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

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

Producing formally verified code is a challenging, time-consuming task. Producing highly optimized code is another challenging, time-consuming task. In this work, we tackle the problem of producing code that is both formally verified and highly optimized. Concretely, we present OptiTrust, an interactive source-to-source optimization framework that aims to provide formal guarantees about the optimized code.

In OptiTrust, the programmer starts with a naive, unoptimized implementation, shown to satisfy a formal specification expressed in separation logic. The programmer then develops a script describing a series of code transformations. The framework provides continuous feedback in the form of human-readable *diff*s over C-style syntax. OptiTrust supports classic transformations, including advanced loop and storage transformations. It can also be extended with user-defined transformations, which may be expressed either as direct modification of the abstract syntax tree, or as the composition of existing transformations.

Crucially, OptiTrust leverages a formal validation technique to guarantee that the optimized code satisfies the same formal specification as the original unoptimized code. Our approach is based on the old idea of proof-carrying code: each transformation refines not only the code, but also refines the loop invariants; moreover, it inserts or updates the necessary separation logic *ghost* operations. Regardless of how the transformations are implemented and exploited, as long as the final output code typechecks—in the sense of separation logic—against the original specification, we have the formal assurance that the output code is correct.

We demonstrate how OptiTrust enables producing verified optimized code on three case studies. First, we parallelize a dot-product computation to give a gentle introduction. Second, we show how to reproduce a compiler-optimized matrix-multiplication from TVM: an industrial-strength, semi-automated, specialized compiler. Third, we show how to reproduce a programmer-optimized blur computation from OpenCV: an optimized, reference library for graphics computations.

1 INTRODUCTION

1.1 Motivation

Producing optimized code is known to be challenging. Producing optimized code with formal correctness guarantees is an even harder problem. Let us first discuss at a high level the possible approaches for producing optimized code, then discuss for each approach the ways in which formal guarantees can be derived.

Program optimization involves a tradeoff between optimization effort and program performance. At one end of the spectrum, the use of a fully automated compiler might provide sufficient performance, with almost no optimization effort. At the other end of the spectrum, carefully writing optimized code by hand often provides superior performance, with much higher optimization effort. As of 2025, numerous state-of-the-art high-performance libraries and compute-intensive programs continue to be developed by writing low-level, optimized code by hand. In-between those two extreme approaches are semi-automated compilers, most of which are targeting domain-specific languages. There, the user provides an unoptimized piece of code together with an *optimization script* that provides the compiler with strategic hints on how to optimize the code. This approach has been popularized in particular by Halide [Ragan-Kelley et al. 2013], specialized for image processing, and TVM [Chen et al. 2018], specialized for machine learning.

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

Regardless of the degree of automation considered, a central question is that of the correctness of the optimized code. When aiming for program verification, the question is: how can the programmer obtain formal guarantees that the optimized code actually performs the intended computations?

Consider first the case of formally verifying optimized code that was written by hand. Carrying out formal verification is known to be highly time-consuming, even for a program that has a moderate length and exhibits reasonably simple invariants. In general, a highly optimized program is a lot longer than its unoptimized counterpart—typically, by more than one order of magnitude, as illustrated in our case studies. Moreover, the program invariants and, in particular, the array-index manipulations, are generally much more intricate in optimized code. As a result, the effort required for directly applying formal verification methods to optimized code is generally prohibitive.

Consider now the case of a program optimized by a compiler. One might think that it is sufficient to formally verify the input program, then to throw away the proof before providing the source code as input to a trusted optimizing compiler. Unfortunately, compilers infamously suffer frequently from subtle bugs that can cause the production of programs that appear to behave correctly most of the time, but might silently produce incorrect results for specific input data on specific runs. Several milestone papers have reported on the pervasiveness of such compiler bugs, including in mainstream compilers such as GCC and LLVM [Sun et al. 2016; Yang et al. 2011].

Hence, a key research question is: how can one increase the trust in the output of optimizing compilers, both for automated compilers and for user-guided compilers? This long-standing challenge has been highlighted by the Turing Award recipient Tony Hoare [2003], who described it as "*A grand challenge for computing research: the construction and application of a verifying compiler that guarantees correctness of a program before running it.*" The goal of our work is to make some progress in that direction, focusing on a restricted array-based programming language, focusing on certain optimizations that are typically found in the upper half of a compiler pipeline, and focusing on user-guided compilers (as opposed to automated compilers).

1.2 Related Work on Trustworthy Compilation

Over the past three decades, three categories of approaches have emerged for tackling the problem of producing guarantees on compiled code.

Translation Validation. The idea of *translation validation* is to use a tool to check that an input program and an output program are semantically related. This idea dates back from the 70's, originating in Samet's PhD thesis [1975]. Pnueli et al. [1998] rediscovered it, and introduced the terminology *translation validation*. The technique was then popularized by Necula [2000], who implemented a translation validation infrastructure in the intermediate phases of GCC. At a high level, in Necula's work, translation validation is applied at the scale of one transformation, using basic blocks as synchronization points, symbolic evaluation to infer a simulation relation, and a custom automated solver to prove the correctness of this relation. Necula's work considered only *structure-preserving* transformations. Subsequent work on translation validation showed how to handle non structure-preserving transformations such as loop inversion, loop unrolling, loop fusion and strength reduction [Goldberg et al. 2005; Zuck et al. 2002], as well as loop-peeling and induction variable strength reduction using *equality saturation* [Tate et al. 2009].

When the input code is expressed in a domain-specific language, translation validation can be easier to exploit at the scale of several compiler passes. For example, Clément and Cohen [2022] instrument the Halide compiler [Ragan-Kelley et al. 2013]. Halide is an industrial-strength domain-specific compiler for image processing used, e.g., to optimize code running in Photoshop and YouTube. In Clément and Cohen's extension, each variable appearing in the output code is accompanied by a statement of the logical value that it evaluates to, this value being expressed

in terms of the variables from the purely functional input code. The applicability of the proposed validator is demonstrated on realistic Halide programs.

In general, though, the fundamental limitation of translation validation is that more complex transformations make it harder to design a validator for checking semantic preservation. Moreover, even if one is able to devise a validator that works in practice, the implementation of such a validator may end up being significantly complex. While the implementation of a validator remains simpler than that of the transformation it checks, one may worry about the trustworthiness of the validator. In the *verified validation* approach, one applies deductive verification techniques to formally prove the correctness of a validator program—which itself checks the correctness of a transformation by relating input and output programs. For example, verified validators have been presented for lazy code motion [Tristan and Leroy 2009], software pipelining [Tristan and Leroy 2010], loop-invariant code motion and a few other transformations [Tristan et al. 2011]. Each of these results, which are specific to a relatively small class of optimizations, involves a substantial amount of work.

Formally Verified Compilers. Constructing a machine-checked proof for a validator that checks the result of a transformation is already a great achievement; constructing a machine-checked proof directly for the transformation itself requires even more work, yet provides even stronger guarantees. Concretely, rather than checking, a posteriori, that a transformation preserves the semantics for each input program, one may aim to check, a priori, that the transformation will always preserve the semantics, for any input program. The most notable examples of *formally verified compilers* include CompCert [Leroy 2009] for C code, and CakeML [Kumar et al. 2014] for ML code. Both are developed using interactive proof assistants—Rocq and HOL4, respectively. In these compilers, most compiler passes are directly verified. Only a few complex or costly passes, such as register allocation, are treated using the aforementioned verified validation approach.

Overall, the major challenge with the direct verification approach is the huge proof effort that it involves. A secondary, yet related challenge stems from the fact that, to decrease the cost of verification, the verified compilers are implemented in a purely functional subset (technically, as a shallow embedding in the theorem prover), hence the compilers may suffer from inefficiencies. For example, executing a purely functional implementation of a graph coloring algorithm involved for register allocation may create a bottleneck with respect to compilation time.

Due to the amount of effort required to verify even classic compiler passes, verified compilers remain, performance-wise, behind state-of-the-art compilers. Moreover, because both hardware and compiler optimizations are continuously evolving, one can expect that CompCert-style compilers will, at any given point in time, lag behind unverified compilers in terms of performance.

Another possibility for leveraging interactive proof assistants is to derive correct-by-construction optimized programs in the style proposed by Liu et al. [2022]. Their ATL language, similarly to Halide [Ragan-Kelley et al. 2013], features purely functional constructs for describing computations over multi-dimensional arrays. ATL is embedded in the Rocq proof assistant, enabling the user to optimize code by applying rewrite rules inside Rocq. Each of these rewrite rules takes the form of a Rocq lemma, proved once and for all to preserve the semantics—that is, to compute equal results. The set of rules includes expressive transformations, some beyond the scope of Halide, and this set may be extended by the user. With the ATL approach, transformations are inherently accompanied by machine-checked proofs of correctness. In other words, no final validation is needed. Once optimized, ATL programs are then translated into imperative C code—this translation is not formally verified. In summary, whereas CompCert and CakeML provide verified compilers that target automated compilation, ATL provides a verified compiler intended for user-guided, high-level compilation steps.

Proof Carrying Code. The third approach for producing guarantees for compiled code is to assume a formal specification for the input code, and to exhibit a proof (i.e., a derivation) establishing that the output code satisfies the same formal specification. The idea of *proof-carrying code* describes a program equipped with a specification and accompanied by sufficiently many formal annotations (e.g., loop invariants) to allow an automated tool to check that the code satisfies its specification. This tool might consist of a simple typechecker, or it might leverage advanced SMT-solvers.

A *proof-preserving transformation* is a transformation that takes an input proof-carrying code, and refines it into an output proof-carrying code. For a whole compilation chain over proof-carrying code, an appropriate terminology is *proof-preserving compilation.*[1]

The concept of *proof-carrying code* has been introduced in Necula's seminal work [Necula 1998], as a generalization of Typed Intermediate Language (TIL) [Tarditi et al. 1996] and of Typed Assembly Language (TAL) [Morrisett et al. 1999]. All these works aimed to carry rich type information down through the compilation chain in order to offer the guarantee that the machine code would not produce certain types of errors, in particular with respect to memory accesses. Concretely, Necula's compiler applies the concept to the case of invariants capturing safety properties, such as the absence of out-of-bound accesses, and carries these properties from a type-safe subset of C down to optimized machine code. The compiler leverages SMT-provers to check that annotated code indeed satisfies the intended safety properties.

Subsequent work tackled the generalization to general functional correctness properties, towards exploiting *proof-preserving compilation* to its full potential. Bannwart and Müller [2005] showed how to transform Hoare logic derivations from Java source code down to Java bytecode, through non-optimizing compilation. Müller and Nordio [2007] showed how to handle complications related to abrupt termination. Nordio et al. [2008] applied similar ideas to transport proofs from a subset of the Eiffel programming language to Microsoft's Common Intermediate Language.

In a distinct line of work, Barthe et al. [2006, 2009] showed how to transform Hoare logic derivations through several standard compiler optimizations, all expressed at the level of the RTL intermediate language. In particular, their work covers constant propagation, loop induction variable strength reduction, dead register elimination, common subexpression elimination, copy propagation, unreachable code elimination, register allocation and function inlining.

More recently, the work on Alpinist [Şakar et al. 2022, 2025] demonstrates the feasibility of transforming GPU code while preserving Viper-based proofs developed using the VerCors framework [Blom et al. 2017]. The Viper prover [Müller et al. 2017] is built on *dynamic frames*, a cousin of separation logic [Reynolds 2002]. The transformations covered are loop tiling, loop unrolling, data prefetching, matrix linearization, and kernel fusion. In Alpinist, the programmer invokes a transformation by means of inserting a *pragma* at the relevant location in the code.

In summary, while the proof-carrying code approach appears as a promising way to tackle the *verified compiler grand challenge*, prior work has not yet demonstrated the ability to handle full functional correctness properties for highly optimized programs that are derived through a chain of transformations. Our work aims to demonstrate this possibility, at least for a couple of realistic case studies.

1.3 Contribution

Description of OptiTrust. In this work, we present a *proof-preserving compilation* approach. It allows one to start from an unoptimized program verified in separation logic [Reynolds 2002] and,

[1]The terminology that we advocate follows the patterns of the standard notions of *semantic-preserving transformation* and of *type-preserving transformation.* In the literature, similar notions to *proof-preserving transformation/compilation* have appeared under various names, including: *proof-transforming compilation* [Bannwart and Müller 2005; Müller and Nordio 2007], *certificate translation* [Barthe et al. 2006, 2009], and *annotation-aware transformation* [Şakar et al. 2022].

via a series of user-guided source-to-source transformations, turn it into a verified program with state-of-the-art performance. Both the unoptimized program and the optimized program consist of *proof-carrying code*. We implement this approach in a framework called OptiTrust and demonstrate its applicability to advanced code transformations. Concretely, OptiTrust code consists of program expressions, annotated with specifications, invariants, and *ghost operations*. Ghost operations are a standard mechanism in program verification that is used pervasively in separation logic to describe updates to the way the memory is described in the logic.

The two key ingredients of OptiTrust are a typechecker and a collection of proof-preserving transformations. The typechecker is able to verify that the code satisfies its specification, by leveraging its code annotations. Technically, OptiTrust implements a *proof-checker* that verifies programs by means of *typechecking* separation logic derivation trees. Thus, in the context of OptiTrust where programs are annotated with specifications, *typechecking* is synonymous for *proof-checking*. The transformations refine not only the input code but also its annotations, enabling the output code to successfully typecheck. After one or several transformations, the fact that the optimized code typechecks against the same specification as the input code formally guarantees the correctness of the optimization process.

Importantly, the program invariants that are carried all the way through the compilation process can be exploited at any step to guide the application of advanced optimizations that would have been out of reach for a traditional compiler. Concretely, an OptiTrust transformation may leverage hypotheses provided in the specification (e.g., that an array size is a multiple of 64), or described in invariants that could be extremely hard to re-synthesize by means of static analysis of the code.

This refinement process allows exploiting an effective synergy between the manipulation of formal invariants and the process of applying code optimizations. On the one hand, formal invariants help for optimization: nontrivial transformations that a compiler considering code alone would have been unable to apply can be justified through the invariants. On the other hand, the step-by-step optimization process helps for verification: a highly optimized output program may involve complex invariants, but those complex invariants can be automatically constructed through stepwise refinements of the simpler invariants associated with the unoptimized, input code.

Positioning of OptiTrust. OptiTrust is not a translation validation approach, because it does not relate the semantics of the output code with the semantics of the input code. OptiTrust is not a formally verified compiler in the strict sense of the term, because the transformation implementations are not formally verified. Instead, OptiTrust follows the proof-carrying code approach, whereby a tool checks that the output code satisfies the desired specification.

Let us compare specifically with the work by Barthe et al. [2009]. First, our work leverages not Hoare logic but separation logic. The past 25 years of work on separation logic has demonstrated that it is the tool of choice for modular program verification, especially reasoning about parallelism and concurrency, two features that are ubiquitous in high-performance code. Second, our work covers transformations over loops, which they did not handle. Third, our work operates on high-level source code and allows providing feedback to the programmer.

Let us compare against Alpinist [Şakar et al. 2022, 2025]. On the one hand, Alpinist considers a slightly larger language featuring GPU constructs, which are not yet supported by OptiTrust. On the other hand, OptiTrust goes further in two important directions. First, OptiTrust presents a broader set of transformations, including in particular transformations on memory storage. Second, whereas Alpinist does not support chaining transformations via higher-order composition, OptiTrust is the first proof-preserving tool to demonstrate the ability to chain transformations in such a way as to reproduce existing, highly optimized code. This is important because higher-order composition of transformations is known to be a key ingredient for specifying complex optimizations at a high

level of abstraction [Hagedorn et al. 2020b; Ikarashi et al. 2025; Ragan-Kelley 2023]. As we explain later on, handling chains of transformations poses significant additional challenges compared with applying transformations in isolation. Indeed, numerous transformations need to introduce ghost operations for reflecting on the changes in the way memory cells are being manipulated. Then, during the application of subsequent transformations, particular care is required for either eliminating or for pushing away these ghost operations.

Finally, let us compare against ATL [Liu et al. 2022]. A key feature that ATL and OptiTrust have in common is their interactivity: users write transformation scripts through which they control what transformations to apply, and where to apply them in the code. Compared with ATL, the added value of OptiTrust is that it supports transformation of effectful code, hence allows the user to control memory-related transformations, e.g., memory reuse optimizations. Handling imperative features adds significant challenges, for specifying side effects, for typechecking side effects using loop contracts and ghost operations, and for updating those annotations through every transformation.

Limitations. Generating high-performance code with formal guarantees for general-purpose programs is a highly ambitious goal. In the work so far, we have considered several simplifications.

We have so far considered case studies that involve mainly array-manipulating programs, and have implemented transformations and separation logic permissions suited for optimizing such programs. The transformations that we have implemented so far are mainly those that were involved in the case studies considered. Besides, for simplicity, we currently ignore complications related to arithmetic overflows, and we treat floating point numbers as reals. We leave it to future work to apply OptiTrust to programs involving recursive computations and pointer-based data structures, and to formally handle fixed-size number representations. Note that separation logic is famously known for being able to handle pointer-based data structures such as linked lists or mutable trees [Charguéraud 2020; Reynolds 2002], and for being able to handle advanced concurrent programming patterns [Jung et al. 2018a; O'Hearn 2019].

OptiTrust represents programs in an imperative λ-calculus that includes constructs for loops and for pointer arithmetic, called Optiλ. The framework relies on a Clang-based frontend for parsing a subset of C++ syntax—C constructs, for the most part. This front-end is outside the trusted code base, because the verification process operates on Optiλ. Regarding the optimized output code, OptiTrust provides a way to export it as C or C++ code. This translation is currently not formalized—a similar limitation as in ATL [Liu et al. 2022]. Doing so would be highly technical due to the daunting complexity of the C and C++ standards. Our long-term plan is to instead directly produce verified assembly code. In summary, at this stage, the formal guarantees that we establish only concern the Optiλ code with respect to the specification that it carries.

A central element of the trusted code base is our typechecker, which implements a standard separation logic, simply adapted to the syntax of our internal language. Only our treatment of loops is somewhat novel; we therefore present a paper proof that our proof rule for loops is derivable from the standard separation logic rule for `for`-loops and of that for `parallel_for`-loops. By building OptiTrust on top of standard separation logic reasoning rules, we can be reasonably confident in its soundness. To achieve utmost confidence, our long-term plan is to extract from OptiTrust programs separation logic derivations expressed in terms of constructs defined in the state-of-the-art Iris framework [Jung et al. 2018a], instantiated on a concrete output programming language. This way, we could use Rocq's typechecker to independently establish the correctness of programs with respect to the formal semantics of the output language—thereby achieving *foundational proof carrying code* [Appel 2001].

In summary, we present a framework that can readily be exploited to optimize certain classes of programs, while acknowledging that future work remains necessary to achieve full generality and utmost formal guarantees.

