamma_2] \text{ is } \textit{idempotent} \text{ iff: } \begin{cases} \Delta.\text{full} = \varnothing \\ (\Gamma_2 \Vdash \Delta.\text{produced}) \boxminus (\Gamma_1 \Vdash \Delta.\text{uninit}) = (\sigma, \varnothing) \\ \hfill \text{for some } \sigma \end{cases}$$

In particular, these criteria rule out terms that consume full resources, or produce resources they did not consume. For example, `x = y+1`, which reads `y` and assigns `x` is idempotent; however `x++`, which modifies `x`, is *not* idempotent. One key property that holds for an idempotent expression e is that the following program equivalence holds: $\boxed{\textbf{let}\ x = e; \hat{\mathcal{E}}[e]} \leftrightarrow \boxed{\textbf{let}\ x = e; \hat{\mathcal{E}}[x]}$.

Duplicating and Deduplicating Instructions. If an instruction T (or possibly a group of instructions) is idempotent, then after a first instruction T, a second instruction T may be inserted or removed without affecting the semantics. The transformation `Instr.dup` and its reciprocal `Instr.dedup` are formalized, for the general case of groups of instructions, as follows.

$$\boxed{\mathcal{E}[T]} \quad \leftrightarrow \quad \boxed{\mathcal{E}[T;T]} \qquad \text{where } T \text{ is idempotent.}$$

Similarly, if an expression e is idempotent, then after the instruction $\textbf{let}\ x = e$, an instruction $\textbf{let}\ y = e$ may be inserted or removed, for a fresh variable y. The corresponding transformations are named `Instr.dup_let` and `Instr.dedup_let`. Thereafter, for brevity, we omit the context surrounding the code snippet, previously written $\mathcal{E}$.

$$\boxed{\begin{array}{l}\textbf{let}\ x = e; \\ \phantom{\textbf{let}\ y = e;}\end{array}} \quad \leftrightarrow \quad \boxed{\begin{array}{l}\textbf{let}\ x = e; \\ \textbf{let}\ y = e;\end{array}} \qquad \text{where } e \text{ is idempotent.}$$

Deduplicating expressions is a building block for common subexpression elimination, which is detailed further on. Duplicating expressions can also improve performance in certain situations: recomputing a simple expression may be cheaper than storing its value in memory and subsequently retrieving this value, especially if the redundant computations are scattered in distinct loops.

6.2 Exploiting Equalities

Read after Write. The transformation `Eq.read_after_write` captures the fact that reading immediately after a write yields the value that was written. On its own, this transformation may seem of little interest; however, it is useful when combined with moves of the read or the write instruction.

$$\boxed{\begin{array}{l}\text{set}(p, v); \\ \hat{\mathcal{E}}[\text{get}(p)];\end{array}} \quad \longmapsto \quad \boxed{\begin{array}{l}\text{set}(p, v); \\ \hat{\mathcal{E}}[v];\end{array}} \qquad \begin{array}{l}\text{correct if } \hat{\mathcal{E}} \text{ is a simple context and } v \text{ is} \\ \text{a pure expression.}\end{array}$$

Results of Idempotent Expressions. The transformation `Eq.idempotent` captures the fact that evaluating an idempotent expression twice yields equal results.

$$\boxed{\begin{array}{l}\textbf{let}\ x = e; \\ \textbf{let}\ y = e; \\ \mathcal{E}[x]\end{array}} \quad \longmapsto \quad \boxed{\begin{array}{l}\textbf{let}\ x = e; \\ \textbf{let}\ y = e; \\ \mathcal{E}[y]\end{array}} \qquad \text{where } e \text{ is idempotent.}$$

6.3 Transformations on Bindings

Inlining/Binding for Pure Expressions. The basic transformation `Variable.inline_pure` eliminates a binding of the form **let** $x = v$, where v is a pure expression, by replacing all occurrences of x with v. This transformation is always correct and requires no check. The reciprocal transformation, `Variable.bind_pure`, introduces a binding for one or several occurrences of a pure expression v. Likewise, it is always correct.

Inlining a Binding with a Single Occurrence, in the Next Instruction. The basic transformation `Variable.inline_one` eliminates a binding **let** $x = e$ in programs where x has exactly one occurrence, and this occurrence is contained in the immediately succeeding instruction, under a simple context $\hat{\mathcal{E}}$. As mentioned earlier, the correctness of this inlining transformation critically relies on the fact that our typing rules ensure that the order of evaluation of subexpressions is irrelevant.

$$\boxed{\begin{array}{l}\textbf{let } x = e;\\ \hat{\mathcal{E}}[x];\end{array}} \longmapsto \boxed{\hat{\mathcal{E}}[e];}$$

correct if $\hat{\mathcal{E}}$ is a simple context, with no other occurrence of x than the one in its hole, and the output program typechecks.

Inlining a Binding with Multiple Occurrences, in the Next Instruction. The transformation `Variable.inline_dup` expands a binding at one of its occurrences, without removing the binding. Here again, we consider an occurrence appearing in an immediately succeeding simple context. This transformation is implemented as a *combined* transformation, decomposed as shown below. Recall that we do not need to devise correctness criteria for combined transformations.

$$\boxed{\begin{array}{l}\textbf{let } x = e;\\ \hat{\mathcal{E}}[x];\end{array}} \xmapsto[(\texttt{Instr.dup\_let})]{} \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \hat{\mathcal{E}}[x];\end{array}} \xmapsto[(\texttt{Eq.idempotent})]{} \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \hat{\mathcal{E}}[y];\end{array}} \xmapsto[(\texttt{Variable.inline\_one})]{} \boxed{\begin{array}{l}\textbf{let } x = e;\\ \hat{\mathcal{E}}[e];\end{array}}$$

Inlining a Binding in the Scope of a Sequence. The combined transformation `Variable.inline` eliminates a binding **let** $x = e$ in the general case. If e is a pure expression, then `Variable.inline_pure` is invoked. Otherwise, we implement the inlining as a combination of several of the aforementioned transformations. Indirectly, our combined transformation enforces the minimal checks required for eliminating a binding **let** $x = e$ without affecting the semantics.

- If x has no occurrences, the effects of e need to be irrelevant to the rest of the program.
- If x has exactly one occurrence, then the effects of e needs to commute with all the instructions located between the binding on x and the occurrence of x.
- If x has several occurrences, then, in addition to the requirement from the previous case, e moreover needs to be idempotent.

Concretely, our transformation proceeds as follows. If there are no occurrences of x, it invokes the transformation `Instr.delete`. If there is exactly one occurrence of x, it attempts to move, using `Instr.swap`, the binding on x just in front of this binding, then invoke `Variable.inline_one`. If there are several occurrences of x in the sequence, then it moves the binding to the front of the first instruction that contains occurrences of x; then it applies the transformation `Variable.inline_dup`; then it repeats the process until reaching the last occurrence of x. We show below an example decomposition of `Variable.inline`, where e is assumed to be idempotent.

$$\begin{array}{lll} & \textbf{let}\ x = e;\ g();\ \mathrm{set}(a, x);\ \mathrm{set}(b, x); & \\ \longmapsto & g();\ \textbf{let}\ x = e;\ \mathrm{set}(a, x);\ \mathrm{set}(b, x); & (\texttt{Instr.swap}) \\ \longmapsto & g();\ \textbf{let}\ x = e;\ \mathrm{set}(a, e);\ \mathrm{set}(b, x); & (\texttt{Variable.inline\_dup}) \\ \longmapsto & g();\ \mathrm{set}(a, e);\ \textbf{let}\ x = e;\ \mathrm{set}(b, x); & (\texttt{Instr.swap}) \\ \longmapsto & g();\ \mathrm{set}(a, e);\ \mathrm{set}(b, e); & (\texttt{Variable.inline\_one}) \end{array}$$

We leave to future work the support, in a combined transformation, of more complex patterns where occurrences of a non-pure binding appear in depth under control flow constructs.