Summary of Contributions. The contributions of this paper can be summarized as follows.

(1) We present the first framework that allows applying sequences of diverse, advanced code transformations directly on an imperative source code, with formal guarantees about the correctness of the output code. Prior work either supported transformations only at the level of functional code, or did not demonstrate the ability to chain multiple, nontrivial transformations. As we show in one of the case studies, we are able to reproduce manually optimized code by combining user interactivity with a sufficiently rich collection of transformations.
(2) We introduce a technical ingredient, called *usage map*, that allows to easily compute, for each instruction or span of instructions, how the separation logic resources available are used by the instructions considered. As we show, this usage information can be key to guiding the code generation process in certain advanced transformations, and can also be exploited to implement a certain form of *contract minimization*.
(3) We present three case studies demonstrating the practicality of OptiTrust. Two of them reproduce existing, production-quality code, for which no formal proof of correctness was previously available. Moreover, our proof scripts for these case studies is short enough that we are confident that the OptiTrust approach provides formal guarantees at a cost much lower than that of applying deductive verification methods directly to the optimized code.

OptiTrust is open-source, and available from: https://github.com/charguer/optitrust.

1.4 Contents of the Paper

We first present the features of OptiTrust through our case studies, in Section 2. Then, we present the key internals of OptiTrust. In Section 3, we present Optiλ, the internal language of OptiTrust, and describe at a high level the bidirectional translation between a C-style surface language and Optiλ. In Section 4, we explain the core of our separation logic typechecker. This part presents mostly standard separation logic concepts, but follows an algorithmic presentation rather than the usual declarative presentation of the reasoning rules. In Section 5, we present a set of representative code transformations. In Section 6, we explain how we compute a *usage map* for every subterm, and discuss how usage maps are exploited for minimizing loop contracts and for guiding advanced code transformations. Finally, we discuss additional related work in Section 7, and conclude in Section 8.

2 OPTITRUST IN PRACTICE

This section presents three case studies. In Section 2.1, we consider a dot-product computation, which provides a minimal yet interesting example on which to present the features of OptiTrust. In Section 2.2, we reproduce manually written code from OpenCV, a very popular, optimized computer vision library. In Section 2.3, we reproduce an optimized implementation of matrix multiplication, similar to the one produced by TVM, an industrial compiler specialized for machine learning applications.

Through these case studies, we describe many aspects of OptiTrust, aiming to make the presentation self-contained and accessible without prior exposure to separation logic. As said in the introduction, OptiTrust currently idealizes numbers: we reason about values of type `int` and `float` as if they were values in $\mathbb{Z}$ and $\mathbb{R}$.

```
__DECL(reduce_sum, "int * (int → float) → float");
__AXIOM(reduce_sum_empty, "forall (f: int → float) →"
  "  0.f =. reduce_sum(0, f)");
__AXIOM(reduce_sum_add_right,
  "forall (n: int) (f: int → float) (_: n >= 0) →"
  "  reduce_sum(n, f) +. f(n) =. reduce_sum(n + 1, f)");
```

Fig. 1. Axiomatization of the reduction operator `reduce_sum`.

```
float dot(const float* a, const float* b, const int n) {
  __requires("A: int → float, B: int → float");
  __requires("n % 1024 = 0");
  __reads("a ⇝ Matrix1(n, A), b ⇝ Matrix1(n, B)");
  __ensures("_Res =. reduce_sum(n, fun j → A(j) *. B(j))");
  float s = 0.f;
  __ghost(rewrite_float_linear, "inside := fun v → &s ↦ v,"
    "by := reduce_sum_empty(fun j → A(j) *. B(j))");
  for (int i = 0; i < n; i++) {
    __spreserves("&s ↦ reduce_sum(i, fun j → A(j) *. B(j))");
    __ghost_begin(focusA, ro_matrix1_focus, "a, i");
    __ghost_begin(focusB, ro_matrix1_focus, "b, i");
    s += a[MINDEX1(n,i)] * b[MINDEX1(n,i)];
    __ghost_end(focusA);
    __ghost_end(focusB);
    __ghost(in_range_bounds, "i", "i_ge_0 <- lower_bound");
    __ghost(rewrite_float_linear, "inside := fun v → &s ↦ v,"
      "by := reduce_sum_add_right(i, fun j → A(j) *. B(j), i_ge_0)");
  }
  __ghost(eq_refl_float, "reduce_sum(n, fun j → A(j) *. B(j))");
  return s;
}
```

Fig. 2. Unoptimized input code for the dot product.

2.1 The Dot Product Case Study

Given two arrays described as A and B, the dot-product operation computes $\sum_{i\in[0,n)} A_i \cdot B_i$. We are here interested in optimizing the code under the assumption that n is a multiple of 1024 (e.g., assuming padding if needed on the input data). We next explain how to formally state the specification in OptiTrust, how to annotate an unoptimized implementation in order for verify its correctness using the OptiTrust typechecker, and how to write a transformation script achieving a naive parallelization of this implementation.

Specification of dot-product. To describe summations in OptiTrust specifications, we introduce a reduction operator, written `reduce_sum(n,f)`, for computing $\sum_{i\in[0,n)} f(i)$. Figure 1 shows our OptiTrust axiomatization of this operator. We follow a presentation close to that of the Rocq proof assistant. We use OCaml-style dot syntax such as `+.` for floating-point operations, as OptiTrust does not yet feature operator overloading for logical expressions. The **`__DECL`** statement declares the existence of the function `reduce_sum`. The two **`__AXIOM`** statements give its characteristic properties: $\sum_{i\in[0,0)} f(i) = 0$, and $\left(\sum_{i\in[0,n)} f(i)\right) + f(n) = \sum_{i\in[0,n+1)} f(i)$. We have manually translated these axioms into Rocq, where we gave them concrete implementation to ensure coherence. We leave it to future work to automate the extraction process to Rocq.

Specification in separation logic. Figure 2 shows how we program, specify and verify the dot-product function in OptiTrust. The code mixes C-style code with *annotations* that take the form of calls to functions whose name begins with a double-underscore. Each of these functions behaves, semantically, as a no-op.

The first line of Figure 2 gives the prototype of the function. The specification, a.k.a. *function contract*, follows. It consists of a description of the assumptions of the input arguments and input state that the function expects—the *preconditions*—as well as the guarantees it provides about the

output values and output state—the *postconditions*. Here, the input arguments and output values include not only the *proper* values that the function receives and returns, but also certain logical entities, called *ghost* variables, that are involved in the statement of the specifications.

The first line of the specification from Figure 2 quantifies two such *ghost arguments* by means of the **__requires** clause. The functions A and B, of type **int → float**, are used to describe the contents of the arrays associated with the pointers a and b. The value of these functions outside the domain of the array is irrelevant.[2] The second line asserts that n is a multiple of 1024.

The **__reads** clause that comes next involves *permissions* over the arguments a and b. The permission `a ⇝ Matrix1(n, A)` specifies in separation logic that a points to an array (a 1-dimensional matrix) of length n, allocated in memory, and whose cells are described by the function A. In other words, the memory location `a+i` contains the value `A(i)`, for any i such that $0 \le i < n$. In separation logic terminology, `Matrix1` is called a *representation predicate*: it relates a pointer, a logical description of the contents, and a piece of the memory state.

The clause **__reads**`("a ⇝ Matrix1(n, A), b ⇝ Matrix1(n, B)")` as a whole asserts that the arrays a and b are provided to the function in read-only mode. Interestingly, because both arrays are passed with a *read* permission, it is possible for a caller to provide the same array twice to the function. For example, one may call `dot(a,a)` for computing the square norm of a vector a. Further on, we will provide further details about the *aliasing* rule.

The last line of the specification includes an **__ensures** clause, which corresponds to a postcondition. It asserts that the result of the dot product function, named _Res, is equal to $\sum_{i \in [0,n)} A(i) \cdot B(i)$.

Macros for array accesses. In the internal language of OptiTrust, to facilitate the implementation of program transformations, we found it useful to syntactically keep track of the array sizes (or, more generally, of matrix dimensions) at every array access operation. Concretely, instead of writing just `a[i]`, we write `a[MINDEX1(n,i)]`, where n denotes the length of a. As we will see in the next case study, a 2-dimensional array access takes the form `a[MINDEX2(n,m,i,j)]`, and is interpreted as `a[i*m+j]`. Keeping track of all the dimensions involved can be useful for swapping dimensions, for example transforming `a[MINDEX2(n,m,i,j)]` into `a[MINDEX2(m,n,j,i)]`. For now, we require the OptiTrust user to provide dimensions everywhere, but presumably in the future most of the size arguments could be elaborated automatically.

Loop contracts. In the internal language of OptiTrust, every loop is equipped with a *loop contract*, describing how each iteration affects the content of memory. A loop contract may contain several clauses, each specifying the evolution of one or several parts of memory during the execution of an iteration. In the surface language in which the user provides the input code, permissions that remain the same inside and outside the loop may be omitted from contracts—they are automatically added if they are needed. For example, in Figure 2, the permission **__sreads**`("a ⇝ Matrix1(n, A), b ⇝ Matrix1(n, B)")` is omitted from the loop contract. The "s" in `sreads` means "shared", reflecting on the idea that all iterations *share* a read permission on the matrices a and b.

The contract of the loop that appears in Figure 2 describes the evolution of the contents of the variable s with respect to i. This clause, named **__spreserves**, asserts that, at the start of the i-th iteration, s contains the value $\sum_{j \in [0,i)} A(j) \cdot B(j)$. The leading "s" in **__spreserves** means "sequentially". It captures the idea that the clause corresponds to the *loop invariant* associated with the *sequential* loop over i. More complex loop contracts will appear in the next case studies.

[2]We here follow a design where arrays are modeled using total functions whose value is unspecified outside of the range of valid indices in the array. This range appears further on in the permission `a ⇝ Matrix1(n,A)`. Another possibility would be to quantify over a logical sequences of length exactly n. For the case study that we have considered, using functions decreased the amount of reasoning involving equalities, and saved the need for relying on additional mathematical libraries.

Besides loop contracts, the body of the function from Figure 2 contains a number of other annotations, which we describe next. Keep in mind that all these annotations have no impact on the semantics of the code, their only purpose is to guide the typechecking process. Typechecking verifies that, together, the code and its annotations constitute a valid separation logic derivation.

Ghosts for focus operations. All the lines that start with the prefix `__ghost` correspond to *ghost operations*, which are standard in separation logic. These annotations play a crucial role in changing the logical description of memory. More specifically, the *ghost focus* operations have the purpose of isolating a specific portion of memory out of a larger one: they are introduced by the keywords `__ghost_begin` and `__ghost_end`. In the present case study, the precondition of the function describes a read-only permission on the entire matrix `a`. To read the value stored at `a[i]` in memory, we need to argue that, from the permission on the entire matrix `a`, we can legitimately extract the permission on a single cell `a[i]`, provided that the array index `i` falls within the bounds. The `__ghost_begin` statement materializes this operation. It leaves not only the permission on the cell `a[i]`, but also a permission to revert the focus operation, that is, to merge the permission on `a[i]` with the permission on "all cells of `a` but the one at index `i`", so as to recover the original permission on the entire matrix. The `__ghost_end` statement materializes this reverse operation, which is called *unfocus*.

The name `focusA` introduced in the code materializes that the `__ghost_begin(focusA, ...)` and `__ghost_end(focusA)` statements go in pair. Tracking ghost pairs this way helps implementing transformations such as *loop fission* or *loop hoisting*, where both the focus and unfocus operations are handled synchronously. Likewise, a ghost pair named `focusB` is involved for focusing and defocusing on the cell `b[i]`.

In many situations, like in the case studies considered in the present paper, a fair number of the necessary *focus* and *unfocus* operations can be inferred automatically.[3] It remains essential, however, for the ghost pairs to be explicitly materialized in the annotated code before applying transformations. Indeed, if focus operations were to remain implicit, then after multiple transformations are applied onto the code, focus operations may become nontrivial to infer; and if focus operations cannot be inferred, the correctness of the optimized code can no longer be verified by means of typechecking.

In summary, focus operations, whether obtained from the programmer or by means of an elaboration phase, are materialized in the abstract syntax tree manipulated by OptiTrust, and are handled by every transformation.

Ghosts for rewrite operations. The purpose of the ghost operations mentioning `rewrite` in their name is to leverage equivalence theorems. The first one, which refers to the property `reduce_sum_empty`, is used to argue that the assignment `s = 0.f` initializes the reduction, in the sense that $\sum_{j\in[0,0)} A(j) \cdot B(j) = 0$.

The second one, which refers the property `reduce_sum_add_right`, is used to argue about the preservation of the loop invariant. Recall that the loop invariant asserts that the iterations before iteration `i` ensure that `s` stores $\sum_{j\in[0,i)} A(j) \cdot B(j)$. The `i`-th iteration of the loop body executes `s += A[i] * B[i]`. At the end of the loop body, we argue, using `reduce_sum_add_right`, that `s` now contains $\sum_{j\in[0,i+1)} A(j) \cdot B(j)$, thereby establishing the loop invariant for the next iteration.

The two aforementioned rewrite operations correspond to explicit proof steps. In such a simple case study, it is known from the state-of-the-art on program verification tools (e.g., Why3 [Filliâtre

[3]This possibility has already been demonstrated as part the ongoing Master project of Elian Morel.

```
float dot(const float* a, const float* b, const int n) {
  float s = 0.f;
  float* const t = (float*)malloc(exact_div(n, 1024) * sizeof(float));
  #pragma omp parallel for
  for (int bi = 0; bi < exact_div(n, 1024); bi++) {
    t[bi] = 0.f;
    for (int i = 0; i < 1024; i++) {
      t[bi] += a[1024 * bi + i] * b[1024 * bi + i];
    }
  }
  for (int bi = 0; bi < exact_div(n, 1024); bi++) {
    s += t[bi];
  }
  free(t);
  return s;
}
```

Fig. 3. Parallelized C code for the dot product.

and Paskevich 2013], or Viper [Müller et al. 2017]) that an off-the-shelf SMT prover could automatically establish the loop invariant. However, more complex programs may defeat the ability of even the best automated provers. Hence, user-provided rewrite operations can be necessary in general.

While we currently rely on the writing of explicit proof terms, we envision the integration of automated proofs to discharge simple proof obligations, as well as the integration of interactive proofs to discharge challenging proof obligations—based on our experience in program verification, we believe exploiting both is necessary to achieve a high productivity in general.

Ghosts for arithmetic facts. Let us comment on the two remaining ghost operations, which establish straightforward arithmetic facts.

The ghost operations involving `in_range_bounds` is used to extract inequalities from a range inclusion. The loop over `i` automatically gives us an assumption `in_range(i, range(0,n,1))`, expressed in terms of the generic Python-style range constructor `range(start,stop,step)`. To exploit `reduce_sum_add_right` as described earlier, we need to prove the side-condition `i >= 0`. The lemma `in_range_bounds` extracts this fact from the `in_range` property.

The ghost operation involving `eq_refl_float` is used to introduce an equality in such a way that the typechecker, upon the **`return`** `s` statement, can establish the claimed postcondition: `_Res = reduce_sum(n, fun j → A(j) *. B(j))`. The need for doing so stems from the fact that our typechecker currently does not integrate specific support for handling equality—in that respect, we have so far followed the design of the Rocq proof assistant.

Optimized code. Figure 3 shows the parallel C implementation that we are aiming to produce using OptiTrust for the dot-product function. This code allows up to `n/1024` threads to independently sum `1024` values. The corresponding results are stored in an array, named `t`, of size `n/1024`. When the threads complete, these partial sums are combined sequentially.[4]

Transformation script. Figure 4 shows our script for compiling the unoptimized annotated code from Figure 2 into the optimized C code from Figure 3. This script is written in the OCaml programming language. For the reader not familiar with OCaml, `f x y` denotes the call of `f` on the arguments `x` and `y`; the symbol `~` is used to provide optional (or named) arguments; `[x; y; z]` denotes a list; and the **`let-in`** construct introduces a variable or a function.

The transformation script from Figure 4 consists of a series of calls to functions from the OptiTrust library. Each transformation may depend on a number of arguments. By convention,

[4]For such a simple example, one might be tempted to rely on OpenMP's facility for parallel reductions. OptiTrust currently relies on OpenMP only for the scheduling of parallel iterations, not for other features. This keeps OptiTrust's internal semantics simple, while providing more control over how to parallelize the reduction.

```
Loop.tile (int 1024) ~index:"bi" ~bound:TileDivides [cFor "i"];
Variable.local_name ~var:"s" ~local_var:"t" [tSpanSeq [cForBody "bi"]];
let factor = trm_get (trm_find_var "s" []) in
Accesses.shift_var ~simpl:Arith.gather_rec ~inv:true ~factor [cFor "bi"; cVarDef "t"];
Loop.hoist [cVarDef "t"];
Loop.fission [tBefore; cFor "bi"; cWriteVar "s"];
Loop.parallel [cFor "bi" ~body:[cFor "i"]];
UnverifiedCleanup.std();
```

Fig. 4. Optimization script for the dot product.

```
for (int i = 0; i < 32; i++) {
 s += a[MINDEX1(n, bi * 32 + i)] * b[MINDEX1(n, bi * 32 + i)];
}
```

```
 float t = s;
 for (int i = 0; i < 32; i++) {
  t += a[MINDEX1(n, bi * 32 + i)] * b[MINDEX1(n, bi * 32 + i)];
 }
 s = t;
```

Fig. 5. Code-only diff for the `Variable.local_name` step from Figure 4, i.e. not showing annotations.

```
  __ghost(rewrite_linear,
     "inside := fun (i: int) -> &s ~~> reduce_sum(i, fun j -> A(j) * "
     "B(j)), by := plus_zero_intro(bi * 32)");
  for (int i = 0; i < 32; i++) {

  __spreserves("&s ~~> reduce_sum(bi * 32 + i, fun j -> A(j) * B(j))");

   s += a[MINDEX1(n, bi * 32 + i)] * b[MINDEX1(n, bi * 32 + i)];

  __ghost(rewrite_float_linear,
     "inside := fun v -> &s ~~> v, by := reduce_sum_add_right(bi * 32 "
     "+ i, fun j -> A(j) * B(j), i_ge_0)");
  __ghost(rewrite_linear,
     "inside := fun (i: int) -> &s ~~> reduce_sum(i, fun j -> A(j) * "
     "B(j)), by := add_assoc_right(bi * 32, i, 1)");
 }
 __ghost(rewrite_linear,
    "inside := fun (i: int) -> &s ~~> reduce_sum(i, fun j -> A(j) * "
    "B(j)), by := mul_add_factor(bi, 32)");
```

```
  __ghost(rewrite_linear,
     "inside := fun (i: int) -> &t ~~> reduce_sum(i, fun j -> A(j) * "
     "B(j)), by := plus_zero_intro(bi * 32)");
  for (int i = 0; i < 32; i++) {

  __spreserves("&t ~~> reduce_sum(bi * 32 + i, fun j -> A(j) * B(j))");

   t += a[MINDEX1(n, bi * 32 + i)] * b[MINDEX1(n, bi * 32 + i)];

  __ghost(rewrite_float_linear,
     "inside := fun v -> &t ~~> v, by := reduce_sum_add_right(bi * 32 "
     "+ i, fun j -> A(j) * B(j), i_ge_0)");
  __ghost(rewrite_linear,
     "inside := fun (i: int) -> &t ~~> reduce_sum(i, fun j -> A(j) * "
     "B(j)), by := add_assoc_right(bi * 32, i, 1)");
 }
 __ghost(rewrite_linear,
    "inside := fun (i: int) -> &t ~~> reduce_sum(i, fun j -> A(j) * "
    "B(j)), by := mul_add_factor(bi, 32)");
 s = t;
```

Fig. 6. Proof-carrying diff for the `Variable.local_name` step from Figure 4, i.e. for the same step as in Figure 5 but showing ghost operations and loop contracts.

the last argument of a transformation always denotes a *target*, indicating where in the code the transformation should be applied. OptiTrust provides an expressive target mechanism for describing, in a concise and robust manner, one or several code locations. For example, the target `[cFor "i"]` refers to the loop over `i`. The target `[cFor "bi"; cVarDef "t"]` refers to the definition of `t` inside the loop over `bi`. The target `[cFor "bi" ~body:[cFor "i"]]` refers to the loop over `bi` whose body contains a loop over `i`. This latter target is used at a time where the code contains two distinct loops over `bi`. More advanced examples of targets will be presented in the next case study.