Binding Introduction. The basic transformation `Variable.bind_one` is essentially the reciprocal of `Variable.inline_one`. Given an instruction of the form $\hat{\mathcal{E}}[e]$, the transformation introduces a binding **let** $x = e$ and turns the instruction into $\hat{\mathcal{E}}[x]$.

$$\boxed{\hat{\mathcal{E}}[e];} \quad \longmapsto \quad \boxed{\begin{array}{l}\textbf{let}\ x = e;\\ \hat{\mathcal{E}}[x];\end{array}} \quad \text{correct if } \hat{\mathcal{E}} \text{ is a simple context.}$$

Folding for Additional Occurrences. The combined transformation `Variable.bind_dup` is essentially the reciprocal of `Variable.inline_dup`. In the scope of a binding **let** $x = e$, this transformation turns $\hat{\mathcal{E}}[e]$ into $\hat{\mathcal{E}}[x]$.

Common Subexpression Elimination. The combined transformation `Variable.bind` is essentially the reciprocal of `Variable.inline`. Internally, it exploits the transformations `Variable.bind_one` and `Variable.bind_dup` to introduce a binding that factorizes the evaluation of common subexpressions. For example, if e is idempotent and commutes with $g()$, the program "$g();\ \mathrm{set}(a, e);\ \mathrm{set}(b, e)$" can be transformed into "$\textbf{let}\ x = e;\ g();\ \mathrm{set}(a, x);\ \mathrm{set}(b, x)$".

6.4 Transformations on Storage

The purpose of this section is to present transformations for introducing, eliminating, and converting between various forms of storage. We present transformations operating on single cells, and omit from the discussion the generalizations to arrays and N-dimensional matrices.

Recall from Section 3 that a pure program variable `const int x = 3` is represented in the OptiTrust AST as **let** $x = 3$, that a non-pure stack-allocated variable `int x = 3` is represented as **let** $x = \mathrm{ref}(3)$, and that an uninitialized variable `int x` is represented as **let** $x = \mathrm{ref\_uninit}()$. For stack-allocated data, the resources produced by ref are automatically reclaimed at the end of the scope. For heap-allocated data, the resources produced by alloc are consumed by the matching call to free.

Separating Declaration from Initialization. For a stack-allocated variable, the basic transformation `Variable.init_detach` separates its declaration from its initialization. This transformation is useful as a preliminary step for the combined transformation that hoists a variable declaration appearing inside a loop into an array allocated outside that loop. The basic transformation `Variable.init_attach` applies the reciprocal operation.

$$\boxed{\textbf{let}\ x = \mathrm{ref}(e);} \quad \leftrightarrow \quad \boxed{\textbf{let}\ x = \mathrm{ref\_uninit}();\mathrm{set}(x, e);}$$

Converting between Stack and Heap Allocation. The basic transformation `Variable.to_heap` transforms an uninitialized stack-allocated storage into a corresponding heap-allocated storage. The transformation takes as optional argument the target at which the free instruction should be inserted; by default, it is placed at the end of the scope. The reciprocal transformation is named `Variable.to_stack`. In the statement below, n denotes the size of the type of x.

$$\boxed{\{T_1;\ \textbf{let}\ x = \mathsf{ref\_uninit}();\ T_2;\}} \quad \leftrightarrow \quad \boxed{\{T_1;\ \textbf{let}\ x = \mathsf{alloc}(n);\ T_2;\ \mathsf{free}(x);\}}$$

Removal of Unused Storage. If a stack-allocated storage is never used, it may be removed by means of the operation `Instr.delete`. Concretely, the instruction **let** x = ref_uninit() may be deleted if x has no occurrences, and the instruction **let** x = ref(e) may be deleted if moreover the effects performed by e are not observed by the rest of the program.

If a heap-allocated space is never used, then it may also be removed. To that end, one needs to delete both the alloc and the corresponding free instructions. Neither of them can be removed independently, because both depend on each other. However, if we move using `Instr.move` the alloc instruction next to the free instruction, or vice-versa, then the group made of the two instructions may be removed at once by means of `Instr.delete`. The combined transformation `Variable.delete`, described below, performs this task.

$$\boxed{\begin{array}{l}\textbf{let}\ x = \mathsf{alloc}(e);\\ T;\\ \mathsf{free}(x);\end{array}} \ \xrightarrow[(\texttt{Instr.move})]{} \ \boxed{\begin{array}{l}\textbf{let}\ x = \mathsf{alloc}(e);\\ \mathsf{free}(x);\\ T;\end{array}} \ \xrightarrow[(\texttt{Instr.delete})]{} \ \boxed{T}$$

Temporary Alternative Storage. The transformation `Variable.local_name` is the most complex that we have implemented in terms of operations on plain sequences of instructions. The transformation `Variable.local_name` operates over a specified group of instructions, say T, for a specified storage, say x. Over this scope, a fresh storage, call it y, is allocated. Just before executing T, the contents of x are copied into y. All instructions from T are updated to use y instead of x. Just after these instructions, the possibly-updated contents of y is copied into x. Depending on the situation, the initial copy from x to y, or the final copy from y into x might be unnecessary—and even ill-typed. Such unnecessary copy operations are omitted.

The variable x may be allocated either on the stack or on the heap. The user may choose to allocate y on the stack or on the heap. Moreover, our implementation supports the general case where x is not just a variable but an N-dimensional matrix. In case where x is a matrix, y may correspond to only a subset (i.e., a tile) of the matrix. The interest of the `local_name` transformation is to enable the program to operate on a local piece of data. Crucially, the memory layout of this data may be refined by subsequent transformations, for example to store the transposed of a matrix in a cache friendly way (as in Section 2.3), or to enable vectorization.

The transformation is described in Figure 20. There, the group of instructions T is represented as $E[x, .., x]$, i.e., as a context with multiple occurrences of x. The context Γ_1 describes the resources available before T, and Γ_2 the resources available after T. This typing information is used not only for checking the correctness criterion, but also for guiding the generation of the output code.

The correctness criterion appear at the bottom of Figure 20. An essential aspect of this criterion is to check that, during the execution of T, the resource R corresponding to the full permission on x is *frozen* (i.e., made unavailable) in order to ensure that no operation may be performed on x via potential aliases of this pointer. The first ghost call uses a standard technique for enforcing such a *freeze* in Separation Logic: introducing a *magic wand* operator ($-\!\!*$), guarded by a token named H in the postcondition of the ghost operation, and bound as H' in the rest of the sequence. The heap predicate H' admits the type Hprop, which is the type of all heap predicates in Separation Logic. This heap predicate H' serves the role of a *key* for unfreezing R at the desired point—here, the end of the scope on which y is used in place of x, where the second ghost call is placed. As far as the

$$\boxed{\begin{array}{l} \Gamma_1 \\ E[x, \ldots, x]; \\ \Gamma_2 \end{array}} \longmapsto \boxed{\begin{array}{l} \textbf{let } y = \mathsf{ref\_uninit}(); \\ T_1; \\ E[y, \ldots, y]; \\ T_2; \end{array}}$$

where E is a multi-hole context with one hole per occurrence of x, and where:

$$\begin{cases} T_1 \equiv \mathsf{set}(y, \mathsf{get}(x)); & \text{if } x \rightsquigarrow \mathsf{Cell} \text{ appears as RO or Full in } \Gamma_1 \\ T_1 \equiv \varnothing & \text{if } x \rightsquigarrow \mathsf{Cell} \text{ appears as Uninit in } \Gamma_1 \end{cases}$$

$$\begin{cases} T_2 \equiv \mathsf{set}(x, \mathsf{get}(y)); & \text{if } x \rightsquigarrow \mathsf{Cell} \text{ appears as Full in } \Gamma_2 \\ T_2 \equiv \varnothing & \text{if } x \rightsquigarrow \mathsf{Cell} \text{ appears as Uninit or RO in } \Gamma_2 \end{cases}$$

correct if the program to the right typechecks successfully, where R is one of:
- $(x \rightsquigarrow \mathsf{Cell})$
- $\alpha(x \rightsquigarrow \mathsf{Cell})$
- $\mathsf{Uninit}(x \rightsquigarrow \mathsf{Cell})$

according to how "$x \rightsquigarrow \mathsf{Cell}$" appears in Γ_1.

$$\boxed{\begin{array}{l} \textbf{let } y = \mathsf{ref\_uninit}(); \\ T_1; \\ \textbf{ghost}(\langle \varnothing \mid R \rangle \longrightarrow \langle H : \mathsf{Hprop} \mid H, (H \twoheadrightarrow R) \rangle); \text{ binding } H \text{ as } H' \\ E[y, \ldots, y]; \\ \textbf{ghost}(\langle \varnothing \mid H', (H' \twoheadrightarrow R) \rangle \longrightarrow \langle \varnothing \mid R \rangle); \\ T_2; \end{array}}$$

Fig. 20. Description of the basic transformation `Variable.local_name`.

present paper is concerned, the magic wand operator can be viewed as a binary operator on heap predicates whose definition needs not be revealed to the user.

6.5 Transformations on Loops

Loop transformations depend on the contracts associated with the loops from the input code. For every loop being modified or introduced, the transformations also need to produce appropriate contracts. In what follows, we present details for loop tiling, loop interchange, loop fission, and loop hoisting. We then list other loop transformations that we have implemented.

Loop Tiling. The basic transformation `Loop.tile` allows tiling (a.k.a. strip-mining) a loop. Concretely, it transforms a loop, say with index i, into two nested loops, with indices j and k. Intuitively, the outer loop on j iterates over the *blocks*, whereas the inner loop on k iterates inside every block. Depending on the form of the input range, and on whether the block size divides the width of the loop range, the transformation is able to generate different ranges for the output loops. For each kind of output, the expression $\mathsf{RecoverIndex}(j, k)$ indicates how to compute the original index i in terms of the two new indices j and k.

The 4 variants supported by `Loop.tile` are described in Figure 21. The ranges of the three loops are written ri, rj and rk, respectively. A range is of the form $\textbf{range}(\mathit{start}, \mathit{stop}, \mathit{step})$. The notation $\mathit{start}..\mathit{stop}$ is a shorthand for $\textbf{range}(\mathit{start}, \mathit{stop}, 1)$. In particular, $0..n$ describes the range of values from 0 inclusive to n exclusive. The contracts for the three loops involved are written χ_i, χ_j and χ_k, respectively. To typecheck the output code, *ghost tiling* operations need to be inserted, materialized before and after the produced loops in the figure. Indeed, the loop on i consumes, in particular, the resource: $\bigstar_{i \in ri}\, \chi_i.\mathsf{excl.pre}$ whereas the loop on j consumes instead: $\bigstar_{j \in rj} \bigstar_{k \in rk}\, \chi_k.\mathsf{excl.pre}$.

$$
\boxed{\begin{array}{l}
\textbf{for}_{\ \chi_i}\ i \in ri\ \{ \\
\quad T; \\
\}
\end{array}}
\longmapsto
\boxed{\begin{array}{l}
\textbf{ghost}(\bigstar_{i \in ri}\ \chi_i.\text{excl.pre} \longrightarrow \bigstar_{j \in rj}\bigstar_{k \in rk}\ \chi_k.\text{excl.pre}); \\
\textbf{for}_{\ \chi_j}\ j \in rj\ \{ \\
\quad \textbf{for}_{\ \chi_k}\ k \in rk\ \{ \\
\quad\quad T'; \\
\quad \} \\
\} \\
\textbf{ghost}(\bigstar_{j \in rj}\bigstar_{k \in rk}\ \chi_k.\text{excl.post} \longrightarrow \bigstar_{i \in ri}\ \chi_i.\text{excl.post});
\end{array}}
$$

where:

$$T' \equiv \mathsf{Subst}\{i := \mathsf{RecoverIndex}(j, k)\}(T)$$

$$\chi_k \equiv \mathsf{Subst}\{i := \mathsf{RecoverIndex}(j, k)\}(\chi_i)$$

$$\chi_j \equiv \begin{cases} \text{vars} \equiv \chi_i.\text{vars} \\ \text{shrd} \equiv \mathsf{Subst}\{i := \mathsf{RecoverIndex}(j, rk.\text{start})\}(\chi_i.\text{shrd}) \\ \text{excl} \equiv \{\text{pre} \equiv \bigstar_{k \in rk}\ \chi_k.\text{excl.pre};\ \text{post} \equiv \bigstar_{k \in rk}\ \chi_k.\text{excl.post}\} \end{cases}$$

with the following possible instantiations for the ranges:

Variant	Range ri	Range rj	Range rk	Formula for recovering i: $\mathsf{RecoverIndex}(j, k)$
A	$0..(m \times b)$	$0..m$	$0..b$	$j * b + k$
B	$0..n$ where b divides n	$0..(n/b)$	$0..b$	$j * m + k$
C	$0..n$ where b divides n	$\textbf{range}(0, n, b)$	$j..j + b$	k
D	$0..n$	$\textbf{range}(0, n, b)$	$j..\min(j + b, n)$	k

Fig. 21. Description of the 4 variants of the basic transformation `Loop.tile`.

Loop Interchange. The basic transformation `Loop.swap` allows interchanging (i.e. swapping) two loops. It is described at the top of Figure 22. There exists a general criterion capturing when two loops may be swapped, however this criterion requires reasoning about the resources required by specific iterations, e.g. $T_{i,j}$ and $T_{i',j'}$ with $i' > i$ and $j > j'$. Instead, we focus on two conditions that are simpler nevertheless sufficient for many practical situations: if at least one of the outer loop or the inner loop is parallelizable, then swapping the two loops is correct. Figure 22 describes the case where the outer loop is parallelizable. The case where the inner loop is parallelizable, not shown, is treated with just a few changes.

The first step is to partition the resources from the inner loop contract depending on where they come from relative to the resources from the outer loop. We name partitions by using the first letter to denote its inner loop origin, and the second letter to denote its outer loop origin. We use I for invariant, R for shared reads, P for exclusive precondition and Q for exclusive postcondition. For example, the inner shared reads are partitioned into RP_i that comes from the outer precondition, and RR that comes from the outer shared reads. Internally, we rely on the subtraction operation for computing the partitions (additional details may be found in Section **??**).