The steps of the transformation script can be described at a high level as follows. `Loop.tile` introduces an inner loop over 1024 consecutive values to sum. `Variable.local_name` declares, inside the outer loop, a variable `t` that "replaces" `s` over that scope: an assignment `t = s` is inserted at the start of the body of the loop, the assignment `s = t` is inserted at the end of the body, and the accumulation `s += a[i] * b[i]` becomes `t += a[i] * b[i]` (we here omit the `MINDEX1` for readability).

Then, `Accesses.shift_var` translates the contents of `t` by the value that `s` within that same scope. The assignments `t = s` and `s = t` become `t = 0` and `s += t`. `Loop.hoist` replaces `t`, a variable defined inside the loop over `bi`, with an array defined outside that loop, with occurrences of `t` being turned into `t[bi]`. `Loop.fission` introduces the two outer loops visible in Figure 3: one to create the partial sums of 1024 consecutive values, and one to add up the partial sums. `Loop.parallel` marks the first loop over the block as parallel. `UnverifiedCleanup.std` triggers C code generation, eliminating all dependencies on the OptiTrust header file, and performing obvious arithmetic simplifications—typically to simplify multidimensional index computations revealed after inlining `MINDEX` functions.

```
37  for (int bi = 0; bi < exact_div(n, 32); bi++) {
39    __spreserves("&s ~~> reduce_sum(bi * 32, fun j -> A(j) * B(j))");
40    __sreads("a ~> Matrix1(n, A)");
41    __sreads("b ~> Matrix1(n, B)");
42    __xpreserves("&t[MINDEX1(exact_div(n, 32), bi)] ~> UninitCell");
```

```
37  for (int bi = 0; bi < exact_div(n, 32); bi++) {
39    __sreads("a ~> Matrix1(n, A)");
40    __sreads("b ~> Matrix1(n, B)");
41    __xwrites(
42      "&t[MINDEX1(exact_div(n, 32), bi)] ~~> reduce_sum((bi + 1) * 32, fun j "
43      "-> A(j) * B(j)) - reduce_sum(bi * 32, fun j -> A(j) * B(j))");
```

```
113 }
114 for (int bi = 0; bi < exact_div(n, 32); bi++) {
116   __spreserves("&s ~~> reduce_sum(bi * 32, fun j -> A(j) * B(j))");
117   __xreads(
118     "&t[MINDEX1(exact_div(n, 32), bi)] ~~> reduce_sum((bi + 1) * 32, fun j "
119     "-> A(j) * B(j)) - reduce_sum(bi * 32, fun j -> A(j) * B(j))");
```

Fig. 7. Proof-carrying diff for the `Loop.fission` step from Figure 4.

Interactive visualization. Transformation scripts are meant to be developed interactively, similarly as when conducting interactive proofs in, e.g., the Rocq proof assistant. This makes it easier to visualize the effects of the script. With the cursor on a line of the transformation script, the OptiTrust user may press a key shortcut for visualizing the *diff* associated with the transformation on that line. For example, the user might visualize the effect on code from the `Variable.local_name` step, and obtain the diff shown in Figure 5. More interestingly, the user might also produce a diff that includes proof annotations, as shown in Figure 6. This diff demonstrates how `Variable.local_name` transparently dealt with rewriting logical expressions inside proof annotations. The mechanisms at play will be explained in Section 5.

Some transformations on proof-carrying code are more advanced. For example, Figure 7 shows the effects of the `Loop.fission` step, which automatically generates loop invariants. First, the invariant on the contents of `s` is isolated to the second loop. Second, while the original loop only required uninitialized storage for the array `t`, the first generated loop now initializes that storage, while the second one reads what the first one wrote in that storage.

Additionally, OptiTrust can produce a complete *execution trace* in the form of an interactive tree, embedded in a webpage. 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.[5] In particular, one can visualize in a trace the annotated code just before the final step that generates the C code and erases annotations. The typechecking of this annotated code justifies the correctness of all the transformations performed in the script before the final step.

2.2 The OpenCV Row-Based Blur Case Study

Blur implementation considered. In image processing, a *blur* is typically used to remove noise and smoothen images. A two-dimensional blur can be decomposed into a *column-based blur*, a *row-based blur*, and (optionally) the application of a normalization pass. Our case study focuses on a *row-based blur* function, as implemented in the state-of-the-art OpenCV library [Bradski et al. 2000].

The output of this function consists of a single-row image, stored in an array named `D`, made of `n` pixels. The input consists of a single-row image, stored in an array named `S`, made of `n+w-1` pixels, where the parameter `w` corresponds to the width of the blur. Each pixel is encoded on `cn` integers, one per color channel. The code accommodates any value of `cn`, but practical values include `cn=1` for grayscale, `cn=3` for RGB, and `cn=4` for RGBA.

Intuitively, the output pixel D_i is computed as the sum of the values of the input pixels from S_i to S_{i+w-1}. This sum is computed independently for each of the `cn` color channels. Thus, technically, the output data is specified by the formula: $D_{i,c} = \sum_{k \in [i, i+w)} S_{k,c}$, where $0 \le i <$ `n` and $0 \le c <$ `cn`.

[5] The traces showing the diff for every major step of the script can be browsed online at: https://www.chargueraud.org/softs/optitrust/traces/index.html. Due to their large size, the traces that include all the substeps are only available by constructing them using a local installation of OptiTrust.

```
void rowSum(const int w, const int* s, int* d, const int n, const int cn) {
  __requires("w >= 0, n >= 1, cn >= 0");
  __requires("S: int * int → int");
  __reads("s ⇝ Matrix2(n+w-1, cn, S)");
  __writes("d ⇝ Matrix2(n, cn, fun (i c: int) → reduce_int_sum(i, i+w, fun k → S(k,c)))");

  for (int i = 0; i < n; i++) { // for each destination pixel
    __xwrites("for c in 0..cn → &d[MINDEX2(n, cn, i, c)] ↦
      reduce_int_sum(i, i+w, fun k → S(k,c))");

    for (int c = 0; c < cn; c++) { // for each color channel
      __xwrites("&d[MINDEX2(n, cn, i, c)] ↦ reduce_int_sum(i, i+w, fun k → S(k,c))");

      int sum = 0;
      __ghost(rewrite_linear, "inside := fun v → &sum ↦ v,"
        "by := reduce_int_sum_empty(i, fun k → S(k,c))");

      for (int k = i; k < i+w; k++) { // for each source pixel within the blur range
        __spreserves("&sum ↦ reduce_int_sum(i, k, fun k0 → S(k0,c))");
        __ghost(assume, "in_range(k, 0..n+w-1)");
        __ghost_begin(focus, ro_matrix2_focus, "s, k, c");
        sum += s[MINDEX2(n+w-1, cn, k, c)];
        __ghost_end(focus);
        __ghost(in_range_bounds, "k, i", "k_ge_i <- lower_bound");
        __ghost(rewrite_linear, "inside := fun v → &sum ↦ v,"
          "by := reduce_int_sum_add_right(i, k, fun k → S(k,c), k_ge_i, k + 1, kp1)");
      }
      d[MINDEX2(n, cn, i, c)] = sum;
} } }
```

Fig. 8. Unoptimized OptiC code for the OpenCV case study.

Code and specification. Figure 8 shows an unoptimized implementation of the row-blur function, decorated with the necessary annotations to ensure typechecking in OptiTrust. Observe that the pointer on matrices are described as arguments of type `int*`. Indeed, in OptiTrust, we represent multidimensional arrays using a flat array representation. This representation is commonplace in high-performance code, as it allows performing simplifications in array accesses. In the code and its annotations, we use the macro `MINDEX2` to convert matrix coordinates into flat indices. For example, `s[MINDEX2(n+w-1,cn,k,c)]` describes the index of the cell at coordinate $(\mathtt{k}, \mathtt{c})$, within a matrix of dimensions $(\mathtt{n+w-1}) \times \mathtt{cn}$. This index is computed by the formula `k*cn+c`.

The next 4 lines give the specification of the function. The first line state arithmetic inequalities on the integer parameters of the function. The second line quantifies a logical function S of two arguments, meant to describe the content of the input matrix. The third line mentions a read permission over `s ⇝ Matrix2(n+w-1,cn,S)`. This asserts that s is a pointer to a matrix of dimension $(\mathtt{n+w-1}) \times \mathtt{cn}$ such that the value of each memory cell is modeled by $S(k, c)$. Implicitly, the representation predicate `Matrix2` specifies a flat memory layout of the matrix, meaning that the memory cell at location `s+k*cn+c` contains the value $S(k, c)$.

The last specification line, `__writes(d ⇝ Matrix2(n,cn,...))`, asserts that the function writes its results in a matrix corresponding to the argument `d`. The matrix `d` should be allocated in memory prior to the function call, and have dimensions $\mathtt{n} \times \mathtt{cn}$. Importantly, the separation logic interpretation of the specification guarantees that the output array `d` does not overlap in memory with the input array `s`. Indeed, separation logic rules out the possibility of *aliasing* between multiple arrays unless all these are all covered by read permissions.

Posterior to the call, a cell at coordinate (i, c) is specified to contain the value $\sum_{j \in [i, i+w)} S(j, c)$. This formula is expressed by means of the logical operation `reduce_int_sum`, which is similar to

`reduce_sum` presented earlier on, only accumulating integer values. We leave it to future work to support a polymorphic reduction function.

Grammar of contracts. In general, a function contract includes 4 kind of clauses. **`__requires`** describes pure preconditions, i.e., input ghost arguments, and propositions giving hypotheses about the input. **`__consumes`** describes the linear preconditions, i.e., the separation logic predicates describing the input state that the function operates on. Symmetrically, **`__ensures`** describes pure postconditions, i.e., output ghost arguments, and propositions characterizing the output. **`__produces`** describes the linear postconditions, i.e., the separation logic predicates describing the output state of the function. Other clauses are derived as syntactic sugar. In particular, **`__reads`** is a shorthand for consuming and producing a read-only limited version of a separation logic predicate. **`__writes`** is a shorthand for consuming uninitialized memory cells, and producing an output state in which these cells are set to specific values.

As said, from the perspective of the semantics, ghost operations are no-ops. From the perspective of the OptiTrust typechecker, however, ghost operations are viewed as conventional function calls. More precisely, a ghost function is viewed as a void function with zero arguments, and equipped with a contract expressed using the exact same grammar of clauses as for program functions.

Loop contracts and ghost operations. Besides the code and its specification, Figure 8 contains loop contracts, focus operations for reading in the input matrix, as well as ghost operations to deal with the reduction at play in a similar way as in the previous case study. There is one new feature to introduce. The loop over `i` is annotated with a clause of the form: **`__xwrites`**`("for c in 0..cn → &d[MINDEX2(n,cn,i,c)] ↦ D(i,c)")`. This clause indicates that the `i`-th iteration of that outer loop requires exclusive access to all the cells in the `i`-th row of the destination matrix `d`. The "x" prefix in **`__xwrites`** stands for "exclusive".

Further in the paper, this same resource will be written as: $\bigstar_{c \in 0..cn}$ `&d[MINDEX2(`n, cn, i, c`)]` $\rightsquigarrow D(i,c)$, where the star symbol is called *iterated separating conjunction* in separation logic. This star symbol is also used to *define* the `Matrix2` predicate: `d` $\rightsquigarrow$ `Matrix2(n,cn,D)` stands for the predicate $\bigstar_{i \in 0..n} \bigstar_{c \in 0..cn}$ `&d[MINDEX2(`n, cn, i, c`)]` $\rightsquigarrow D(i,c)$. The purpose of a focus operation is to extract one particular item out of a star, or out of a group of nested stars. For example, the focus operation surrounding the write into `sum` in Figure 8, named `ro_matrix2_focus`, allows us to isolate the memory cell that contains the value $S(k,c)$ from the matrix described by S.

The loop over `c` also involves a **`__xwrites`** annotation. This annotation asserts that each iteration independently operates on the cell at location `&d[MINDEX2(n,cn,i,c)]`. When all the writable resources involved in a loop can be partitioned in such a way across the iterations, the loop is *parallelizable*. Here, for instance, the loop on `i` and the loop on `c` could run in parallel without any data race. The loop on `k`, however, cannot be parallelized because the resource on the cell `sum` is updated by all iterations, sequentially.

Optimized code. The code from Figure 9 corresponds to the optimized code that we produce using OptiTrust. This code is structured just like the handwritten OpenCV library code, which may be viewed online.[6]

This optimized implementation is *multi-versioned* code, with dedicated execution paths for handling specific values of the parameters. The branches `w == 3` and `w == 5` correspond to values

[6]https://github.com/opencv/opencv/blob/4.10.0/modules/imgproc/src/box_filter.simd.hpp#L75: The OpenCV code is implemented as a class with the types `T` and `ST` as template arguments, whereas for the moment our code refers to fixed, idealized integer types; we look forward to add support for polymorphism and fixed-size integers. The OpenCV code also traverses certain arrays by incrementing pointers, whereas we use explicit array indexing everywhere. In general, this choice is not performance critical and we leave OptiTrust support for pointer shifting to future work.

```
void rowSum(int n, int cn, int w,
            int* s, int* d) {
  if (w == 3) {
    for (int ic = 0; ic < cn * n; ic++) {
      d[ic] = s[ic]
            + s[cn + ic]
            + s[2 * cn + ic];
    }
  } else if (w == 5) {
    for (int ic = 0; ic < cn * n; ic++) {
      d[ic] = s[ic]
            + s[cn + ic]
            + s[2 * cn + ic]
            + s[3 * cn + ic]
            + s[4 * cn + ic];
    }
  } else if (cn == 1) {
    int sum = 0;
    for (int k = 0; k < w; i++) {
      sum += s[k];
    }
    d[0] = sum;
    for (int i = 0; i < n - 1; i++) {
      sum += s[i + w] - s[i];
      d[i + 1] = sum;
    }
  } else if (cn == 3) {
    int sum0 = 0;
    int sum1 = 0;
    int sum2 = 0;
    for (int k = 0; k < 3 * w; k += 3) {
      sum0 += s[k];
      sum1 += s[k + 1];
      sum2 += s[k + 2];
    }
    d[0] = sum0;
    d[1] = sum1;
    d[2] = sum2;
    for (int i = 0; i < 3 * n - 3; i += 3) {
      sum0 += s[3 * w + i] - s[i];
      sum1 += s[3 * w + i + 1] - s[i + 1];
      sum2 += s[3 * w + i + 2] - s[i + 2];
      d[i + 3] = sum0;
      d[i + 4] = sum1;
      d[i + 5] = sum2;
    }
  } else if (cn == 4) {
    // [...] similar to cn == 3
  } else {
    for (int c = 0; c < cn; c++) {
      int sum = 0;
      for (int k = 0; k < cn * w; k += cn) {
        sum += s[c + k];
      }
      d[c] = sum;
      for (int i = c; i < cn * n - cn + c;
           i += cn) {
        sum += s[cn * w + i] - s[i];
        d[cn + i] = sum;
  } } } }
```

Fig. 9. 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.

of the width that are commonly provided by library users. For these small constant values of `w`, the inner loop on `k` from Figure 9 is unfolded. Otherwise, the loop on `k` is not unfolded and a standard algorithmic optimization called *sliding window* is applied. Note that Halide [Ragan-Kelley et al. 2013], the state-of-the-art specialized compiler for image processing, does not support the introduction of sliding windows, and that the developers of Halide do not plan to lift this limitation.[7]

The branch of the code that uses the sliding window optimization is then further specialized for commonly used parameters: `cn == 1`, `cn == 3`, and `cn == 4`. For these small constant values of `cn`, the outer loop on `c` is unfolded, then the multiple occurrences of the loop on `i` that result from this unfolding are fused into a single loop. The final **`else`** branch in the code from Figure 9 corresponds to the generic implementation. Moreover, in the last three branches, the loops are reindexed to augment the counter `i` by steps of `cn`, thereby saving multiplication operations.

The last `UnverifiedCleanup.std` transformation step shown in Figure 9 triggers the generation of C code. In the OpenCV implementation of `rowSum`, the input pixels in `s` are encoded on `cn` integers of type **`int`**, whereas the output pixels in `d` are encoded on `cn` integers of type **`int`**. As said in Section 1.3, OptiTrust currently supports only idealized integers, and does not yet provide practical means of checking the absence of overflows. Thus, we simply indicate in the script the precision at which we wish the various integer data structures to be compiled. This specialization of integer representations constitutes, in the current state of OptiTrust, an unverified step. Compared with the manually written code of OpenCV, for which nothing is verified, the fact that our code is

[7]Halide does not support sliding windows for reasons explained on: https://github.com/halide/Halide/issues/180. Hence, the programmer either needs to manually refine the code to introduce the sliding window before scheduling; or needs to exploit other transformation tools specialized in sliding window optimizations [Chaurasia et al. 2015; Kanetaka et al. 2024].

```
Specialize.variable_multi ~mark_then:fst ~mark_else:"anyw"
  ["w", int 3; "w", int 5] [cFunBody "rowSum"; cFor "i"];
Reduce.unroll [nbMulti; cMark "w"; cFor "k"];
Loop.collapse [nbMulti; cMark "w"; cFor "i"];

Loop.swap [nbMulti; cMark "anyw"; cFor "i"];
Reduce.first_then_slide ~mark_alloc:"acc" [nbMulti; cMark "anyw"; cFor "i"];
Variable.elim_reuse [nbMulti; cMark "acc"];
Loop.shift_range (StartAtZero) ~simpl:no_simpl [nbMulti; cMark "anyw"; cFors ["k"; "i"]];
Loop.scale_range ~factor:(trm_find_var "cn" []) ~simpl:no_simpl
  [nbMulti; cMark "anyw"; cFors ["k"; "i"]];

Specialize.variable_multi ~mark_then:fst ~mark_else:"anycn" ~simpl:no_simpl
  ["cn", int 1; "cn", int 3; "cn", int 4] [cMark "anyw"; cFor "c"];
Loop.unroll [nbMulti; cMark "cn"; cFor "c"];
Target.foreach [nbMulti; cMark "cn"] (fun c →
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "i"];
  Instr.gather_targets [c; cStrict; cArrayWrite "d"];
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "k"];
  Instr.gather_targets [c; cFor "i"; cArrayWrite "d"]; );
Loop.shift_range ~simpl:no_simpl (ShiftBy (trm_find_var "c" [cMark "anycn"]))
  [cMark "anycn"; cFor "i"];
UnverifiedCleanup.std ~int_size:[(typ_u8, "s"), (typ_u16,  "d\\|sum.*")] ();
```

Fig. 10. Optimization script for the OpenCV case study.

proved correct except for potential overflows should, we argue, already be viewed as a significant improvement.

Optimization script. Figure 10 presents our optimization script. Let us first comment on the targets, then on the transformations.

As said earlier, the *targets* that appear as last argument of each transformation indicate where in the code the transformations should be applied. Let us comment on additional target features. The constraint `cMark "anyw"` requires visiting an AST node that carries the mark `"anyw"`. Such marks are introduced by transformations, on demand of the programmer. The modifier `tBefore` allows targeting the interstice before an instruction. The modifier `nbMulti` indicates that the programmer expects to find not one but multiple AST nodes that match this target. In general, if the transformation passed to a target refers to multiple locations, the transformation is applied in turns at each location. Our implementation relies on marks to avoid path invalidation issues, e.g., when applying a change at the first targeted location invalidates the coordinates of the second target location. It is also possible to explicitly iterate over several code locations using the construct `Target.foreach`, visible near the end of Figure 10.

The key steps of the optimization script for `rowSum` are as follows. `Specialize.variable_multi` introduces a cascade of if-statements for testing specific variable values. `Loop.unroll` unrolls a loop with a statically known number of iterations. `Loop.collapse` takes two nested loops and replaces them with a single loop that iterates over the product space. `Loop.swap` takes two nested loops and swaps them. `Reduce.first_then_slide` introduces a sliding window optimization on a map-reduce computation. `Variable.elim_reuse` takes a variable defined to copy another, and eliminates that definition, reusing the storage from the copied variable. `Loop.shift_range` and `Loop.scale_range` allow altering the iteration range of a loop. `Loop.fusion_targets` fuses targeted loops into a single one. `Instr.gather_targets` reorders instructions in a sequence to make the targeted instructions consecutive. The reader should keep in mind that each of these transformations updates not only the code but also all the ghost operations and loop contracts.

Conclusion. This case study demonstrates how OptiTrust can be used to interactively optimize code that was previously manually optimized due to a gap in the compiler landscape. OptiTrust

allows chaining transformations that operate on high-level reduction algorithms, mid-level structured control flow, as well as relatively low-level code fine-tuning that includes getting exactly the desired index computations. Without OptiTrust, manual optimization typically involves handwriting, testing and debugging optimized code, a time consuming process that provides no formal guarantees and allows little automation.

2.3 The TVM Matrix-Multiply Case Study

Matrix-multiply implementation considered. TVM [Chen et al. 2018] is a state-of-the-art, industrial-strength, semi-automatic compiler for machine learning. The tutorial from TVM v0.19 presents an optimization script[8] (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 multiplication.[9] The aim of this third case study is to demonstrate the ability of OptiTrust to produce code that matches the performance delivered by specialized compilers such as TVM. We show that we are able to generate proof-carrying code that features the exact same optimization patterns as in the TVM tutorial, and benchmark the resulting C code to confirm its performance.

Unoptimized code. Let us first specify that the function considered computes a matrix product. To that end, we introduce the definition of the corresponding logical operation, written `matmul(A, B, p)`, which asserts that the result C is such that: $C_{i,j} = \sum_{k \in [0,p)} A_{i,k} \cdot B_{k,j}$. Here, p denotes the common dimension to the matrices `A` and `B`.[10] This operation is defined in terms of `reduce_sum`, which was axiomatized in Figure 1, as follows.

```
__DEF(matmul, "fun (A B: int * int → float) (p: int) →
  fun (i j: int) → reduce_sum(p, fun k → A(i, k) *. B(k, j))");
```

The **`__reads`** and **`__writes`** clauses at the top of Figure 11 specify that the function takes as argument two pointers `a` and `b` describing the input matrices, as well as a third pointer to a third matrix where to write the output. Here, `a` and `b` could be aliased (e.g., to square a matrix), but the matrix `c` is assumed to not overlap with `a` and `b` in memory.

The remaining of Figure 11 shows an unoptimized implementation of matrix-multiply. The ingredients are similar as in the previous case study: **`__xwrites`** in the loop contracts, ghost focus operations for reading in the two input matrices, as well as ghost rewrite operations to deal with the reduction. Here again, some annotations could be inferred automatically with additional tooling.

Interestingly, our unoptimized code corresponds to a straightforward imperative implementation of matrix multiply. This contrasts with the TVM input, which requires the user to manually modify the reference implementation by introducing a per-block transposed matrix.

[8] https://github.com/apache/tvm/blob/v0.19.0/gallery/how_to/optimize_operators/opt_gemm.py. Note that the TVM scheduling API has been completely refactored after v0.19, and that we do not attempt to compare to the newer TVM design.

[9] 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 the MKL provides optimized implementation for a fixed set of functions, whereas 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.

[10] The need for the argument `p` in `matmul(A, B, p)` stems from our decision to use functions to model the values stored in the matrices. If we were to model the contents of the matrices using mathematical matrices that carry their size information, then the predicate would take the form `matmul(A, B)` and the precondition **`__reads`**`("a ⇝ Matrix2(m, p, A), b ⇝ Matrix2(p, n, B)")` would have be expressed instead as **`__reads`**`("a ⇝ Matrix2(A), b ⇝ Matrix2(B)")` together with a clause of the form **`__requires`**`("A.dim_y = B.dim_x")`. We might try to support this alternative style in the future, after improving support for reasoning about equalities.

```
void mm(float* c, float* a, float* b, int m, int n, int p) {
  __requires("A: int * int → float, B: int * int → float");
  __reads("a ⇝ Matrix2(m, p, A), b ⇝ Matrix2(p, n, B)");
  __writes("c ⇝ Matrix2(m, n, matmul(A, B, p))");

  for (int i = 0; i < m; i++) {
    __xwrites("for j in 0..n → &c[MINDEX2(m, n, i, j)] ↦ matmul(A, B, p)(i, j)");

    for (int j = 0; j < n; j++) {
      __xwrites("&c[MINDEX2(m, n, i, j)] ↦ matmul(A, B, p)(i, j)");

      float sum = 0.f;
      __ghost(rewrite_float_linear, "inside := fun v → &sum ↦ v,"
        "by := reduce_sum_empty(fun k → A(i, k) *. B(k, j))");
      for (int k = 0; k < p; k++) {
        __spreserves("&sum ↦ reduce_sum(k, fun k0 → A(i, k0) *. B(k0, j))");

        __ghost_begin(focusA, ro_matrix2_focus, "a, i, k");
        __ghost_begin(focusB, ro_matrix2_focus, "b, k, j");
        sum += a[MINDEX2(m, p, i, k)] * b[MINDEX2(p, n, k, j)];
        __ghost_end(focusA);
        __ghost_end(focusB);

        __ghost(in_range_bounds, "k", "k_ge_0 <- lower_bound");
        __ghost(rewrite_float_linear, "inside := fun v → &sum ↦ v,"
          "by := reduce_sum_add_right(k, fun k → A(i, k) *. B(k, j), k_ge_0)");
      }

      c[MINDEX2(m, n, i, j)] = sum;
} } }
```

Fig. 11. Unoptimized OptiC code for matrix multiplication.

Optimized code. As said, our aim is to produce an output equivalent to that of TVM. Because TVM output code is expressed not as C code but directly in the intermediate representation of LLVM, we manually inspected the TVM schedule, intermediate representation, and LLVM IR output to infer what C code we should generate. The code we produce using OptiTrust is shown in Figure 12. Compared with the naive code from Figure 11, the optimized code integrates 8 key optimizations:

(1) The body of the generic matrix multiplication function `mm` is specialized to the size `1024`.
(2) An auxiliary matrix named `pB` is allocated to store the coefficients of the matrix `B` in a different order. Intuitively, `pB` corresponds to a per-block transposed representation of `B`. The introduction of this auxiliary matrix induces a cost for the initial copy, but then greatly improves the memory access patterns.
(3) 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.
(4) Results are not accumulated into a scalar accumulator, but instead into a stack-allocated array named `sum` of size 32×32 that contains all scalar accumulators for a block.
(5) Around the inner vectorized loops, the locally relevant row of `sum` is promoted to a smaller array `s` that can be mapped onto a few 256-bit vector registers. On every iteration of the loop over `i`, two `memcpy` operations are used for synchronizing `s` with `sum`.
(6) 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.
(7) The 4 loops tagged as `#pragma omp simd` come from unrolling a loop, this increases instruction-level parallelism without relying on the C compiler heuristics.
(8) The outer loops are marked as parallel, via the `#pragma omp parallel for` directive.

```
void mm1024(float* A, float* B, float* C) {
  float* pB = (float*)malloc(1048576 * sizeof(float));
  #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[32 * bj + 4096 * bk + 1024 * k + j];
  } } } }
  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* sum = (float*)malloc(1024 * sizeof(float));
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          sum[32 * i + j] = 0.f;
      } }
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(&s[0], &sum[32 * i], 32 * sizeof(float));
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[32768 * bi + 4 * bk + 1024 * i]
                  * pB[32768 * bj + 128 * bk + j];
          }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[32768 * bi + 4 * bk + 1024 * i + 1]
                  * pB[32768 * bj + 128 * bk + j + 32];
          }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[32768 * bi + 4 * bk + 1024 * i + 2]
                  * pB[32768 * bj + 128 * bk + j + 64];
          }
          #pragma omp simd
          for (int j = 0; j < 32; j++) {
            s[j] += A[32768 * bi + 4 * bk + 1024 * i + 3]
                  * pB[32768 * bj + 128 * bk + j + 96];
          }
          memcpy(&sum[32 * i], &s[0], 32 * sizeof(float));
      } }
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          C[32768 * bi + 32 * bj + 1024 * i + j] = sum[32 * i + j];
      } }
      free(sum);
  } }
  free(pB);
}
```

Fig. 12. Our optimized code for the matrix-multiply case study. This code features the same optimization patterns as the reference output of TVM.

Again, this particular set of optimizations corresponds exactly to the ones performed in the reference TVM tutorial. Our goal was to demonstrate that OptiTrust puts the control in the end of the user, allowing to apply a desired set of optimizations, not to come up with new optimizations.

Appendix A presents an excerpt of our optimized code for matrix multiplication with full functional correctness annotations, that is, just before the final extraction to C code.

```
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"];
Loop.simd [cFor ~body:[cPlusEq ()] "j"];
Loop.parallel [cFunBody "mm1024"; cStrict; cFor ""];
Loop.unroll [cFor ~body:[cPlusEq ()] "k"];
UnverifiedCleanup.std ();
```

Fig. 13. Optimization script for the matrix-multiply case study.

Optimization script. Figure 13 shows our optimization script, which consists of only 10 lines of transformations. Internally, though, the high-level transformations mentioned in the script trigger the application of 55 low-level transformations that directly alter the program code, and of 61 transformations that refine only the annotations. An illustrative example is the call to `Loop.reorder_at` on Line 4 of Figure 13. 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 the local variable named `sum` in Figure 11 into the 2D-array named `sum` in Figure 12.

Conclusion. In summary, the execution of the transformation script from Figure 13 compiles the separation logic annotated code from Figure 11 into the highly optimized C code from Figure 12, which implements exactly the same optimizations as the TVM output. Note that, the per-block transposition (which corresponds to variable `pB`) is not backed into our input code. Moreover, unlike with TVM, our transformed code comes with a separation logic derivation establishing its correctness with respect to the logical specification of matrix-multiplication. Our output C code delivers similar performance as that produced by TVM, up to statistical noise in the execution time. Plots of the distribution of execution times may be found in the appendix (Appendix B).

3 OPTITRUST'S INTERNAL AST

OptiTrust's internal language, called Optiλ, consists of an imperative λ-calculus that includes constructs for loops and for pointer arithmetic. All the transformations are applied on Optiλ.

For the end-user, OptiTrust exposes a surface language, called OptiC, in which the case studies were written. OptiC features C-style assignments, with mutable local variables and *l-values*. The motivation is to present a syntax familiar to the HPC programmer. OptiTrust provides both an encoding from OptiC to Optiλ, and a *decoding* from Optiλ back to OptiC. This way, we are able to report to the end-user the effect of a transformation in terms of a *diff* expressed between two pieces of OptiC code. To minimize the changes in the *diff*, the encoding translation attaches annotations on certain subterms; these annotations can be exploited when computing the reciprocal translation.

The use of a λ-calculus as internal language considerably helps to tame the complexity of the design and implementation of the proof rules and of the code transformations. The use of an intermediate language with simpler semantics is commonplace, both in the domain of compilation and in the domain of program verification. For example, Why3 [Filliâtre and Paskevich 2013] serves as intermediate verification language for C, Java, and Ada programs; and Viper [Müller et al. 2017] serves as intermediate verification language for Java, Rust, Go, OpenCL, etc. Although intermediate

R	$\ni$	$\mid$	$\mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$	range for range-based **for** loops
π	$\ni$	$\mid$	$\mathbf{seq} \mid \mathbf{par}$	execution mode for range-based **for** loops
r	$\ni$	$\mid$	$\varnothing \mid x$	result of a sequence
t	$\ni$	$\mid$	x	variables
		$\mid$	$b \mid n$	boolean values, and number values
		$\mid$	$\{t_1; ...; t_n; r\}$	sequence with result r
		$\mid$	$\mathbf{let}\ x = t$	variable definition
		$\mid$	$\mathbf{fun}(a_1, ..., a_n) \mapsto t$	function definition
		$\mid$	$t_0(t_1, ..., t_n)$	function call
		$\mid$	$\mathbf{for}^{\pi}\ (i \in R)\ t$	possibly parallel, range-based **for** loop
		$\mid$	$\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$	conditional
		$\mid$	$t_1 \boxplus t_2$	address computation

Fig. 14. Grammar of Optiλ, the internal λ-calculus of OptiTrust. The actual abstract syntax tree moreover features placeholders for carrying type information, as well as annotations used to guide the reverse translation.

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, as supported by OptiTrust.

Section 3.1 presents Optiλ, whose constructs appear in the statement of the proof rules and in the description of transformations. Section 3.2 details OptiC and its associated translation. Even though these ingredients are not a central contribution of OptiTrust, we find it relevant to present them at a high level, in order to relate the code presented in the case studies with our formalism. For additional details, we refer to a workshop article focusing specifically on the bidirectional translation [Bertholon and Charguéraud 2025].

3.1 OptiTrust's Internal AST

Figure 14 gives the grammar of Optiλ. 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 next describe the grammar, starting with the less common features. The standard, call-by-value semantics, may be found in Appendix C.

Sequences. A sequence is a term that consists of a list of subterms with side effects or **let** bindings, to be executed in order. Additionally, after the sequential execution of the subterms, a sequence may return a value. A sequence is written $\{t_1; ...; t_n; r\}$, where r denotes the optional return value for the sequence.[11] This return value may be absent, in that case we write it $\varnothing$. In our current implementation, the result value r is syntactically restricted to be either $\varnothing$ or a variable. We translate a statement of the form **return** t that appears in terminal position of a C-style function into "**let** $x = t;\ x$" where x is a fresh variable name.

A sequence $\{t_1; ...; t_n; r\}$ introduces a lexical scope. If t_i is of the form **let** $x = t$, then the variable x may occur in any t_j for $j > i$. The variable x does not scope beyond the closing brace. In Optiλ, those **let** bindings can only appear as instructions in a sequence. Every function body consists of a sequence block, even if the sequence contains a single instruction.

Moreover, in Optiλ, we enforce that all the instructions in a sequence do not have a return value. To do so, we insert calls to the built-in function "ignore" around instructions that are not

[11]This presentation of sequences is similar to that found in, e.g., the Rust language.

of type **void** in the OptiC code. Eliminating the implicitly ignored returned values coming from the user-facing language makes the AST more uniform, thereby simplifying typechecking and transformations.

Sequences in Optiλ may also include *ghost instructions*. A ghost instruction behaves, semantically, as a no-op. It guides, however, the typechecker of OptiTrust, typically by altering the way the memory state is described in the separation logic invariants. These invariants may be exploited for guiding code transformations, and for checking their correctness. A key interest of our design 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 depend on the result value. From the perspective of our bidirectional translation, ghost instructions are treated exactly like regular function calls.

Manipulation of heap and stack cells. To account for heap-allocated data, OptiTrust provides the following standard primitive functions: $\mathsf{heapAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}$ for allocating an uninitialized cell of type $\hat{\tau}$ on the heap, 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. More generally, heapAlloc can be used for matrix allocation. For example $\mathsf{heapAlloc}_{\mathsf{UninitMatrix2}_{\mathsf{int}}(5,8)}$ allocates an uninitialized matrix of 5×8 integers. Additionally, to account for stack-allocated variables, OptiTrust includes special functions. The operation $\mathsf{stackAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}()$ allocates a memory cell of type $\hat{\tau}$ on the stack without initializing its contents. The corresponding space is automatically reclaimed at the end of the surrounding sequence. Like for heapAlloc, stackAlloc can also be used to allocate matrices on the stack. The operation $\mathsf{ref}(t)$ also allocates a memory cell on the stack but initializes it with t. These two special operations are meant to occur as part of a **let** binding, for example **let** $x = \mathsf{ref}(3)$, occurring directly within a sequence. Note that a binding **let** $x = \mathsf{ref}(t)$ is semantically equivalent to **let** $x = \mathsf{stackAlloc}_{\mathsf{Cell}_{\hat{\tau}}}(); \mathsf{set}(x, t)$ where $\hat{\tau}$ is the type of t. The two stack-allocation operators, apart from their implicit-free behavior, are treated like other primitive functions.