Then, we appropriately place the resources obtained from the partitioning in the contracts χ_i' and χ_j' associated with the swapped loops. Compared with χ_j, the new contract χ_j' essentially adds a $\bigstar_i$ operator to certain components. Compared with χ_i, the new contract χ_i' removes occurrences of the $\bigstar_j$ operators. Note that the loop on i remains parallelizable. Around the new loop nest, a

$$\begin{array}{|l|}\hline \textbf{for}_{\ \chi_i}\ i \in r_i\ \{ \\ \quad \textbf{for}_{\ \chi_j}\ j \in r_j\ \{ \\ \quad\quad T; \\ \quad \} \\ \} \\ \hline \end{array} \longmapsto \begin{array}{|l|}\hline \textbf{ghost}(\bigstar_i \bigstar_j PP_{i,j} \longrightarrow \bigstar_j \bigstar_i PP_{i,j}); \\ \textbf{for}_{\ \chi'_j}\ j \in r_j\ \{ \\ \quad \textbf{for}_{\ \chi'_i}\ i \in r_i\ \{ \\ \quad\quad T; \\ \quad \} \\ \} \\ \textbf{ghost}(\bigstar_j \bigstar_i QQ_{i,j} \longrightarrow \bigstar_i \bigstar_j QQ_{i,j}); \\ \hline \end{array}$$

correct if: parallelizable(χ_i), that is, χ.shrd.inv $= \varnothing$.

The contracts from the input code are decomposed as follows:

$$\begin{aligned} \chi_j.\mathsf{shrd} &= \{\mathsf{inv} = (IP_{i,j} \star IR_j),\ \mathsf{reads} = (RP_i \star RR)\} \\ \chi_j.\mathsf{excl} &= \{\mathsf{pre} = (PP_{i,j} \star PR_j),\ \mathsf{post} = (QQ_{i,j} \star PR_j)\} \\ \chi_i.\mathsf{shrd} &= \{\mathsf{inv} = \varnothing,\ \mathsf{reads} = (\bigstar_j PR_j \star IR_{r_j.\mathsf{start}} \star RR)\} \\ \chi_i.\mathsf{excl} &= \{\mathsf{pre} = (\bigstar_j PP_{i,j} \star IP_{i,r_j.\mathsf{start}} \star RP_i),\ \mathsf{post} = (\bigstar_j QQ_{i,j} \star IP_{i,r_j.end} \star RP_i)\} \end{aligned}$$

The contracts for the output code are built as follows:

$$\chi'_i \equiv \begin{cases} \mathsf{vars} \equiv \chi_j.\mathsf{vars} \\ \mathsf{shrd} \equiv \{\mathsf{inv} \equiv \varnothing,\ \mathsf{reads} \equiv (PR_j \star IR_j \star RR)\} \\ \mathsf{excl} \equiv \{\mathsf{pre} \equiv (PP_{i,j} \star IP_{i,j} \star RP_i),\ \mathsf{post} \equiv (QQ_{i,j} \star IP_{i,r_j.\mathsf{next}(j)} \star RP_i)\} \end{cases}$$

$$\chi'_j \equiv \begin{cases} \mathsf{vars} \equiv \chi_j.\mathsf{vars} \\ \mathsf{shrd} \equiv \{\mathsf{inv} \equiv (\bigstar_i IP_{i,j} \star IR_j),\ \mathsf{reads} \equiv (\bigstar_i RP_i \star RR)\} \\ \mathsf{excl} \equiv \{\mathsf{pre} \equiv (\bigstar_i PP_{i,j} \star PR_j),\ \mathsf{post} \equiv (\bigstar_i QQ_{i,j} \star PR_j)\} \end{cases}$$

Fig. 22. The basic transformation Loop.swap, in the particular case where the outer loop is parallelizable.

pair of ghost operations is inserted for swapping groups of resources—a necessary step to match the resources required by the new loop nest.

Loop Fission. The basic transformation `Loop.fission` breaks a loop with body $T_1; T_2$ into two loops over the same range, a first loop with body T_1, and a second loop with body T_2. The transformation is described in Figure 23. As for loop swapping, there exists a general correctness criteria expressed using inequalities on indices, but for now we focus on a simpler yet practical criteria.

Our criterion asserts that loop fission is correct if the resources modified by T_1 at any iteration i do not interfere with the resources modified by T_2 at any other iteration $i' \neq i$. To implement this check, we inspect the usage maps Δ_1 and Δ_2 associated with T_1 and T_2, respectively. If T_1 modifies one resource from χ.shrd, then T_2 must not use this same resources; symmetrically, if T_2 modifies a resource, then T_1 must not use it. Note, however, that T_1 and T_2 are allowed to both read the same resource; moreover, the resources exclusively modified or produced by T_1 at the i-th iteration of the first loop may be consumed by T_2 at the i-th iteration of the second loop.

There remains to explain how to synthesize the contracts χ_1 and χ_2, associated with the two generated loops, from the original contract χ. For shrd resources, we project the subsets of χ.shrd resources used by T_1 and T_2. For excl resources, we need to synthesize the invariant at the cut point, written R. The first loop takes the exclusive resources from χ.excl.pre to R, whereas the second loop takes the exclusive resources from R to χ.excl.post. At a high level, this set of resources R is computed by subtracting the shared resources as well as the local allocations from T_1, described by

$$\boxed{\begin{array}{l}\textbf{for}_{\ \chi}\ i \in r_i\ \{\\ \quad T_1; \Delta_1\\ \quad \Gamma\\ \quad T_2; \Delta_2\\ \}\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{for}_{\ \chi_1}\ i \in r_i\ \{\\ \quad T_1;\\ \}\\ \textbf{for}_{\ \chi_2}\ i \in r_i\ \{\\ \quad T_2;\\ \}\end{array}} \quad \begin{array}{l}\text{correct if:}\\ \begin{cases} i \text{ not free in } \chi.\mathsf{shrd}\\ \chi.\mathsf{shrd} \Vdash (\Delta_1.\mathsf{notRO} \Cap \Delta_2) = \varnothing\\ \chi.\mathsf{shrd} \Vdash (\Delta_2.\mathsf{notRO} \Cap \Delta_1) = \varnothing\\ \text{the output program typechecks}\end{cases}\end{array}$$

with:

$$\_, F \equiv \Gamma \boxminus \chi.\mathsf{shrd.inv} \qquad \_, R \equiv F \boxminus \mathsf{StackAllocCells}(T_1)$$

$$\chi_1 \equiv \begin{cases}\mathsf{vars} \equiv \chi.\mathsf{vars}\\ \mathsf{shrd} \equiv \chi.\mathsf{shrd} \Vdash \Delta_1\\ \mathsf{excl} \equiv \{\mathsf{pre} \equiv \chi.\mathsf{excl.pre},\ \mathsf{post} \equiv R\}\end{cases} \qquad \chi_2 \equiv \begin{cases}\mathsf{vars} \equiv \chi.\mathsf{vars}\\ \mathsf{shrd} \equiv \chi.\mathsf{shrd} \Vdash \Delta_2\\ \mathsf{excl} \equiv \{\mathsf{pre} \equiv R,\ \mathsf{post} \equiv \chi.\mathsf{excl.post}\}\end{cases}$$

where $\chi.\mathsf{shrd} \Vdash X$ is a shorthand for $\{\mathsf{inv} = (\chi.\mathsf{shrd.inv} \Vdash X), \mathsf{reads} = (\chi.\mathsf{shrd.reads} \Vdash X)\}$.

Fig. 23. The basic transformation `Loop.fission`.

$$\boxed{\begin{array}{l}\textbf{for}_{\ \chi}\ i \in r_i\ \{\\ \quad \Gamma_1\ T_1; \Delta_1\\ \quad \Gamma_2\ T_2; \Delta_2\\ \}\end{array}} \longmapsto \boxed{\begin{array}{l}T_1;\\ \textbf{for}_{\ \chi'}\ i \in r_i\ \{\\ \quad T_2;\\ \}\end{array}} \quad \begin{array}{l}\text{correct if:}\\ \begin{cases} i \text{ not free in } T_1\\ T_1 \text{ idempotent}\\ \Delta_1 \Cap \Delta_2.\mathsf{notRO} = \varnothing\end{cases}\end{array}$$

with: $\quad \begin{array}{l}\_, I' \equiv \Gamma_2 \boxminus \chi.\mathsf{excl.pre}\\ \_, I \equiv I' \boxminus \chi.\mathsf{shrd.reads}\end{array} \qquad \chi' \equiv \begin{cases}\mathsf{vars} \equiv \chi.\mathsf{vars}\\ \mathsf{shrd} \equiv \{\mathsf{inv} \equiv I,\ \mathsf{reads} \equiv \chi.\mathsf{shrd.reads}\}\\ \mathsf{excl} \equiv \chi.\mathsf{excl}\end{cases}$

correct if r_i is nonempty, or the program on the right typechecks successfully with $R \equiv \Gamma_2 \Vdash \Delta_1.\mathsf{produced}$

$$\boxed{\begin{array}{l}T_1;\\ \textbf{for}_{\ \chi'}\ i \in r_i\ \{\\ \quad T_2;\\ \}\\ \textbf{ghost}(R \longrightarrow \mathsf{IntoUninit}(R));\end{array}}$$

Fig. 24. The basic transformation `Loop.move_out`.

$\chi.\mathsf{shrd}$ and $\mathsf{StackAllocCells}(T_1)$, from the context Γ computed by our typechecker at the location just between T_1 and T_2.

Observe that the loop contracts χ_1 and χ_2 generated by the loop fission transformation may contain a larger set of resources than strictly necessary. We describe further on, in Section 6.6, a procedure for minimizing loop contracts.

Loop Invariant Code Motion. The basic transformation `Loop.move_out` applies to a loop with body $T_1; T_2$, where T_1 performs instructions that are redundant at every iteration. It produces as output a code that first executes T_1, exactly once, then executes a loop with body T_2. The transformation is formalized in Figure 24. We assume for simplicity the loop range to be provably nonempty, or T_1 to be provably deletable. Alternatively, T_1 could be wrapped into a conditional.

The key properties to check are that T_1 is the same for all iterations (it does not depend on i), can be safely deduplicated (it is idempotent as required by `Instr.dedup`), and does not interfer with the remaining instructions of the loop, described by T_2 (that is, the condition $\Delta_1 \Cap \Delta_2.\mathrm{notRO} = \varnothing$). Note that, contrarily to the `Instr.move` criteria, it is safe for T_2 to read resources modified by T_1. Additionally, we describe how to check whether T_1 is deletable in the case of a possibly empty loop as typechecking an auxiliary program, as we have done before.

Other Loop Transformations. There are other important loop transformations that we support.

- `Loop.fusion` (reciprocal of `Loop.fission`): it fuses two consecutive loops into a single one.
- `Loop.collapse` (reciprocal of `Loop.tile`): it collapses two nested loops into a single one.
- `Loop.hoist_alloc`: hoist a variable allocated inside a loop into an array allocated outside the loop; more generally, it hoists a matrix of dimension N allocated inside a loop into a matrix of dimension $N + 1$ allocated outside the loop.
- `Loop.shift`: reindex a loop by applying a positive or negative offset to its values.
- `Loop.scale`: reindex a loop using an index that takes either smaller or larger steps.
- `Loop.extend_range`: that extends the range of a loop by wrapping its body in a conditional.
- `Loop.unroll`: unrolls a loop whose range is statically known.
- `Loop.parallel`: to set or unset a parallel flag on a loop using our parallelizable criteria.

6.6 Transformations on Annotations

Modification to Contracts and Ghost Code. The semantics of a program is fully determined by its actual C code: it does not depend in any way on the *ghost* code nor on the function and loop contracts. Therefore, contracts may be freely modified, and ghost instructions may be freely inserted, deleted, or modified. The requirement is to reach, after one or several updates, a set of annotations for which the typechecking of the same C code succeeds. These modifications can be applied performed either directly by the programmer, or during the evaluation of transformations—without need for specific correctness criteria.

Minimization of Loop Contracts. The aforementioned loop transformations produce correct resource annotations, yet these annotations might be suboptimal for later transformations. Typically, the generated loop contracts would include clauses covering a set of resources possibly larger than strictly necessary. For example, after the basic loop fission transformation, the contract of the first loop would typically mention resources that are in fact only used by the instructions from the second loop. Mentioning unnecessary resources in a contract may impede the applicability of further transformations. OptiTrust therefore includes a procedure, implemented as a basic transformation, to *minimize* loop contracts. OptiTrust's combined transformations for loops systematically include a call to this procedure.

The loop contract minimization procedure takes as input a loop with contract χ, and updates this contract to χ', without modifying the code. The procedure depends on the usage map Δ computed for the instructions T that constitute the loop body.

$$\boxed{\textbf{for}\ {}^{\pi}_{\chi}\ i \in r_i\ \{T; \Delta\}} \quad \longmapsto \quad \boxed{\textbf{for}\ {}^{\pi}_{\chi'}\ i \in r_i\ \{T\}}$$

Intuitively, the contract χ' is obtained by filtering out and by weakening resources from χ, depending on their usage in Δ. First, if a resource is unused by T and thus is absent from Δ or has usage joinedFrac, then it is excluded from χ'. As a result, certain variables that were quantified in χ might no longer have occurrence in χ', hence they can be removed as well. Second, if a resource appears with fraction 1 in χ, yet this resource is marked as splittedFrac is Δ, then this resource is replaced with a read-only version of it. Technically, an additional fraction variable must be quantified in χ', and this fraction variable is used for describing the resource as read-only.

Internally, the implementation of contract minimization reuses our *minimization of triple* procedure (Section 5.4 and **??**). Details may be found in **??**.

Moving and Cancellation of Ghost Instructions. OptiTrust includes a transformation that attempts to remove pairs of ghost transformations that cancel each other. Indeed, the sequence $\textbf{ghost}(H \longrightarrow H')$; $\textbf{ghost}(H' \longrightarrow H)$ is equivalent to a no-op. More generally, the user as well as combined transformations may request a ghost instruction to be moved so as to be (logically) executed as early as possible in the program; or, symmetrically, to be executed as late as possible. Moving ghost instructions in such a way may lead to the apparition of pairs of cancellable pairs of ghost instructions; and, even when ghost instructions do not disappear, moving them away from, e.g., a loop kernel, may unlock certain transformations.

7 RELATED WORK

The most closely related frameworks were discussed in the introduction. In this section, we comment on the remaining related work, focusing in turn on each of the ingredients that constitute OptiTrust.