Numbers. As said in the introduction, we currently ignore complications related to overflows and precision. Concretely, in Optiλ, the type **int** corresponds to $\mathbb{N}$, and the type **float** corresponds to $\mathbb{R}$. The specialization of number representations is currently left as a user-guided, unverified step, which takes place during the final extraction of C code.

*Possibly parallel range-based **for** loops.* The construct $\mathbf{for}^{\pi}\ (i \in \mathbf{range}(t_{\mathsf{start}}, t_{\mathsf{stop}}, t_{\mathsf{step}}))\ t_{\mathsf{body}}$ describes a *range-based **for** loop*. The immutable variable i denotes the loop index. The loop range consists of the loop bounds and the per-iteration step, that 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. The loop is tagged with an *execution mode* π that can be either **seq** of **par**. If π is set to **seq**, the loop is executed sequentially (i.e. one iteration at a time). If π is set to **par**, then the loop is treated as a parallel loop. The flag **par** corresponds to the directive: `#pragma openmp parallel`. The restrictions imposed by OpenMP on the ranges of parallel **for** loops essentially constrains them to fit the same $\mathbf{range}(t_{\mathsf{start}}, t_{\mathsf{stop}}, t_{\mathsf{step}})$ format. In this paper, we omit the execution mode of a loop when it is irrelevant.

Arrays. Arrays are allocated by means of a call to stackAlloc or heapAlloc. If a corresponds to the address of an array, then the memory address of i-th cell of the array a can be computed by the operation $a \boxplus i$. This operation corresponds to the C pointer arithmetic operation `a+i`, that is more intuitively written `&a[i]`. The contents of that cell may be retrieved by evaluating $\mathsf{get}(a \boxplus i)$. The address-shifting operation is here presented as a construct of the grammar. From the perspective of typechecking, however, we treat this operation as a function application for better factorization.

Other language constructs. The other language constructs of Optiλ are standard. They include function calls and conditionals. Our implementation accounts for a diversity of literal types. For simplicity, we consider in the formalization only two kinds of literals: the metavariable b denotes a boolean literal (either **true** or **false**), and the metavariable n denotes an integer literal. OptiTrust also supports record data types (a.k.a. struct); their handling is described in our workshop paper [Bertholon and Charguéraud 2025].

Other primitive operations. Besides the aforementioned primitive operations for manipulating heap and stack cells, Optiλ provides primitive functions that correspond to the arithmetic and boolean operators of the C language. One notable exception are the short-circuiting operators `&&` and `||` from C. We encode them in Optiλ using conditionals, carrying annotations for guiding the reverse translation as detailed further on. This keeps Optiλ semantics simple.

Annotations. In addition to the ghost instructions presented earlier, each subterm of an Optiλ program can carry a number of extra information that do not affect the semantics in the form of annotations. Currently, our internal AST carries the following information:

- user-placed marks allow referring to subterms by name in the targets of transformations;
- separation logic contracts for functions and loops;
- type information for all bindings, operators, and for every subterm;
- style annotations to guide the reverse translation from Optiλ to OptiC (see Section 3.2).

While loops. We do not yet support while-loops, nor the handling of abrupt termination, as triggered by **break**, **continue**, and non-final **return** statements. The treatment of abrupt termination in Separation Logic is well-understood [Cao et al. 2018; Kumar et al. 2014], yet it adds a fair amount of additional complexity.

3.2 Bidirectional Translation between OptiC and Optiλ

OptiTrust *encodes* OptiC input programs into the internal Optiλ language. Then, after one or several transformations on the internal AST, OptiTrust *decodes* programs back into OptiC. In the rest of this section, we first explain how C syntax is parsed to produce the OptiC AST. We then present the key ideas of the encoding from OptiC to Optiλ by means of example, and explain how annotations in Optiλ are used to ensure that a *round-trip* property holds.

Initial Parsing. The implementation of OptiTrust currently relies on Clang for parsing C syntax. The Clang AST is then translated into the OptiC AST. During this initial translation, some amount of semantically-irrelevant information may be lost. In particular, we currently do not attempt to keep in our ASTs information about comments and spacing. Beyond this initial translation, no more information is lost. Indeed, a *round-trip* property holds: encoding an OptiC program is the reciprocal to decoding an Optiλ program. Crucially, a lot of style information is preserved in Optiλ programs through annotations, as described next.

Annotations. To deal with the fact that several OptiC expressions might admit the same encoding in Optiλ, our translation attaches annotations on certain Optiλ terms. For example, the term `e1 && e2` and the term `e1 ? e2 : false` both admit the same encoding. When translating the latter form, we attach a "use-?" annotation on the conditional to indicate that it should remain printed as a ternary operator.

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, *pure* and *non-pure*, depending on whether the address of the variable needs to be manipulated. A variable x can only be *pure* if there is no assignment operation on x and if the address of the variable x is never computed via the address-of operator.[12] Equivalently, a variable x can be *pure* if and only if x could have been declared with the modifiers **const register**, in the terminology of the C standard. For such a pure variable, its definition, say **const int** x = 3, is encoded simply as **let** $x = 3$. For a non-pure variable, its encoding involves a stack-allocation. For example, the definition **int** x = 3 is encoded as **let** $x = \mathsf{ref}(3)$, the assignment x = 4 is encoded as $\mathsf{set}(x, 4)$, and an occurrence of &x is encoded as x.

The programmer may want to translate variables that can be pure into stack-allocated cells, to enable further code transformations. Hence, we need to rely on a keyword (or attribute) to indicate which variables should be translated without stack allocation. We could rely on **const register**, yet for brevity we decided that in OptiC the keyword **const** alone would indicate the intention of the programmer to introduce a pure variable. Figure 15 provides an example translation.

```
const int x = 3;                          ⟷ let_int x = 3;
f(x);                                     ⟷ f(x);

int z;                                    ⟷ let_ptr(int) z = stackAlloc_UninitCell_int();
z = 6;                                    ⟷ set(z, 6);
const int v = z;                          ⟷ let_int v = get(z);

int* const a = malloc(sizeof(int));       ⟷ let_ptr(int) a = heapAlloc_UninitCell_int();
*a = *a + 2;                              ⟷ set(a, get(a) + 2);
free(a);                                  ⟷ free(a);

int y = 5;                                ⟷ let_ptr(int) y = ref_int(5);
f(y);                                     ⟷ f(get(y));
y = y + 2;                                ⟷ set(y, get(y) + 2);
y += 4;                                   ⟷ inplaceAdd(y, 4);
y++;                                      ⟷ ignore(getThenIncr(y));

int* const p = &y;                        ⟷ let_ptr(int) p = y;
*p = *p + 2                               ⟷ set(p, get(p) + 2);

int* q = &y;                              ⟷ let_ptr(ptr(int)) q = ref_ptr(int)(y);
q = &z;                                   ⟷ set(q, z);
*q = *q + 2;                              ⟷ set(get(q), get(get(q)) + 2);
```

Fig. 15. Example translation from OptiC into the Optiλ. The functions ignore, inplaceAdd, and getThenIncr are provided by OptiTrust's library. The example assumes a function **void** f(**int**) to be defined.

4 VERIFYING PROOFS VIA TYPECHECKING

When designing OptiTrust, we have been seeking to build it upon the well-established ideas of separation logic [Reynolds 2002], which have demonstrated their expressiveness and scalability. In this section, we essentially adapt the standard proof rules of separation logic to the specificities of

[12] For example, a variable x cannot be *pure* if the code includes an occurrence of &x, or an expression of the form &x.f or &x[i]. That said, a variable x could be *pure* despite occurring below an address-of operator. For example, x could be a pure variable and appear as part of the expression &(x→f), which is encoded as $x.f$.

the grammar of Optiλ. We introduce one novelty, namely our proof rule for **for**-loops. It corresponds to the union of two standard separation logic proof rules: the one for sequential **for**-loops, and the one for parallel **for**-loops. Being able to describe intermediate combinations of the instantiation of these two rules enables the OptiTrust user to progressively refine a sequential **for**-loop into a parallel **for**-loop, by eliminating the sequential dependencies one after the other, and by refining at each step the loop contract accordingly.

A separation logic *triple* typically takes the form $\{H\}\ t\ \{\lambda r.H'\}$, where H denotes the precondition, H' denotes the postcondition, and r binds a name for the result of t. In OptiTrust, to ease the manipulation of contracts in transformations, we syntactically separate the *pure* and the *linear* parts of the pre- and postconditions, and write them E and F, respectively. Moreover, we introduce a canonical name **res** for referring to the result of the term in the postcondition, following a common practice in program verification tools, e.g., ESC/Java [Flanagan et al. 2002] and Why3 [Filliâtre 2003]. Hence, our (semantic) triples take the form $\{\Gamma\}\ t\ \{\Gamma'\}$, where Γ decomposes as $\langle E \mid F\rangle$, and Γ' decomposes as $\langle E' \mid F'\rangle$, and where **res** is bound in E'.

Our typechecking algorithm computes, for every subterm, a *context*, written Γ, that describes the pure facts and linear permissions available for typing that subterm. Our algorithm is described in terms of *algorithmic triples*, written $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$. By design of our algorithm, they take the form $\{\!\{\langle E \mid F\rangle\}\!\}\ t\ \{\!\{\langle E, E' \mid F'\rangle\}\!\}$, purposely repeating the pure facts from E to ease the propagation of information. The algorithmic triples, which correspond specifically to the triples computed by our algorithm, entail semantics triples, in the sense that: $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\} \implies \{\Gamma\}\ t\ \{\Gamma'\}$. Triples will be later extended in Section 6 to the form $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$, where Δ denotes a *usage map* that provides a summary explaining which resources are used by each subterm, and how they are used.

Our algorithmic proof rules aim to automate the computation of the *frames*—the permissions that are not used—as done in nearly all frameworks based on separation logic, e.g., [Cao et al. 2018; Charguéraud 2011; Jacobs and Piessens 2008; Jung et al. 2018b]. They also aim to automate the manipulation of *fractional permissions*—the tooling used to describe read-only permissions [Boyland 2003] and used extensively in modern concurrent separation logic [Jung et al. 2018a]. To that end, we follow the technique introduced in the Chalice verification tool [Heule et al. 2013], whereby fractional permissions are "carved out" in cascade (details are given further on in Section 4.1). Overall, we are basing the logic of OptiTrust on standard, time-tested separation logic techniques.

We argue for the soundness of OptiTrust's typechecker by showing how each of our typing ingredients maps to the corresponding standard notion from separation logic. In Appendix I, we present the most interesting technical proof, which establishes that our algorithmic rule for typechecking for-loops is derivable from two standard reasoning rules of separation logic: the invariant-based rule for sequential loops, and the contract-based rule for parallel loops. Additional details related to the realization of triples and the frame property may be found in the first author's PhD manuscript [Bertholon 2025, §4.8 and appendices A-D]. Note that we are not interested in developing a thorough formalization of our algorithmic proof rules. As explained in the introduction, our long-term plan is to extract, from OptiTrust derivations, standard separation logic derivations that could be independently typechecked in Rocq or Lean, thereby paving the way to foundational verification of optimized code.

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* and *contracts*. Section 4.3 presents the contracts for primitive functions. Section 4.4 presents the typing judgment for *logical expressions*. Section 4.5 presents the *entailment relation* and *subtraction procedure*, which corresponds to an algorithmic implementation of entailment. Section 4.6 presents our algorithmic proof rules, which

define the judgment $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, with Section 4.7 focusing specifically on function calls and Section 4.8 focusing on loops.

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

4.1 Grammar of Resources

A context Γ consists of a pair of the form $\langle E \mid F \rangle$, where E contains pure resources and F contains linear resources. A pure resource describes a fact that remains true until the end of the program, or describes a program variable permanently bound to a given value. Pure resources may be freely duplicated during typechecking. Each linear resource describes the ownership of a given subset of the memory. Two linear resources that appear in a same context must describe disjoint parts of the memory, assuming they are *full*. A linear resource is not full if it was split into *fractional* resources, in which case several fractional linear resources may cover the same parts of memory. Subsequently, resources that were split may be joined back together. In any case, a linear resource cannot be duplicated and cannot be silently dropped. We next describe the grammar of pure and linear resources.

Pure resources. The pure part of a typing context contains resources that are bindings of the form "$x : \tau$", where τ corresponds either to an OptiC type or to a *logical type*. An OptiC type is denoted by the meta-variable $\hat{\tau}$. 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 immutable program variables to OptiC types, and a Rocq context, which binds ghost variables to Rocq types. Let us give examples of bindings that may appear in a pure context—that is, in the pure part of a context :

- "$\tau : \mathsf{Type}$" quantifies a type variable, useful for expressing polymorphism in Optiλ.
- "$x : \tau$" quantifies a variable of type τ; and "$x : \hat{\tau}$" quantifies a variable with the OptiC type $\hat{\tau}$.
- "$f : \tau_1 \xrightarrow{\text{logic}} \tau_2$" quantifies a logical function that takes an argument of type τ_1 and returns an argument of type τ_2. Logical functions corresponds to functions that are pure and terminating. "$f : \forall (x_1 : \tau_1)(x_2 : \tau_2),\ \tau_3$" quantifies a type dependent logical function with two arguments. In that example, x_1 can appear in τ_2 and τ_3, and x_2 can appear in τ_3. In practice the non-dependent logical function syntax "$(\tau_1 \times ... \times \tau_n) \xrightarrow{\text{logic}} \tau_0$" is syntactic sugar for "$\forall (x_1 : \tau_1)...(x_n : \tau_n),\ \tau_0$" with all x_i fresh from all τ_j.
- "$P : \mathsf{Prop}$" quantifies an abstract proposition; and "$Q : \tau \xrightarrow{\text{logic}} \mathsf{Prop}$" quantifies an abstract logical predicate over values of type τ.
- "$p : P$" quantifies a proof witness of a proposition P; for example "$p : i > 0$" captures the assumption that i is positive.
- "$p : \mathsf{Spec}(f, [a_1, ..., a_n], \gamma)$" describes a *function specification*[13] asserting that the function f expects arguments named a_i and admits the *function contract* γ.
- "$H : \mathsf{HProp}$" quantifies an abstract heap predicate[14], and "$I : \mathsf{int} \xrightarrow{\text{logic}} \mathsf{HProp}$" quantifies an abstract invariant parameterized by an integer.

Linear resources. The linear part of a typing context contains resources that are bindings of the form "$y : H$", where H is a *heap predicate*, and where y is a name. For example, "$y : p \rightsquigarrow v$" is a

[13] Function contracts may appear in typing contexts, while typing contexts are involved in stating function contracts. This form of *impredicativity* is standard in higher-order separation logic [Charguéraud 2020; Varming and Birkedal 2008].

[14] In formalizations of separation logic, HProp is typically defined as "state $\xrightarrow{\text{logic}}$ Prop", where state denotes the type of a memory state. This definition, however, needs not be revealed to the OptiTrust user.

Syntax in OptiC	Syntax in the theory	Description
$p \mapsto v$	$p \mapsto v$	gives access to the cell at address p, specified to contain v
$p \rightsquigarrow$ UninitCell	$p \rightsquigarrow \mathsf{UninitCell}_{\hat{t}}$	gives write-only access to the cell at address p
for i in $R \to H(i)$	$\bigstar_{i \in R} H(i)$	union of resources $H(i)$, for i in the range R
_RO(α, H)	αH	read-only version of the resource H with fraction α
_Dealloc(p, H)	$\mathsf{Dealloc}(p, H)$	allows to free p by giving away the resource H
_Wand(H_1, H_2)	$H_1 \mathrel{-\!\!\!\star} H_2$	allows transforming H_1 into H_2 once, via a ghost call

Table 1. Grammar of heap predicates. User-defined representation predicates are left to future work.

Syntax in OptiC	Desugared version
$p \rightsquigarrow$ Matrix1(n, M)	$\bigstar_{i \in 0..n} p \boxplus \mathsf{mIndex1}(n, i) \mapsto M(i)$
$p \rightsquigarrow$ Matrix2(m, n, M)	$\bigstar_{i \in 0..m} \bigstar_{j \in 0..n} p \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow M(i, j)$
$p \rightsquigarrow$ UninitMatrix1(n)	$\bigstar_{i \in 0..n} p \boxplus \mathsf{mIndex1}(n, i) \rightsquigarrow \mathsf{UninitCell}_{\hat{t}}$
$p \rightsquigarrow$ UninitMatrix2(m, n)	$\bigstar_{i \in 0..m} \bigstar_{j \in 0..n} p \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow \mathsf{UninitCell}_{\hat{t}}$

Table 2. Syntactic sugar for heap predicates describing arrays and matrices.

resource. This name y is used in particular to refer to resources in usage maps. A heap predicate H describes ownership of a memory region. When a linear context contains several resources, each resource must describe a disjoint part of the memory. Interestingly, heap predicates guarantee the absence of hidden aliasing.

Table 1 gives the grammar of the currently supported heap predicates, using mathematical notations, with the type explicit. The constructions are explained in the paragraphs that follow. Table 2 shows the pattern for defining heap predicates for arrays and matrices.

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 memory region gives read-only access to this region. Possessing the full fraction (i.e. 1) of a memory region gives read-write exclusive access to this region.

For any fraction α and heap predicate H, if we have a resource αH at hand in the context, we can *split* it into two disjoint read-only permissions βH and $(\alpha - \beta)H$. This splitting operation can be performed for any fraction β such that $0 < \beta < \alpha$. In that case, we can alternatively say that a subfraction βH was *carved* out of the initial permission αH.

For simple enough heap predicates such as all those built from constructions presented in Table 1, possessing both αH and βH is equivalent to possessing the single resource $(\alpha + \beta)H$.[15] In practice, OptiTrust currently only manipulates heap predicates such that this equivalence holds. This means that two fractions of the same heap predicate can always be merged together, to reconstitute bigger fractions.

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 proof rules. The

[15]Heap predicates using disjunction or existential quantifiers cannot be merged together in the general case. Indeed, the resource $\frac{1}{2}(p \rightsquigarrow \mathsf{Cell} \vee q \rightsquigarrow \mathsf{Cell}) \star \frac{1}{2}(p \rightsquigarrow \mathsf{Cell} \vee q \rightsquigarrow \mathsf{Cell})$, does not imply a permission to write in either p or q, while a permission $p \rightsquigarrow \mathsf{Cell} \vee q \rightsquigarrow \mathsf{Cell}$ does.

operation CloseFracs repeatedly applies the following rewrite rule:

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

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 $1H$.