General-Purpose Compilers. General purpose compilers such as GCC or ICC are able to apply a large class of program optimizations, from the classic ones such as inlining, dead code elimination, move of instructions to more advanced ones such as loop fission, loop fusion, or loop reordering. The same transformations are available in OptiTrust, yet with three major differences. First, general-purpose compilers apply these transformations on an intermediate representation. In contrast, OptiTrust applies it at the source level, allowing to produce human-readable feedback. Second, general-purpose compilers relies on fully-automated procedures, often guided by heuristics, to determine what transformations to apply. In contrast, OptiTrust transformations are fully controlled by the programmer, either directly via basic transformations, or indirectly via combined transformations. Third, general-purpose compilers rely on static analysis applied to plain C code to determine whether certain transformations are applicable, and as a result may lack information to trigger a transformation. In contrast, OptiTrust leverages expressive resource typing information to justify the correctness of transformations, significantly enlarging the set of applicable transformations.

Guidance in General-Purpose Compilers. To introduce human guidance in general-purpose compilers, a common approach is to insert *pragmas* into the code. For example, Scout [Krzikalla et al. 2011] is a pragma-based tool for guiding source-to-source transformations that introduce vector instructions. As another example, Radtke and Weinzierl [2024] makes use of C++ attributes for switching between array-of-structures and structures-of-arrays over the scope of specific computation kernels; the compiler automatically inserts instructions for copying the data before and after the loop. A similar approach could be expressed as an OptiTrust transformation, by composing the `local_name` transformation for arrays (discussed in Section 6.4) with the `aos_to_soa` transformation (not discussed in the paper). The main limitation of *pragma*-based approaches is that they are ill-suited for describing sequences of optimizations. Indeed, there is no easy way to attach a pragma to a line of code that is generated by a first optimization. Kruse and Finkel [2018] suggest the possibility to stack up pragmas, by providing labels as additional pragma arguments: a second pragma may refer to the labels introduced by the transformation described in a first pragma. Yet, this approach does not scale up well beyond a handful of successive transformations. OptiTrust, in contrast, supports chains of dozens of transformations.

Domain-Specific Compilers. Another possible approach to overcome the limitations of general-purpose compilers is to leverage *domain specific languages* (DSL), such as Halide [Ragan-Kelley et al. 2013], TVM [Chen et al. 2018], Fireiron [Hagedorn et al. 2020a] (used at Nvidia), PartIR [Alabed et al. 2024] (used at DeepMind). Specialized compilers can benefit from carefully tuned heuristics. Yet, even for programs expressed in a specific DSL, the optimization search space remains vast, hence programmer guidance is key to achieve good performance. Halide and its descendants makes use of a script, called a *schedule*, for guiding the compilation strategy.

For DSLs, the language restriction is also their Achilles' heel: as soon as the user's application requires a single feature that falls outside of what the DSL can express, the programmer loses most if not all of the benefits of the DSL. In practice, DSLs typically support the possibility to include foreign functions (or, inlined general-purpose code), however these foreign functions must be treated as black box by the DSL compiler, preventing the applications of any domain-specific optimization across this black box.

In contrast to DSLs, OptiTrust sticks to a standard, general-purpose language. At the same time, OptiTrust retains the ability to manipulate domain-specific operations and to exploit transformations that are specific to these operations, as illustrated with the *reduce* function in our OpenCV case study. At any point in the transformation script, an occurrence of a domain-specific operation may be lowered into standard C code, thereby enabling further lower-level optimizations.

Code Transformations via Rewrite Rules. A rewrite rule maps a code pattern to another code pattern. A number of tools exploit rewrite rules to perform source-to-source transformations. For example, TXL [Cordy 2006] is a multi-language rewrite system, whose patterns are expressed at the level of syntax, using grammars. Coccinelle [Lawall and Muller 2018] allows the programmer to describe *semantic patches* in C code. CodeBoost [Bagge et al. 2003] applies the Stratego program transformation language [Bravenboer et al. 2008] to C++ code. CodeBoost can be used to turn high-level operations on matrices and vectors into typical high-performance source code.

OptiTrust provides a much more expressive language for describing transformations, going far beyond rewrite rules. Although many transformations *can* be encoded as rewrite rules, the encoding involved can be cumbersome or inefficient. For example, reconstructing a for-loop for a series of similar blocks of instructions can be encoded via rewrite rules, yet the blocks must be merged into the for-loop one by one. Other transformations, especially those involving contracts, would be challenging to express as rewrite rules. For example, *loop contract minimization* (Section 6.6) would require the rewrite rule to depend on side-conditions and meta-operations that involve resources and usage maps.

Intermediate Languages. The use of an intermediate language with simpler semantics is commonplace, both in the domain of compilation and in the domain of program verification. Let us cite a few examples. The Common Intermediate Language (CIL) serves as intermediate compilation language for the whole .NET ecosystem [Gough and Gough 2001]. Why3 [Filliâtre and Paskevich 2013] serves as intermediate verification language for C, Java, and Ada programs. Viper [Müller et al. 2017] serves as intermediate verification language for Java, Rust, Go, OpenCL, etc. Although intermediate languages are commonplace, We are not aware of any framework that leverages a translation into an intermediate language *and* provides a reciprocal translation back to the source language, with a round-trip property such as that provided by OptiTrust.

Source Code Manipulation Frameworks. Frameworks that offer more expressiveness than rewrite rules generally give access to the abstract syntax tree (AST) of the source code. Traditional compilers employ an AST, but they are not designed for synthesizing pieces of AST at the source level. Moreover, traditional compilers operate on intermediate representations, and lose the structure of the original code. These two limitations of general-purpose compilers have motivated the development of frameworks that are specifically designed to support code transformations (and code analyses) at the level of C code. ROSE [Quinlan 2000; Quinlan and Liao 2011] and Cetus [Bae et al.

2013; Dave et al. 2009] are two such frameworks that provide facilities for manipulating C ASTs. Source-to-source transformation frameworks have also been employed to produce code targeting GPUs [Amini 2012; Konstantinidis 2013; Lebras 2019]. These frameworks implement generic optimization strategies, in a similar fashion as general-purpose compilers. In contrast, OptiTrust leverages transformation scripts to guide the optimization of a specific program. Moreover, the OptiTrust infrastructure supports resource typing, which provides much more precise information than the classic static code analyses implemented in the frameworks such as ROSE and Cetus.

Transformation Scripts. Expressing a series of source-level transformations for a specific program can be done by means of a transformation script. Such scripts have appeared in particular in the context of polyhedral transformations [Bagnères et al. 2016b; Bondhugula et al. 2008], for example in Loopy [Namjoshi and Singhania 2016] and in work by Zinenko et al. [2018a]. CHiLL [Chen et al. 2008; Rudy et al. 2011] includes transformations that go beyond the polyhedral model. It has been applied to generate finely tuned CUDA code from high-level linear algebra kernels. POET [Yi and Qasem 2008; Yi et al. 2014] is a scripting language for performing program transformations, for C/C++ as well as other languages. POET has been employed to generate optimized code for linear algebra kernels, including semi-automated exploration of a search space of possible optimizations.

Several pieces of work already discussed in the introduction exploit transformation scripts. Halide [Ragan-Kelley et al. 2013], TVM [Chen et al. 2018] feature schedules that can be viewed as transformation scripts. Elevate [Hagedorn et al. 2020b] expresses the transformation script in the form of a composition of functions. ATL [Liu et al. 2022] leverages "tactic"-based proof scripts as support for expressing transformations scripts. LARA consists of a transformation script featuring declarative queries as well as arbitrary JavaScript instructions.

All this related work demonstrates a strong interest in leveraging transformation scripts for putting control of optimizations in the hand of the programmer. Systems differ in what language they targeted, and what transformations they support. None of the aforementioned systems support in their transformation scripts a system for targeting program points with the expressiveness and conciseness offered by OptiTrust targets. Moreover, as far as we know, LARA [Silvano et al. 2019] and OptiTrust are the only two frameworks making use of transformation scripts for applying general-purpose transformations at the level of C code. OptiTrust is the first to demonstrate the use of transformation scripts to produce high-performance code for state-of-the-art benchmarks. Moreover, unlike LARA, OptiTrust checks that the transformations requested by the programmer preserve the semantics of the code.

Proof-Transforming Compilation. The notion of *Proof Carrying Code* [Necula 1998] refers to the idea that compilers could be instrumented to carry invariants from high-level source code down to low-level code. The original line of work on Proof Carrying Code did not aim at full functional correctness properties, but rather focused on simpler invariants capturing safety properties, such as the absence of out-of-bound accesses.

Subsequent work introduced the notion of *Proof-Transforming Compilation* to refer to a compiler that takes as input a formally-verified program and produces as output compiled code accompanied by a formal proof (i.e., a proof tree in a program logic) that the compiled code satisfies the same functional correctness specification as the input program. In particular, the PhD work of César Kunz [Barthe et al. 2009; Kunz 2009] shows how to realize proof-transforming compilation for standard compiler optimizations, applied at the level of the RTL intermediate language. More recently, the work on Alpinist [Sakar et al. 2022] demonstrates the feasibility, for a small number of GPU-oriented optimizations, of transforming GPU code while preserving logical invariants.

Our results obtained so far with OptiTrust demonstrates the feasibility, for a fair number of general-purpose code optimizations, of transforming C code while preserving resource-based invariants. In future work, we look forward to extending OptiTrust in order to handle richer logical invariants and to produce optimized programs accompanied by formal proofs of correctness.

Separation Logic. OptiTrust leverages a standard Separation Logic. The most closely related program logics are VST [Cao et al. 2018], a program verification tool for C, and RefinedC [Sammler et al. 2021], a very expressive type system for C. Both these systems are grounded on the Iris framework [Jung et al. 2018a,b], at this day the most advanced formalization of Concurrent Separation Logic. Other tools, such as Alpinist [Sakar et al. 2022] leverage Viper's *dynamic frames* technique [Müller et al. 2017], a cousin of Separation Logic.

Fractional resources [Boyland 2003] are nowadays considered a standard ingredient of Separation Logic [Jung et al. 2018a]. Following common practice, OptiTrust leverages the notion of fractional resources to describe read-only resources. The technique of making fractions essentially transparent to the end-user is directly inspired by the work by Heule et al. [2013] implemented in the Chalice verification tool.

OptiTrust is, as far as we know, the first framework to compute usage information, describing precisely how the Separation Logic resources available during the evaluation of a subterm are actually used by that subterm; and the first framework to exploit Separation Logic information for guiding the behavior of programmer-requested code transformations (as, e.g., in `local_name`, described in Section 6.4).

Contract Inference. OptiTrust currently requires the programmer to annotate the input program with function and loop contracts. One may wonder the extent to which such contracts could be automatically inferred, at least for reasonably simple programs. Journault and Miné [2018] show that, by leveraging abstract interpretation, for functions such as matrix-multiplication or similar linear algebra operations, full functional correctness specifications can be automatically computed. Spies et al. [2024] leverage *bi-abduction* in Separation Logic (a technique at the heart of the *infer* automated program analysis tool [Calcagno et al. 2019]) to demonstrate automatic inference of specifications in array- and pointer-manipulating functions of up to a dozen lines of C code. We leave it to future work to integrate ideas from specification synthesis into OptiTrust.

REFERENCES

2022. Unsequenced functions. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2956.htm ISO/IEC JCT1/SC22/WG14 document N2956.

Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, Adam Paszke, Norman A. Rink, Michael Schaarschmidt, Timur Sitdikov, Agnieszka Swietlik, Dimitrios Vytiniotis, and Joel Wee. 2024. PartIR: Composing SPMD Partitioning Strategies for Machine Learning. arXiv:2401.11202 [cs.LG] https://arxiv.org/abs/2401.11202

Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. 2020. Programming languages for data-Intensive HPC applications: A systematic mapping study. *Parallel Comput.* 91 (2020), 102584. https://doi.org/10.1016/j.parco.2019.102584

Mehdi Amini. 2012. *Source-to-source automatic program transformations for GPU-like hardware accelerators.* Ph.D. Dissertation. Ecole Nationale Supérieure des Mines de Paris.

Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. 2013. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *Int. J. Parallel Program.* 41, 6 (2013), 753–767. https://doi.org/10.1007/S10766-012-0211-Z

O.S. Bagge, K.T. Kalleberg, M. Haveraaen, and E. Visser. 2003. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation.* 65–74. https://doi.org/10.1109/SCAM.2003.1238032

Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016a. Opening Polyhedral Compiler's Black Box. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO).*

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

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 177–183. https://doi.org/10.1145/3317550.3321441

Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. 2018. Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors. In *24th International Conference on Parallel and Distributed Computing (Euro-Par) (Lecture Notes in Computer Science, Vol. 11014)*. Springer, Cham, 749–763. https://doi.org/10.1007/978-3-319-96983-1_53

Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. 2009. Certificate Translation for Optimizing Compilers. *ACM Trans. Program. Lang. Syst.* 31, 5, Article 18 (jul 2009), 45 pages. https://doi.org/10.1145/1538917.1538919

João Bispo and João MP Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565. https://www.sciencedirect.com/science/article/pii/S2352711019302122/pdf

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated Formal Methods*, Nadia Polikarpova and Steve Schneider (Eds.). Springer International Publishing, Cham, 102–110.

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

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.* 72, 1–2 (jun 2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok. Yang. 2019. Go Huge or Go Home: POPL'19 Most Influential Paper Retrospective. https://blog.sigplan.org/2020/03/03/go-huge-or-go-home-popl19-most-influentialpaper-retrospective/

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/S10817-018-9457-5

Arthur Charguéraud. 2020a. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (aug 2020), 34 pages. https://doi.org/10.1145/3408998

Arthur Charguéraud. 2020b. Separation Logic for Sequential Programs (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 116 (aug 2020), 34 pages. https://doi.org/10.1145/3408998

Gaurav Chaurasia, Jonathan Ragan-Kelley, Sylvain Paris, George Drettakis, and Frédo Durand. 2015. Compiling high performance recursive filters. In *Proceedings of the 7th Conference on High-Performance Graphics, HPG 2015, Los Angeles, California, USA, August 7-9, 2015*, Michael C. Doggett, Steven E. Molnar, Kayvon Fatahalian, Jacob Munkberg, Elmar Eisemann, Petrik Clarberg, and Stephen N. Spencer (Eds.). ACM, 85–94. https://doi.org/10.1145/2790060.2790063

Lorenzo Chelini, Martin Kong, Tobias Grosser, and Henk Corporaal. 2021. LoopOpt: Declarative Transformations Made Easy. In *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems* (Eindhoven, Netherlands) *(SCOPES '21)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3493229.3493301

Chun Chen, Jacqueline Chame, and Mary W. Hall. 2008. *CHiLL: A Framework for Composing High-Level Loop Transformations.* Technical Report 08-897. University of Southern California.