To simplify carve, merge and unification operations, during typechecking we normalize heap predicates of the form $\star_{i \in R}\, \alpha H_i$ into $\alpha(\star_{i \in R}\, H_i)$. While this operation can be unsound in certain models of separation logic, it is well justified in *unbounded separation logic* [Dardinier et al. 2022].[16]

Resources for uninitialized cells. Separation logic can guarantee that a program never reads from an uninitialized memory cell. To provide this guarantee, allocation of a memory cell at address p is specified to produce not a regular heap predicate of the form $p \mapsto v$, but instead an uninitialized predicate written $p \rightsquigarrow \mathsf{UninitCell}$ in OptiTrust.[17]

Then, the specification of the read operation requires a permission of the form $\alpha(p \mapsto v)$, which cannot be extracted from a permission of the form $p \rightsquigarrow \mathsf{UninitCell}$.

On the contrary, the specification of an operation writing a value v at a location p requires a permission of the form $p \rightsquigarrow \mathsf{UninitCell}$, and produces the full permission $p \rightsquigarrow v$. In order to be able to execute a write operation when we already have a full permission, we allow a permission $p \mapsto v$ to be weakened into $p \rightsquigarrow \mathsf{UninitCell}$ at any time.

Concretely, when our typechecker requires $p \rightsquigarrow \mathsf{UninitCell}$ in a context where $p \mapsto v$ is available, it applies the weakening rule on-the-fly. This weakening can also be performed under separating conjunctions. We write $\mathsf{Uninit}(H)$ such weakening operation on an arbitrary heap predicate H. It returns a heap predicate when it succeeds and $\bot$ otherwise. If H is already an uninitialized resource, then $\mathsf{Uninit}(H) = H$.

Permission to free. In OptiTrust, heap allocations return two permissions: an uninitialized heap predicate H of the form $p \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}$, $p \rightsquigarrow \mathsf{UninitMatrix1}_{\hat{\tau}}(n)$, or $p \rightsquigarrow \mathsf{UninitMatrix2}_{\hat{\tau}}(m, n)$ and a second permission written $\mathsf{Dealloc}(p, H)$. Such permission $\mathsf{Dealloc}(p, H)$ can be later used together with H to free the allocated cells. The separation between the permission to write and the permission to free helps to make specifications and transformations more generic: indeed, unless the code needs to free memory, the permission $\mathsf{Dealloc}(p, H)$ can be simply ignored.

Magic wand permission. As we saw in the case studies, the ghost operation focus that extracts a permission to a specific cell from a permission to the full array also generates a second permission representing the rest of the array. In separation logic, such remainder permission is usually modelled by a heap predicate called *magic wand*. A magic wand is denoted $H_1 \mathrel{-\!\!\star} H_2$, where H_2 is a heap predicate from which H_1 was extracted. The predicate $H_1 \mathrel{-\!\!\star} H_2$ can then be recombined with H_1 to get H_2 back.

[16] In standard separation logic [Reynolds 2002], $\frac{1}{2}(p \rightsquigarrow v) \star \frac{1}{2}(p \rightsquigarrow v)$ is different from $\frac{1}{2}(p \rightsquigarrow v \star p \rightsquigarrow v)$ because the latter is never satisfiable. Indeed, $p \rightsquigarrow v \star p \rightsquigarrow v$ mentions p twice so it cannot be true for any memory region, therefore taking a fraction $\frac{1}{2}$ of such region is not possible either. Unbounded separation logic [Dardinier et al. 2022] resolves the issue by considering that temporary memory regions can possess a resource with a fraction bigger than one, but that such non-standard fractions are forbidden on memory regions described at the level of Hoare triples, ensuring the usual non-aliasing properties.

[17] Certain separation logic frameworks reuse the permission $p \mapsto v$ to encode uninitialized permissions by extending the grammar of values with a special uninitialized value written $\bot$. The specification for a read operation requires in its precondition that the value read must not be $\bot$. So far, we have found that a dedicated uninitialized representation predicate to be simpler to handle in our framework.

4.2 Grammar of Contexts and Contracts

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. For a context $\Gamma = \langle E \mid F \rangle$, the projection "$\Gamma$.pure" returns E, and the projection "Γ.linear" returns F.

Following standard practice from proof assistants, unused variable names may be elided. For example the context $\langle p : \mathrm{ptr}(\mathrm{int}), n : \mathrm{int}, n > 0 \mid p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}} \rangle$ contains two anonymous resources: $n > 0$ and $p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}}$. Internally, though, all context items are given an identifier.

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

Alias bindings. The pure part E of a context Γ may contain bindings of a special form, called *alias bindings*. Such a binding takes the form "$x_i := v_i : \tau_i$". 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 Rocq. An alias binding "$x_i := v_i : \tau_i$" may also be interpreted as a conventional binding that associates x_i to a singleton type whose sole inhabitant is v_i.

In contexts, we use a special variable **res** as a canonical name to denote the result value of an expression. Therefore, if t has a non-void type τ then, in the triple $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, this variable **res** may be bound in Γ' as an alias. The variable **res** also appears in function contracts, to specify properties about the return value of the function.

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, we define $\Gamma_1 \circledast \Gamma_2$ as $\langle \Gamma_1.\mathrm{pure}, \Gamma_2.\mathrm{pure} \mid \Gamma_1.\mathrm{linear} \star \Gamma_2.\mathrm{linear} \rangle$, assuming the variables in this result are well-scoped (that is, Γ_1 and Γ_2 have disjoint domains and the formulas in Γ_2 are well-scoped in Γ_1.pure). Observe that $[E] \circledast F = \langle E \mid F \rangle$.

Grammar of function contracts. To guide the typechecker, every function carries a *function contract*, written γ, thus taking the form $\mathbf{fun}(a_1, ..., a_n)_\gamma \mapsto t$. Such a contract consists of two contexts, one for the *precondition*, written γ.pre, and one for the *postcondition*, written γ.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 contract γ takes the form $\{\mathrm{pre} = \Gamma_{\mathrm{pre}}\ ;\ \mathrm{post} = \Gamma_{\mathrm{post}}\}$. 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}.

The relationship with the concrete OptiC syntax for function contracts is explained in Figure 16. Any function contract can be expressed using exclusively a combination of **`__requires`**, **`__ensures`**, **`__consumes`** and **`__produces`** clauses. The other three clauses **`__preserves`**, **`__reads`** and **`__writes`** provide syntactic sugar for common patterns, to avoid repetitions. Moreover, one clause can specify multiple resources at once. For example, **`__requires`**`("x1: int, x2: int")` is interpreted as **`__requires`**`("x1: int"); `**`__requires`**`("x2: int")`.

OptiC clause	γ.pre	γ.post
`__requires("`$x : \tau$`");`	$\langle x : \tau \vert \rangle$	$\langle \vert \rangle$
`__ensures("`$x : \tau$`");`	$\langle \vert \rangle$	$\langle x : \tau \vert \rangle$
`__consumes("`$y : H$`");`	$\langle \vert y : H \rangle$	$\langle \vert \rangle$
`__produces("`$y : H$`");`	$\langle \vert \rangle$	$\langle \vert y : H \rangle$
`__preserves("`$y : H$`");`	$\langle \vert y : H \rangle$	$\langle \vert y : H \rangle$
`__reads("`$y : H$`");`	$\langle \alpha : \mathsf{frac} \mid y : \alpha H \rangle$	$\langle \vert y : \alpha H \rangle$
`__writes("`$y : H$`");`	$\langle \vert y : \mathsf{Uninit}(H) \rangle$	$\langle \vert y : H \rangle$

Fig. 16. Interpretation of OptiC syntax for the clauses defining a function contract γ.

OptiC clause	χ.vars	χ.excl.pre	χ.excl.post	χ.shrd.reads	χ.shrd.inv
`__requires("`$x : \tau$`");`	$x : \tau$	$\langle \vert \rangle$	$\langle \vert \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xrequires("`$x : \tau$`");`	$\varnothing$	$\langle x : \tau \vert \rangle$	$\langle \vert \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xensures("`$x : \tau$`");`	$\varnothing$	$\langle \vert \rangle$	$\langle x : \tau \vert \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xconsumes("`$y : H$`");`	$\varnothing$	$\langle \vert y : H \rangle$	$\langle \vert \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xproduces("`$y : H$`");`	$\varnothing$	$\langle \vert \rangle$	$\langle \vert y : H \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xpreserves("`$y : H$`");`	$\varnothing$	$\langle \vert y : H \rangle$	$\langle \vert y : H \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xreads("`$y : H$`");`	$\alpha : \mathsf{frac}$	$\langle \vert y : \alpha H \rangle$	$\langle \vert y : \alpha H \rangle$	$\varnothing$	$\langle \vert \rangle$
`__xwrites("`$y : H$`");`	$\varnothing$	$\langle \vert y : \mathsf{Uninit}(H) \rangle$	$\langle \vert y : H \rangle$	$\varnothing$	$\langle \vert \rangle$
`__sreads("`$y : H$`");`	$\alpha : \mathsf{frac}$	$\langle \vert \rangle$	$\langle \vert \rangle$	$y : \alpha H$	$\langle \vert \rangle$
`__srequires("`$x : \tau$`");`	$\varnothing$	$\langle \vert \rangle$	$\langle \vert \rangle$	$\varnothing$	$\langle x : \tau \vert \rangle$
`__spreserves("`$y : H$`");`	$\varnothing$	$\langle \vert \rangle$	$\langle \vert \rangle$	$\varnothing$	$\langle \vert y : H \rangle$

Fig. 17. Interpretation of OptiC syntax for the clauses defining a loop contract χ.

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

$$\begin{cases} \mathsf{vars} = E_{\mathsf{vars}} & \text{Pure variables, scoping over the other contract components} \\ \mathsf{excl} = \gamma_{\mathsf{excl}} & \text{Exclusive per-iteration resources} \\ \mathsf{shrd} = \begin{cases} \mathsf{reads} = F_{\mathsf{reads}} & \text{Read only resources shared between iterations} \\ \mathsf{inv} = \Gamma_{\mathsf{inv}} & \text{Sequential invariant, threaded through iterations} \end{cases} \end{cases}$$

The relationship with OptiC syntax is described in Figure 17. The components of the record are explained next.

We call E_{vars} the *loop ghost variables*. The variables from E_{vars} scope over γ_{excl}, F_{reads} and Γ_{inv}.

We call γ_{excl} the *exclusive per-iteration contract*. This γ_{excl} has the same structure as a function contract, and may (and typically does) refer to the loop index. γ_{excl}.pre.linear correspond to the *consumed per-iteration resources*, and γ_{excl}.post.linear corresponds to the *produced per-iteration resources*. γ_{excl}.pre.pure corresponds to pure bindings whose value can depend on the iteration, but that must be chosen once before the loop starts. These bindings can be used inside γ_{excl}.pre.linear and in γ_{excl}.post. γ_{excl}.post.pure is a pure context of pure resources ensured by each iteration independently of each other. These resource bindings can be used inside γ_{excl}.post.linear.

We call F_{reads} the *shared reads*. In practice this context consists of read-only resources. Resources in F_{reads} cannot refer to the loop index. Each loop iteration receives a subfraction of each resource in F_{reads}, and must give it back at the end.

$$\{\}\quad \mathsf{heapAlloc}_{C_{\hat{\tau}}}()\quad \{[\mathbf{res} : \mathsf{ptr}(\hat{\tau})] \circledast \mathbf{res} \rightsquigarrow C_{\hat{\tau}} \circledast \mathsf{Dealloc}(\mathbf{res}, \mathbf{res} \rightsquigarrow C_{\hat{\tau}})\}$$
$$\{[\hat{\tau} : \mathsf{Type}, x : \mathsf{ptr}(\hat{\tau}), \alpha : \mathsf{frac}, v : \hat{\tau}] \circledast \alpha(x \mapsto v)\}\quad \mathsf{get}(x)\quad \{[\mathbf{res} := v : \hat{\tau}] \circledast \alpha(x \mapsto v)\}$$
$$\{[\hat{\tau} : \mathsf{Type}, x : \mathsf{ptr}(\hat{\tau}), y : \hat{\tau}] \circledast x \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}\}\quad \mathsf{set}(x, y)\quad \{x \mapsto y\}$$
$$\{[\hat{\tau} : \mathsf{Type}, x : \mathsf{ptr}(\hat{\tau}), H : \mathsf{HProp}] \circledast \mathsf{Dealloc}(x, H) \circledast H\}\quad \mathsf{free}(x)\quad \{\}$$

Fig. 18. Contracts assigned to key primitive functions; $\hat{\tau}$ denotes a C type; x and y denote program variables. $C_{\hat{\tau}}$ is either $\mathsf{UninitCell}_{\hat{\tau}}$, $\mathsf{UninitMatrix1}_{\hat{\tau}}(n)$, or $\mathsf{UninitMatrix2}_{\hat{\tau}}(m, n)$, for size expressions m and n.

Finally, we call Γ_{inv} the *sequential invariant*. It corresponds to a standard loop invariant in sequential separation logic, and it typically depends on the loop index. Γ_{inv}.pure corresponds to pure resources that are given or assumed at the beginning of each iteration, and that each iteration must choose or ensure for the next one with the subsequent loop index. Similarly, Γ_{inv}.linear corresponds to linear resources that are consumed at the beginning of each iteration and that must be produced at the end of the iteration for the next loop index.

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

4.3 Contracts for Primitive Functions

Contracts for memory operations. Figure 18 gives the contracts that we axiomatize for the operations on heap cells, here presented using triples for improved readability. These contracts illustrate key mechanisms of the formalism. A heap allocation produces an uninitialized permission over a single cell or a matrix and a permission to free these allocated cells. A write operation requires an uninitialized permission and returns a full permission. A read operation requires a read-only permission and returns it. A free operation requires a permission to free, the associated uninitialized permission and returns nothing. 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. Additionally, we can see that bindings on **res** appear in output contexts.

Contracts for arithmetic operations are described later on, in Section 4.4.

Contracts for ghost functions. In addition to contracts for primitive heap-manipulating functions, OptiTrust provides contracts for primitive ghost functions. For example, the ghost function swap_groups allows swapping two iterators (iterated separating conjunctions). It is involved for example in the Loop.swap transformation, which is used in our case studies (Section 2), and which is presented further on in Section 5.5. The transformation is specified as shown below, where H is a heap predicate that depends on the two indices i and j. The type range corresponds to the type of loop ranges.

$$\{[R_i : \mathsf{range}, R_j : \mathsf{range}, H : (\mathsf{int}, \mathsf{int}) \xrightarrow{\text{logic}} \mathsf{HProp}] \circledast \bigstar_{i \in R_i} \bigstar_{j \in R_j} H(i, j)\}$$
$$\mathsf{swap\_groups}$$
$$\{\bigstar_{j \in R_j} \bigstar_{i \in R_i} H(i, j)\}$$

The OptiTrust user can define custom ghost functions to factorize common resource-manipulation patterns. Ghost functions are written and typechecked like regular C functions, with the restriction that their bodies must contain only calls to other ghost functions, sequences and range-based **for** loops. Importantly, the body of a ghost function never needs to be executed, it simply serves as a proof witness.

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

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

$$\textsc{LogicFun}\ \frac{(E, x_1 : \tau_1, ..., x_n : \tau_n) \vdash t : \tau_0}{E \vdash (\mathbf{fun}(x_1 : \tau_1, ..., x_n : \tau_n) \mapsto t) : \forall (x_1 : \tau_1)...(x_n : \tau_n),\ \tau_0}$$

$$\textsc{LogicApp}\ \frac{E \vdash t_0 : \forall (x_1 : \tau_1)...(x_n : \tau_n),\ \tau_0 \qquad \forall i \in [1, n],\ \ E \vdash t_i : \mathsf{Subst}\{x_1 \mapsto t_1; ...; x_{i-1} \mapsto t_{i-1}\}(\tau_i)}{E \vdash t_0(t_1, ..., t_n) : \mathsf{Subst}\{x_1 \mapsto t_1; ...; x_n \mapsto t_n\}(\tau_0)}$$

Fig. 19. Selected rules defining the typing judgment for logical expressions, written $E \vdash t : \tau$. Arithmetic operations such as $t_1 + t_2$ are viewed as functions calls and are therefore handled by the rule LogicApp.

4.4 Typechecking of Logical Expressions

A *logical expression* is an expression that may appear in specifications and invariants; technically, a logical expression is an expression whose evaluation terminates and does not depend on the memory state. Logical expressions include program variables (which are always immutable in Optiλ), constant literals, logical propositions, heap predicates, C types, logical types, pure function definitions, and pure function calls. Figure 19 shows the main proof rules for logical expressions. The judgment is written $E \vdash t : \tau$, where E is a pure context.

An arithmetic expression (e.g. $t_1 + t_2$) can be considered as a logical expression (e.g. the application of the logical function for addition) if its two arguments are pure. This allows us to express the contract for the primitive arithmetic operations by referring to the corresponding logical expression. For instance, the contract for addition is: $\{[a : \mathsf{int}, b : \mathsf{int}]\}\ (a + b)\ \{[\mathbf{res} := a \mathbin{\hat{+}} b : \mathsf{int}]\}$, where for clarity we distinguished the addition operator from the programming language denoted $+$, and the addition operator from the logic denoted $\hat{+}$. Partial primitive arithmetic functions have a contract expressed with the corresponding logical expression, but those require an additional precondition. For example, 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 the divisor is equal to zero.

4.5 Entailment and Subtraction

An essential ingredient of separation logic is the *entailment* judgment, which captures a form of weakening. The relation $F_1 \Rightarrow F_2$ asserts that any memory state described by F_1 can also be described by F_2.[18] By extension, we define the judgment $\Gamma_1 \Rightarrow \Gamma_2$. Recall that the contexts decompose between their pure part and their linear part. The entailment $\langle E_1 | F_1 \rangle \Rightarrow \langle E_2 | F_2 \rangle$ asserts that, for any instantiation of the variables bounds in E_1, a memory state described by F_1 can, for a well-chosen

[18]The literature on separation logic consider both *linear* and *affine* entailment relations—see, e.g., [Charguéraud 2020]. OptiTrust is based on a linear entailment relation, disallowing resources to be silently "dropped". This property allows us to check the absence of memory leaks: every allocated piece of memory must be freed eventually.

instantiation of the bindings in E_2, also be described by F_2. (Note that E_1 binds in E_2 and F_2.) Let us give a few examples.

- $\langle x : \mathsf{loc} \mid x \rightsquigarrow v \rangle \Rightarrow \langle \alpha : \mathsf{frac} \mid \alpha(x \rightsquigarrow v), (1-\alpha)(x \rightsquigarrow v) \rangle$
- $\langle y : \mathsf{loc} \mid y \rightsquigarrow v \rangle \Rightarrow \langle \varnothing \mid y \rightsquigarrow \mathsf{UninitCell} \rangle$
- $\langle a : \mathsf{loc}, m : \mathsf{int}, n : \mathsf{int} \mid \bigstar_{i \in 0..m} \bigstar_{j \in 0..n} (a \boxplus \mathsf{mIndex2}(m, n, i, j)) \rightsquigarrow v \rangle$
  $\Rightarrow \langle \mid \bigstar_{j \in 0..n} \bigstar_{i \in 0..m} (a \boxplus \mathsf{mIndex2}(m, n, i, j)) \rightsquigarrow v \rangle$
- $\langle n : \mathsf{int}, P : \mathsf{even}\, n \mid \varnothing \rangle \Rightarrow \langle m : \mathsf{int}, Q : n = 2m \mid \varnothing \rangle$
- Not true: $\langle x : \mathsf{loc} \mid x \rightsquigarrow v \rangle \Rightarrow \langle \varnothing \mid \varnothing \rangle$, as linear resources cannot be dropped.
- Not true: $\langle x : \mathsf{loc} \mid x \rightsquigarrow v \rangle \Rightarrow \langle x : \mathsf{loc} \mid x \rightsquigarrow v,\ x \rightsquigarrow v \rangle$, as linear resources cannot be duplicated.

The algorithmic typechecking rules require a sound algorithmic implementation of the entailment judgement. For example, at the end of a function, the typechecker must verify that the description of the current state *entails* the claimed postcondition of the function. The operator that implements the algorithmic counterpart of entailment is called *subtraction*. Most—if not all—practical verification frameworks based on separation logic include a subtraction operator. The proof 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 full resources into uninitialized resources on-the-fly, however it does not support splitting read-only resources on-the-fly (recall Section 4.1). This subtraction operation may fail (that is, return $\bot$) if a resource in Γ' cannot be matched against a corresponding resource in Γ. Otherwise, $\Gamma \boxminus \Gamma'$ returns a result of the form (σ, F) such that, intuitively, $\Gamma \Rightarrow \mathsf{Subst}\{\sigma\}(\Gamma') \circledast F$, where F denotes the *frame* and where σ provides the instantiation of the pure bindings.[19] In particular, $(\sigma, \varnothing) = (\Gamma \boxminus \Gamma')$ implies $\Gamma \Rightarrow \Gamma'$. Details on the implementation of subtraction operators may be found in appendix D.

The *carving subtraction operation*, written $\Gamma \ominus \Gamma'$, extends $\Gamma \boxminus \Gamma'$ with the ability to carve out a fraction of a read-only permission from Γ, each time a corresponding read-only permission appears in Γ'. The operation $\Gamma \ominus \Gamma'$ produces a result of the form $(E_{\mathsf{frac}}, \sigma, F)$, where E_{frac} is a pure context containing only bindings of the form "$\alpha : \mathsf{frac}$". Intuitively, the entities returned are such $\Gamma \Rightarrow [E_{\mathsf{frac}}] \circledast \mathsf{Subst}\{\sigma\}(\Gamma') \circledast F$.[20]

4.6 Typechecking of Terms

We are now ready to describe our typechecking algorithm, which we do by means of presenting the rules for deriving the algorithmic triples written $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$.

As said previously, in order to maximize the amount of information that is propagated forward during the typechecking, all the *pure* bindings from Γ are repeated in Γ'. The pure bindings that appear in Γ' but not in Γ correspond either to the binding or **res** (which names the result of t), or to the quantification of ghost variables in the postconditions, or to the statement of propositions as part of the postcondition.

In general, the *linear* bindings of Γ' can differ arbitrarily from the linear bindings of Γ. We nevertheless enforce the invariant that a permission can appear under the same name in both Γ and Γ' if and only if this resource is not updated by t. This situation concerns permissions that t uses for read-access only, or permissions that t does not use at all.

Figure 20 presents the proof rules, except the ones for function calls and for loops, which are explained further on.

[19] Technically, if $\Gamma \boxminus \Gamma' = (\sigma, F)$, then both the entailments $\Gamma \Rightarrow \Gamma' \circledast F$ and $\Gamma \Rightarrow \mathsf{Specialize}_{\Gamma}\{\sigma\}(\Gamma') \circledast F$ hold, for the definition of Specialize presented further on in Section 4.7.

[20] Technically, if $\Gamma \ominus \Gamma' = (E_{\mathsf{frac}}, \sigma, F)$, then both the entailments $\Gamma \Rightarrow \Gamma' \circledast \langle E_{\mathsf{frac}} | F \rangle$ and $\Gamma \Rightarrow [E_{\mathsf{frac}}] \circledast \mathsf{Specialize}_{\Gamma}\{\sigma\}(\Gamma') \circledast F$ hold.

$$\frac{\Gamma.\text{pure} \vdash t : \tau \qquad t \text{ is a literal or a variable}}{\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma \circledast [\mathbf{res} := t : \tau]\}\!\}}\ \textsc{LitOrVar}$$

$$\frac{\{\!\{\Gamma_0\}\!\}\ t\ \{\!\{\Gamma_1\}\!\} \qquad \Gamma_2 = \text{Rename}\{\mathbf{res} := x\}(\Gamma_1)}{\{\!\{\Gamma_0\}\!\}\ \mathbf{let}\ x = t\ \{\!\{\Gamma_2\}\!\}}\ \textsc{Let}$$

$$\frac{\forall i \in [1, n].\quad \{\!\{\Gamma_{i-1}\}\!\}\ t_i\ \{\!\{\Gamma_i\}\!\} \qquad \Gamma_r = \begin{cases} \text{Rename}\{r := \mathbf{res}\}(\Gamma_n) & \text{if } r \neq \varnothing \\ \Gamma_n & \text{if } r = \varnothing \end{cases}}{\{\!\{\Gamma_0\}\!\}\ \big(t_1; ...; t_n; r\big)\ \{\!\{\Gamma_r\}\!\}}\ \textsc{Seq}$$

$$\frac{\{\!\{\Gamma_0\}\!\}\ \big(t_1; ...; t_n; r\big)\ \{\!\{\Gamma_r\}\!\} \qquad (\varnothing, F) = \Gamma_r \boxminus \text{StackAllocCells}(t_1, ..., t_n)}{\{\!\{\Gamma_0\}\!\}\ \big\{t_1; ...; t_n; r\big\}\ \{\!\{\langle \Gamma_r.\text{pure} \mid F\rangle\}\!\}}\ \textsc{Block}$$

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

$$\frac{\begin{array}{c}\{\!\{\Gamma_0\}\!\}\ t_1\ \{\!\{\Gamma_1\}\!\} \qquad \{\!\{\text{Learn}\{\mathbf{res} = \mathbf{true}\}(\Gamma_1)\}\!\}\ t_2\ \{\!\{\Gamma_2\}\!\} \qquad \{\!\{\text{Learn}\{\mathbf{res} = \mathbf{false}\}(\Gamma_1)\}\!\}\ t_3\ \{\!\{\Gamma_3\}\!\} \\ (\_, \varnothing) = \Gamma_2 \boxminus \Gamma_r \qquad (\_, \varnothing) = \Gamma_3 \boxminus \Gamma_r\end{array}}{\{\!\{\Gamma_0\}\!\}\ \mathbf{if}_{\Gamma_r}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \{\!\{\Gamma_r\}\!\}}\ \textsc{If}$$

Fig. 20. Algorithmic proof rules for establishing triples of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, except the rules for function calls and for-loops, which are presented further on.

Literals and variables. Consider a term t that corresponds either to a program variable or to a literal. In its triple, of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, the output context Γ' is obtained by extending Γ with an alias binding from **res** to t itself. Alias bindings were defined in Section 4.2. Note that it is acceptable for t to appear in a logical context because when t is a literal or a variable, t qualifies as a logical expression. The type of t is computed by means of the typing judgment for logical expressions, defined in Section 4.4.

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. This replacement is implemented by the Rename operator, described next.

Renaming on contexts. The operation $\text{Rename}\{\rho\}(\Gamma)$ where ρ denotes a map that associates resource names to other resource names, renames certain keys in a context Γ. The keys from ρ may or may not be bound in Γ. The values from ρ must be fresh from Γ. For example, $\text{Rename}\{x := x', y := y'\}(\langle E_1, x : \tau, E_2 \mid F\rangle)$, where y has no occurrence in E_1, E_2 or F, evaluates to $\langle E_1, x' : \tau, \text{Subst}\{x := x'\}(E_2) \mid \text{Subst}\{x := x'\}(F)\rangle$. Renaming can also be applied to the names of linear resources. (Recall that such names do not scope over other context entities.) For example, $\text{Rename}\{y := y'\}(\langle E \mid F_1, y : H, F_2\rangle)$ evaluates to $\langle E \mid F_1, y' : H, F_2\rangle$.

Sequence of instructions. Let us return to Figure 20. 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 $(t_1; ...; t_n; r)$. Starting from an input context Γ_0, each subterm t_i makes the context evolves from Γ_{i-1} to Γ_i. Recall from Section 3.1 that each subterm t_i must have unit type (a.k.a. **void** type), else it would have been wrapped into a call to the "ignore" function. The sequence itself may return a value identified by the optional result variable r. If such a result variable is set, the final context is patched to include a **res** binding instead of the original r binding.

Scope blocks. The proof 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 = ref(t) or **let** x = stackAlloc$_C()$, with such instructions occurring directly within a sequence.[21] The auxiliary function StackAllocCells$(t_1, ..., t_n)$ synthesizes, based on the syntax of the terms t_i that appear in the sequence at hand, a separated conjunction of the uninitialized version of all the resources allocated in the sequence $t_1, ..., t_n$. Formally, StackAllocCells and its auxiliary function StackAllocCell are defined as follows.

$$\mathsf{StackAllocCells}(t_1, ..., t_n) = \mathsf{StackAllocCell}(t_1) \star \cdots \star \mathsf{StackAllocCell}(t_n)$$

$$\mathsf{StackAllocCell}(t) = \begin{cases} p \rightsquigarrow C & \text{if } t \text{ is of the form "}\textbf{let } p = \mathsf{stackAlloc}_C()\text{"} \\ p \rightsquigarrow \mathsf{UninitCell}_\tau & \text{if } t \text{ is of the form "}\textbf{let } p = \mathsf{ref}(t')\text{" with } t' : \tau \\ \varnothing & \text{if } t \text{ does not contain a stack allocation} \\ \text{typing error} & \text{if } t \text{ contains a stack allocation not directly bound by a } \textbf{let} \end{cases}$$

These resources described by StackAllocCells$(t_1, ..., t_n)$ are subtracted from the context available at the end of the sequence. Crucially, the subtraction operation checks that the resources indeed appear in the resource set after the execution of the sequence.

We take a conservative approach for pure typing context scopes: when a sequence is exited, each immutable program variable that goes out of scope is generalized as a ghost variable, and all ghost variables introduced in the scope are kept. Generalizing immutable program variables as ghost variable is a no-op in practice since all the program variables are already in the pure context. For now, only contract annotations on functions, loops and on conditionals act as abstraction barriers that filter pure contexts.

Function definition. Consider a function definition $\textbf{fun}(a_1 : \hat{\tau}_1, ..., a_n : \hat{\tau}_n)_\gamma \mapsto t$, with arguments a_i of type $\hat{\tau}_i$, with body t, and with contract γ. Recall from Section 4.2 that the function contract consists of a precondition γ.pre and a postcondition γ.post, both described as contexts. The function is a closure that may capture free variables from the current context. In the rule, the pure variables from the current context are described as Γ_0.pure. Note, however, that the function 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 context Γ_r that entails the postcondition γ.post. The postcondition of the function definition itself binds **res** with the correct function type (**res** : $\hat{\tau}_f$) and gives its specification hypothesis (Spec(**res**, $[a_1, ..., a_n]$, γ)). As explained earlier in Section 4.2, this hypothesis captures $\{\gamma.\text{pre}\}$ **res**$(a_1, ..., a_n)$ $\{\gamma.\text{post}\}$, which is indeed the triple intended for the function named **res**.

Conditionals. Consider a conditional of the form **if** t_1 **then** t_2 **else** t_3. The condition t_1 is evaluated in the input context Γ_0 and produces a context Γ_1. Then, both branches t_2 and t_3 need to typecheck in the context Γ_1. This context needs to be patched to reflect the knowledge that t_1 evaluated to

[21] The implementation of StackAllocCells forces all stack allocations to be directly bound in a **let** instead of being allowed to occur in subexpression. In the latter case, the proof rule fails.

either **true** or **false**, depending on the branch. The patch is implemented by means of the operation $\mathsf{Learn}\{\mathbf{res} = b\}(\Gamma)$. This operation applies the following three steps.

(1) If an aliasing binding of the form $\mathbf{res} := v : \mathsf{bool}$ appears in Γ, then the operation replaces this binding with a conventional binding $\mathbf{res} : \mathsf{bool}$, and extends Γ with an equality $[\mathbf{res} = v]$.
(2) It specializes the variable **res** with b, that is, it removes the binding $\mathbf{res} : \mathsf{bool}$, 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_1 is a test of the form `x == y`, and consider the evaluation of $\mathsf{Learn}\{\mathbf{res} = \mathbf{true}\}(\Gamma_1)$. The output context of t_1 contains the alias binding $\mathbf{res} := (\mathtt{x==y}) : \mathsf{bool}$. At step (1), this alias binding is replaced with an equality $\mathbf{res} = (\mathtt{x==y})$. At step (2), **res** is replaced with **true**, hence the equality becomes $\mathbf{true} = (\mathtt{x==y})$. At step (3), this hypothesis is rewritten as the logical equality `x = y`.

The **then** branch t_2 produces an output context Γ_2, and likewise the **else** branch t_3 produce an output context Γ_3. What should be the output context of the entire conditional $\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$? It must be a context, call it Γ_r, that both Γ_2 and Γ_3 entail. This context Γ_r is usually called the *join* context in program logics. In general, there is no way to automatically infer join contexts—it is almost as hard as inferring contracts for loops. Therefore, typechecking and verification tools must resort to a combination of user-provided annotations and heuristics. For now, we assume join contexts to be provided by the user. In our box-blur case study (Section 2.2), the conditionals appear in terminal position in the body of a function, hence our typechecker can simply instantiate the join context using the (user-provided) postcondition of that function. We leave it to future work to devise heuristics well-suited for our typesystem, in order to reduce the number of situations where OptiTrust users need to provide annotations.

4.7 Typechecking of Function Applications

The proof rule for function application handles the particular case where the subterms are program variables—that is, for function calls in *A-normal form*. The typechecking of effectful subterms depends on *usage maps*, and is thus explained further on, in Section 6.8. Consider thus a function application of the form $x_0(x_1, ..., x_n)$, where the x_i are program variables. Its proof rule is as follows.

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

The input context Γ_0 must contain a pure entry of the form $\mathsf{Spec}(x_0, [a_1, ..., a_n], \gamma)$ for the function x_0. This same context Γ_0 must entail the precondition $\gamma.\mathsf{pre}$, instantiated for the arguments x_i provided to the function. This instantiation is performed by means of the operation Specialize, which is detailed shortly afterwards. The entailment is checked by means of the carving subtraction operation defined in Section 4.5. This 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 $\gamma.\mathsf{post}$, adding the frame F and $\Gamma_0.\mathsf{pure}$, then invoking the CloseFracs operation described in Section 4.1 for eagerly recombining carved-out fractions. The role of σ and ρ, which corresponds to optional user-provided annotations, is explained next. (Such annotations are commonly found both in proof assistants and in program verification frameworks.)

The map σ allows 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 give a name to all the resources that are added to the context by the postcondition of the called function, thus avoiding name conflicts. In practice, the map ρ is partially given by user annotations or by transformations and is extended with fresh variable names for each missing entry during the first typechecking of each function application.

Specialization of contexts. The operator Specialize instantiates the precondition of a function on specific arguments. In case of a polymorphic function, type arguments are instantiated as well. The specialization operation is a partial function written $\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$. Its implementation essentially matches the standard process of typechecking function applications in higher-order, dependently typed theory, only extended to also take into account substitutions in linear resources. The technical details are described in appendix E. Here, we present a representative example.

Consider a function f whose input is described by a context $\Gamma = \langle A : \mathsf{Type},\ C : \mathsf{Type},\ n : \mathsf{int},\ p : \mathsf{ptr}(A),\ b : A,\ c : C \mid p \rightsquigarrow \mathsf{Matrix1}_A(n)\rangle$, 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 $\sigma = (n := 7,\ p := q)$. 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: $\langle C : \mathsf{Type},\ b : \mathsf{int},\ c : C \mid q \rightsquigarrow \mathsf{Matrix1}_{\mathsf{int}}(7)\rangle$. Observe how the types and arguments that are being specialized (namely A, n and p) are eliminated from the pure part of the context, and how the corresponding values (namely int, 7 and q) are substituted in the entities that remain.

4.8 Typechecking of For-Loops

Our rule handling **for** loops is the most technical. Intuitively, it combines the expressiveness of two standard separation logic rules: the rule for sequential loops and the rule for parallel loops. These two rules are stated below, and explained next.

$$\frac{\{[E_0,\ i : \mathsf{int},\ i \in R] \circledast \Gamma_{\mathsf{inv}}(i)\}\ t\ \{\Gamma_{\mathsf{inv}}(i + R.\mathsf{step})\}}{\{[E_0] \circledast \Gamma_{\mathsf{inv}}(R.\mathsf{start})\}\ \mathbf{for}^{\mathbf{seq}}\ (i \in R)\ t\ \{\Gamma_{\mathsf{inv}}(R.\mathsf{end})\}}\ \textsc{ForSeq}$$

$$\frac{\{[E_0,\ i : \mathsf{int},\ i \in R] \circledast \Gamma_{\mathsf{pre}}(i)\}\ t\ \{\Gamma_{\mathsf{post}}(i)\}}{\{[E_0] \circledast \circledast_{i \in R} \Gamma_{\mathsf{pre}}(i)\}\ \mathbf{for}^{\mathbf{par}}\ (i \in R)\ t\ \{\circledast_{i \in R} \Gamma_{\mathsf{post}}(i)\}}\ \textsc{ForPar}$$

The sequential rule involves a traditional *loop invariant*, written $\Gamma_{\mathsf{inv}}(i)$, describing the state at the start of the i-th iteration. Consider a loop that iterates from 0 to n by steps of 1. The whole loop consumes $\Gamma_{\mathsf{inv}}(0)$ and ultimately produces $\Gamma_{\mathsf{inv}}(n)$. To typecheck the body, we need to prove that, for any i in the range $0..n$, the body consumes $\Gamma_{\mathsf{inv}}(i)$ and produces $\Gamma_{\mathsf{inv}}(i + 1)$.