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. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. USENIX Association. https://www.usenix.org/system/files/osdi18-chen.pdf

James R Cordy. 2006. The TXL source transformation language. *Science of Computer Programming* 61, 3 (2006), 190–210.

Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. 2009. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer* 42, 12 (2009), 36–42. https://doi.org/10.1109/MC.2009.385

Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. 2022. A survey of software implementations used by application codes in the Exascale Computing Project. *The International Journal of High Performance Computing Applications* 36, 1 (2022), 5–12. https://doi.org/10.1177/10943420211028940

Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming* 21, 5 (october 1992), 313–348.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—Where Programs Meet Provers. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 125–128. http://hal.inria.fr/hal-00789533

Jean-Christophe Filliâtre. 2003. *Why: a multi-language multi-prover verification tool.* Research Report 1366. LRI, Université Paris Sud. http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. 2002. Extended Static Checking for Java. In *Programming Language Design and Implementation (PLDI).* 234–245. http://www.soe.ucsc.edu/~cormac/papers/pldi02.ps

John John Gough and K John Gough. 2001. *Compiling for the. Net Common Language Runtime.* Prentice Hall PTR.

Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020a. Fireiron: a data-movement-aware scheduling language for GPUs. In *Proceedings of the ACM International Conf. on Parallel Architectures and Compilation Techniques.* 71–82.

Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020b. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (aug 2020), 29 pages. https://doi.org/10.1145/3408974

Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. 2013. Abstract Read Permissions: Fractional Permissions without the Fractions. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–334.

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022).* Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576341

Matthieu Journault and Antoine Miné. 2018. Inferring functional properties of matrix manipulating programs by abstract interpretation. *Form. Methods Syst. Des.* 53, 2 (oct 2018), 221–258. https://doi.org/10.1007/s10703-017-0311-x

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018a. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Yamato Kanetaka, Hiroyasu Takagi, Yoshihiro Maeda, and Norishige Fukushima. 2024. SlidingConv: Domain-Specific Description of Sliding Discrete Cosine Transform Convolution for Halide. *IEEE Access* 12 (2024), 7563–7583. https://doi.org/10.1109/ACCESS.2023.3345660

Vasilios Kelefouras and Georgios Keramidas. 2022. Design and Implementation of 2D Convolution on x86/x64 Processors. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3800–3815. https://doi.org/10.1109/TPDS.2022.3171471

Athanasios Konstantinidis. 2013. *Source-to-source compilation of loop programs for manycore processors.* Ph. D. Dissertation. Imperial College London.

Michael Kruse and Hal Finkel. 2018. A Proposal for Loop-Transformation Pragmas. *CoRR* abs/1805.03374 (2018). arXiv:1805.03374 http://arxiv.org/abs/1805.03374

Olaf Krzikalla, Kim Feldhoff, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. 2011. Scout: A Source-to-Source Transformator for SIMD-Optimizations. In *Euro-Par Workshops (2) (LNCS, Vol. 7156).* Springer.

César Kunz. 2009. *Proof preservation and program compilation.* Ph. D. Dissertation. École Nationale Supérieure des Mines de Paris. https://pastel.archives-ouvertes.fr/pastel-00004940/file/thesis-ckunz.pdf

Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18).* USENIX Association, 13 pages.

Youenn Lebras. 2019. *Code optimization based on source to source transformations using profile guided metrics.* Ph. D. Dissertation. Université Paris-Saclay (ComUE). https://www.theses.fr/2019SACLV037.pdf

Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2017. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering.* IOS Press, 104–125. https://doi.org/10.3233/978-1-61499-810-5-104

Kedar S. Namjoshi and Nimit Singhania. 2016. Loopy: Programmable and Formally Verified Loop Transformations. In *Static Analysis - 23rd International Symposium, SAS (LNCS, Vol. 9837).* Springer.

George Ciprian Necula. 1998. *Compiling with proofs.* Ph. D. Dissertation. Carnegie Mellon University.

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Pedro Pinto, Joao Bispo, Joao Cardoso, Jorge Gomes Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovic, Martin Golasowski, Katerina Slaninova, Radim Cmar, et al. 2020. Pegasus: Performance Engineering for Software Applications Targeting HPC Systems. *IEEE Transactions on Software Engineering* (2020). https://repositorio-aberto.up.pt/bitstream/10216/127756/2/405707.pdf

Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel processing letters* 10, 02n03 (2000), 215–226. https://digital.library.unt.edu/ark:/67531/metadc741175/m2/1/high_res_d/793936.pdf

Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. 1.

Pawel K. Radtke and Tobias Weinzierl. 2024. Compiler support for semi-manual AoS-to-SoA conversions with data views. arXiv:2405.12507 [cs.PL] https://arxiv.org/abs/2405.12507

Jonathan Ragan-Kelley. 2023. Technical Perspective: Reconsidering the Design of User-Schedulable Languages. *Commun. ACM* 66, 3 (feb 2023), 88. https://doi.org/10.1145/3580370

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Conference on Programming Language Design and Implementation*. 12 pages. https://doi.org/10.1145/2491956.2462176

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. 2011. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg.

Ömer Sakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. 2022. Alpinist: An Annotation-Aware GPU Program Optimizer. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13244)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 332–352. https://doi.org/10.1007/978-3-030-99527-0_18

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 158–174. https://doi.org/10.1145/3453483.3454036

Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R. Beccari, Luca Benini, Loïc Besnard, João Bispo, Radim Cmar, João M.P. Cardoso, Carlo Cavazzoni, Daniele Cesarini, Stefano Cherubin, Federico Ficarelli, Davide Gadioli, Martin Golasowski, Antonio Libri, Jan Martinovič, Gianluca Palermo, Pedro Pinto, Erven Rohou, Kateřina Slaninová, and Emanuele Vitali. 2019. The ANTAREX domain specific language for high performance computing. *Microprocessors and Microsystems* 68 (2019), 58–73. https://doi.org/10.1016/j.micpro.2019.05.005

Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. 2024. Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq. *Proc. ACM Program. Lang.* 8, PLDI, Article 183 (jun 2024), 25 pages. https://doi.org/10.1145/3656413

Manish Vachharajani, Neil Vachharajani, David I. August, and Spyridon Triantafyllis. 2003. Compiler Optimization-Space Exploration. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 204. https://doi.org/10.1109/CGO.2003.1191546

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Conference on Programming Language Design and Implementation* (San Jose, California, USA). Association for Computing Machinery, 12 pages. https://doi.org/10.1145/1993498.1993532

Qing Yi and Apan Qasem. 2008. Exploring the Optimization Space of Dense Linear Algebra Kernels. In *LCPC*.

Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing Compiler Optimizations through Programmable Composition for Dense Matrix Computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, United Kingdom) *(MICRO-47)*. IEEE Computer Society, USA, 596–608. https://doi.org/10.1109/MICRO.2014.14

Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. 2018a. *Declarative Transformations in the Polyhedral Model*. Research Report RR-9243. https://hal.inria.fr/hal-01965599

Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2018b. Visual Program Manipulation in the Polyhedral Model. *ACM Trans. Archit. Code Optim.* 15, 1, Article 16 (mar 2018), 25 pages. https://doi.org/10.1145/3177961