The parallel rule involves a *per-iteration contract*, expressed by the pair of $\Gamma_{\mathsf{pre}}(i)$ and $\Gamma_{\mathsf{post}}(i)$. This contract explains what each iteration, taken independently of all the other iterations, consumes and produces. The resources associated with each of the parallel iterations must be disjoint, to guarantee that the iterations can be safely scheduled on concurrent threads. If the parallel loop iterates i over the range $0..n$, the whole loop consumes $\circledast_{i \in 0..n} \Gamma_{\mathsf{pre}}(i)$ and produces $\circledast_{i \in 0..n} \Gamma_{\mathsf{post}}(i)$.

For example, if the i-th iteration operates on an array cell described by $(p \boxplus \mathsf{mIndex1}(n, i)) \mapsto M(i)$, then the resource $\circledast_{i \in 0..n} \Gamma_{\mathsf{pre}}(i)$ corresponds to the entire array $p \rightsquigarrow \mathtt{Matrix1(}n\mathtt{,}\ M\mathtt{)}$. In

general, however, $\Gamma_{\text{pre}}(i)$ may involve ghost variables. In such case, the interpretation of the iterated star over contexts is more subtle—we give details further on.

The parallel rule allows for all the parallel iterations to *read* in the same shared resources. To achieve this, a $\frac{1}{n}$ fraction of a read permission on a resource H may be provided to each of the n iterations, as part of $\Gamma_{\text{pre}}(i)$ and $\Gamma_{\text{post}}(i)$. In OptiTrust, we find it more convenient for contract manipulations to provide dedicated support for tracking the read-only resources that the whole loop obtains and distributes over to each of the iterations.

We show below the OptiTrust rule for a loop of the form $\mathbf{for}^{\pi}\ (i \in R)_{\chi}\ t$, subsuming both the sequential rule and the parallel rule. The expression R.start corresponds to the first index in the range R, R.step corresponds to the increment between two consecutive indices of the range R, and R.end corresponds to the one-past-end index of the range R. Note that this R.end can differ from the stop index written in the source code denoted by R.stop. For instance, $\mathbf{range}(0, 3, 2)$.stop = 3 but $\mathbf{range}(0, 3, 2)$.end = 4.

The loop contract is written χ. The entity χ.shrd.inv corresponds to Γ_{inv}, the entity χ.excl.pre corresponds to Γ_{pre}, the entity χ.excl.post corresponds to Γ_{post}, and the entity χ.shrd.reads corresponds to a the read-only resources that every iteration can access. In addition, χ.vars is used to quantify ghost variables.

$$\frac{\begin{array}{c}\Gamma^{\text{out}}_{\text{pre}} = [\chi.\text{vars}] \circledast (\circledast_{i \in R}\, \chi.\text{excl.pre}) \circledast \chi.\text{shrd.reads} \circledast \text{Subst}\{i := R.\text{start}\}(\chi.\text{shrd.inv}) \\ (E_{\text{frac}}, \sigma^{\text{out}}, F) = \Gamma_0 \ominus \Gamma^{\text{out}}_{\text{pre}} \\ \Gamma^{\text{in}}_{\text{pre}} = [\chi.\text{vars}] \circledast \chi.\text{excl.pre} \circledast \tfrac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv} \\ \{\!\{ [\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \Gamma^{\text{in}}_{\text{pre}} \}\!\}\ t\ \{\!\{ \Gamma^{\text{in}}_{\text{post}} \}\!\} \\ (\sigma^{\text{in}}_{\text{post}}, \varnothing) = \Gamma^{\text{in}}_{\text{post}} \boxminus \chi.\text{excl.post} \circledast \tfrac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \text{Subst}\{i := i + R.\text{step}\}(\chi.\text{shrd.inv}) \\ \Gamma^{\text{out}}_{\text{post}} = \text{Subst}\{\sigma^{\text{out}}\}((\circledast_{i \in R}\, \chi.\text{excl.post}) \circledast \chi.\text{shrd.reads} \circledast \text{Subst}\{i := R.\text{end}\}(\chi.\text{shrd.inv})) \\ \Gamma_r = \text{CloseFracs}([\Gamma_0.\text{pure}, E_{\text{frac}}] \circledast F \circledast \Gamma^{\text{out}}_{\text{post}}) \\ \pi = \mathbf{par} \implies \text{parallelizable}(\chi)\end{array}}{\{\!\{ \Gamma_0 \}\!\}\ \mathbf{for}^{\pi}\ (i \in R)_{\chi}\ t\ \{\!\{ \Gamma_r \}\!\}}\ \textsc{For}$$

The input context of the loop Γ_0 must contain the per-iteration preconditions ($\circledast_{i \in R}\, \chi.\text{excl.pre}$), the read resources ($\chi$.shrd.reads), and the sequential invariant for the start index (χ.shrd.inv instantiated with $i := R$.start). Symmetrically, the output context of the loop Γ_r contains the per-iteration postconditions ($\circledast_{i \in R}\, \chi.\text{excl.post}$), the read resources ($\chi$.shrd.reads), and the sequential invariant for the stop index (χ.shrd.inv instantiated with $i := R$.end).

The substitution σ^{out} here plays the same role as σ' in the App rule from the previous section: it describes the instantiation of the ghost arguments and resources, when matching the set of resources available before reaching the loop against the set of resources requested by the loop. We omit details concerning the quantification of ghost variables described by χ.vars, the management of fresh fractions described by E_{frac}, and the call to CloseFracs for merging back carved fractions.

The body t of the loop is typechecked in a context that binds an index i of type int, a hypothesis of type $i \in R$, the per-iteration preconditions for the current index i (that is, χ.excl.pre), the read resources (a fraction $\frac{1}{R.\text{len}}$ of χ.shrd.reads;[22] from the user's perspective, this fraction provides essentially the same permissions as χ.shrd.reads in whole), and the sequential invariant for the current index (χ.shrd.inv). Symmetrically, the postcondition of the body contains the per-iteration

[22] If χ.shrd.reads takes the form $(y_0 : H_0, \ldots, y_n : H_n)$, then a fraction α of this resource set is defined as $(y_0 : \alpha H_0, \ldots, y_n : \alpha H_n)$. Besides, the writing of $\frac{1}{R.\text{len}}$ is justified by the fact that we can assume R.len to be nonzero when typechecking of the loop body. Indeed, if R.len = 0, then the assumption $i \in R$, which is defined $0 \le i < R$.len, is equivalent to False.

postconditions (χ.excl.post), the read resources (again, $\frac{1}{R.\mathsf{len}}$ of χ.shrd.reads); and the sequential invariant for the next index (χ.shrd.inv with i instantiated as $i + R.\mathsf{step}$).

Iterated separating conjunction of a context. There remains to describe one technicality, namely the interpretation of $\circledast_{i\in R}\,\Gamma(i)$, a pattern that appears in $\circledast_{i\in R}\,\chi.\mathsf{excl.pre}$ and $\circledast_{i\in R}\,\chi.\mathsf{excl.post}$. For the linear part of $\Gamma(i)$, the separating conjunction simply distributes: $\bigstar_{i\in R}\,\langle|y_0 : H_0, \ldots, y_n : H_n\rangle$ is defined as $\langle|y_0 : (\bigstar_{i\in R}\,H_0), \ldots, y_n : (\bigstar_{i\in R}\,H_n)\rangle$. For the pure part, the situation is more complicated, because we need to transform certain variables quantified in $\Gamma(i)$ into functions of i. Consider the example $\Gamma_i = \langle n : \mathsf{int}, P : 0 \le n < 9 \mid (p \boxplus i) \mapsto n\rangle$, which asserts that there exists an integer n in the range 0..9 such that $p \boxplus i$ points to n. The conjunction $\circledast_{i\in R}\,\Gamma_i$ needs to express the fact that, for each i in R, there exists an integer n_i in the range 0..9 such that $p \boxplus i$ points to n_i. The context $\circledast_{i\in R}\,\Gamma_i$ is thus equal to: $\langle n : \mathsf{int} \xrightarrow{\text{logic}} \mathsf{int}, P : (\forall(i : \mathsf{int})(x_R : i \in R), 0 \le n(i) < 9) \mid \bigstar_{i\in R}\,(p \boxplus i) \mapsto n(i)\rangle$.

We formalize the rules describing the computation of a separating conjunction over the pure part of a context as shown below. The definition discriminates on whether variables have a type that is *obviously inhabited*, e.g., a program type or a basic logical type. For those, the variable is turned into a logical function, thereby avoiding the need for manipulating dependently typed functions. For example, $n : \mathsf{int}$ is turned into $n : \mathsf{int} \xrightarrow{\text{logic}} \mathsf{int}$. For other variables, typically variables whose type is a proposition, we replace the type τ with the type $\forall(i : \mathsf{int})(x_R : i \in R), \tau$. For example, $P : \ 0 \le n(i) < 9$ is turned into $P : \ \forall(i : \mathsf{int})(x_R : i \in R), 0 \le n(i) < 9$.

$$\bigcircledast_{i\in R}^{x_R} \langle x : \tau, E \mid F\rangle = \begin{cases} [x : \tau] \circledast \bigcircledast_{i\in R}^{x_R} \langle E \mid F\rangle & \text{if } x \text{ does not occur in } \langle E \mid F\rangle \text{ and } i \text{ not occur in } \tau \\ [x : \mathsf{int} \xrightarrow{\text{logic}} \tau] \ \circledast \ \bigcircledast_{i\in R}^{x_R} \mathsf{Subst}\{x := x(i)\}(\langle E \mid F\rangle) & \text{if } \tau \text{ is obviously inhabited} \\ [x : \forall(i : \mathsf{int})(x_R : i \in R), \tau] \ \circledast \ \bigcircledast_{i\in R}^{x_R} \mathsf{Subst}\{x := x(i, x_R)\}(\langle E \mid F\rangle) & \text{otherwise} \end{cases}$$

5 PROOF-PRESERVING TRANSFORMATIONS

In OptiTrust, we are interested in specification-preserving transformations: a transformation may replace a term that satisfies a given triple with any other term that satisfies the same triple. Formally, if t_1 is a subterm (including code and separation logic annotations) of the input program that typechecks in OptiTrust as $\{\Gamma\}\ t_1\ \{\Gamma'\}$, then t_1 may be replaced with another term t_2 that typechecks in OptiTrust as $\{\Gamma\}\ t_2\ \{\Gamma'\}$. In practice, we simply enforce that the specifications associated with the top-level functions do not change. The body of these functions may change arbitrarily, but if at the end of the transformation script the updated body still typechecks against the same specification, then the optimized code is correct.

In this section, we present representative examples of transformations that we have implemented in OptiTrust. Transformations fall in two categories: the *basic* transformations that directly modify the AST, and the *combined* transformations that are defined as the composition of basic transformations. We will mainly focus on presenting *basic* transformations.

For some transformations, we document the existence of sufficient conditions under which the transformation is guaranteed to be proof-preserving—i.e., for which the output code will certainly typecheck. When the sufficient condition documented for a transformation is not satisfied by the input code, the transformation can be viewed as a best-effort at achieving proof-preservation: depending on the code at hand, the transformation may still result in a well-typed program. Indeed, there exists legitimate optimizations that are either too specific or too complex to be captured by a general-purpose sufficient condition. Finally, if the output code does not typecheck, it might be the case that this code is nonetheless correct w.r.t. the specification. In that case, the user needs to repair the proof (or to apply subsequent transformations towards reaching a code on which

such a repair is possible). The ability to break and repair typechecking is a notable strength of the proof-carrying approach in contrast with the transitive semantics-preservation approach.

In this section, we first introduce the notation used for describing transformations. We then present the deceptively simple inlining and rewriting transformations, which illustrate well the basics of proof-preservation. Next, we present key transformations, including loop tiling, loop interchange, and loop hoisting, as well as a transformation for introducing temporary alternative storage. We will also present transformations on annotations, which are often necessary for making the code transformations applicable. We delay the presentation of the loop fission transformation to Section 6, that introduces the additional concept of usage maps. We omit from the presentation other transformations implemented in OptiTrust, including loop fusion (reciprocal of fission), loop collapse (reciprocal of tiling), loop unrolling, as well as loop-range manipulations (shift, scale, extend).

5.1 Notation for Transformations

Transformations apply to instructions or groups of instructions. They depend on typing context information. They produce code with possibly updated loop contracts, and possibly including new 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, establishing triples of the form $\{\!\{\Gamma_1\}\!\}\ t\ \{\!\{\Gamma_2\}\!\}$. In this section, we use an alternative syntax, better-suited for describing the input of transformations. If t denotes an instruction, we write $\Gamma_1\ t;\ \Gamma_2$ as straight-line syntax for $\{\!\{\Gamma_1\}\!\}\ t\ \{\!\{\Gamma_2\}\!\}$, where any of Γ_1 or Γ_2 can be omitted if not needed by the transformation. More generally, we write $\Gamma_1\ T;\ \Gamma_2$, where T denotes a (possibly empty) group of instructions.

Program contexts. Transformations generally apply to a program subterm, that is, apply under a *program context*. Unlike evaluation contexts, program contexts can reach subterms that are not in evaluation position. We let the meta-variable $\mathcal{E}$ range over program contexts. For example, evaluating a subexpression $1 + 1$ that appears in a program 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 $t_1; t_2$.

In general, a transformation is applicable in any program context. We write $t \leftrightarrow t'$ to mean that $\mathcal{E}[t]$ can be replaced with $\mathcal{E}[t']$ for any program context $\mathcal{E}$.

Evaluation contexts. Some transformations operate on subexpressions that appear inside an instruction. For those, we may state sufficient proof-preservation criteria by restricting program contexts to rule out nontrivial control-flow arising from, e.g., a conditional. Recall from Section 6.8 that an *evaluation context*, written $\hat{\mathcal{E}}$, denotes a program context whose holes (possibly just one) are in evaluation position. For example, $f(g(\square, 2), g(3, a + 4))$ is an evaluation context with a single hole written $\square$. For any evaluation context $\hat{\mathcal{E}}$, the following program equivalence is key.

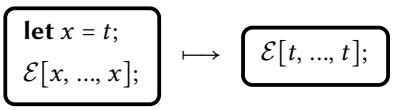
If $\hat{\mathcal{E}}[t]$ is well-typed, then $\boxed{\hat{\mathcal{E}}[t]} \leftrightarrow \boxed{\begin{array}{l}\textbf{let } x = t;\\ \hat{\mathcal{E}}[x]\end{array}}$

The proof-preservation argument relies on the hypothesis that the input code typechecks against our proof rules. Indeed, the SUBEXPR rule ensures 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 **let** $x = g_2()$; $f(g_1(), x, g_3())$ because, if the former term is well-typed, then the effects of $g_2()$ 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 context available. In the output of transformations, we write **ghost**$(\Gamma \longrightarrow \Gamma')$ to mean the insertion of an appropriate ghost call $g()$, such that g admits Γ as precondition, and that the call to g adds the resources in Γ' to the context.[23]. Concretely, the effect of **ghost**$(\Gamma \longrightarrow \Gamma')$ is to consume the resources Γ then to produce the resources Γ'.

5.2 Inlining

Let us now get familiar with the idea of proof-preserving transformation by discussing a simple transformation that inlines a variable binding of the form **let** $x = t$ throughout the subsequent instructions. This transformation is described as shown below, where $\mathcal{E}$ denotes a multi-hole program context. Each hole corresponds to one occurrence of the variable x. The transformation replaces all these occurrences of x with a copy of t.



The key question is: *is this transformation proof-preserving?* In other words, if we assume the input code to be annotated with separation logic annotations in a way that yields a valid derivation, then is it the case that the output code still yields a valid derivation? Let us first discuss the case where t corresponds to a logical expression (as defined in Section 4.4), then explain how the transformation needs to be refined if t involves side effects.

If t denotes a logical expression, then the transformation needs to replace all occurrences of x with t not only throughout the code, but also throughout all proof annotations (e.g., loop contracts, ghost arguments). The validity of the derivation is then guaranteed to be preserved. To see why, recall from Figure 20 that **let** $x = t$ introduces an alias-binding from x to t in the typing context. This binding means that, during any unification task involved in the typechecking process, x is eagerly replaced by t. Therefore, if we replace x by t throughout the code and its annotations, the typechecker will perform exactly the same unifications as in the original code. In summary, inlining of logical expressions is always proof-preserving.

Consider now the more general situation where t may perform side effects. When we inline **let** $x = t$, by what should we replace occurrences of x inside logical expressions, i.e., the *logical occurrences* of x? We cannot use t as it is not a logical expression. Instead, we need to inspect the typechecking derivation of the input code. The context immediately after the instruction **let** $x = t$ must include an alias binding of the form $[x := v]$, where v denotes the logical result of t. The inlining transformation needs to substitute all logical occurrences of x with v. The transformation can be summarized as follows, where $\overline{x}$ denotes the logical occurrences of x in a context $\mathcal{E}$.

[23]Technically, the ghost function g admits a context Γ'' as its postcondition and the call inserted by the transformation specifies a renaming map ρ such that $\Gamma' = \text{Rename}\{\rho\}(\Gamma'')$ Recall that these renaming maps are used by the proof rule APP described in Section 4.6.

$$\boxed{\begin{array}{l}\textbf{let } x = t;\\ \Gamma \circledast [x := v]\\ \mathcal{E}[x, ..., x, \overline{x}, ..., \overline{x}]\end{array}} \longmapsto \boxed{\mathcal{E}[t, ..., t, \overline{v}, ..., \overline{v}];}$$

When is this transformation semantic-preserving? Observe that in the output code, the term t is not evaluated at the same place as in the input code, traversing other operations. Moreover, it may be evaluated a different number of times (possibly zero or several times). Typical analysis-based approaches would have to check whether t can be duplicated (i.e. is t *idempotent*), deleted (i.e. are the effects of t *observable*), and commutes with the traversed operations.

Interestingly, with OptiTrust's validation approach based on proof-carrying code, there is no need to devise such complex criteria and analyses. It suffices to typecheck the output code. If typechecking is successful, then the output code satisfies the same specification as the input code, hence the transformation is legitimate.

Bonus: inlining of function calls. To inline a call, we first wrap it inside a local block, in which we introduce explicit variable bindings for every function argument, using the same names as in the function declaration. Then, we inline the function body. As an optional post-processing, the user may rename the introduced variables or inline certain bindings.

5.3 Rewriting

Another simple transformation that requires particular care for proof-preservation is rewriting a subterm into an equivalent one. Consider for example the simplification of $1 + 2$ into 3. Suppose here that the context contains an hypothesis named H that corresponds to a proof of $1 + 2 = 3$. The code transformation that we consider is thus as follows.

$$\boxed{\begin{array}{l}\Gamma \circledast [H : 1 + 2 = 3]\\ \mathcal{E}[1 + 2];\end{array}} \longmapsto \boxed{\mathcal{E}[3];}$$

Yet, there are situations where simplifying a well-typed input code this way would lead to an ill-typed output code. The problem stems from the fact that the typechecking of certain subexpressions from $\mathcal{E}$ might rely on pattern matching against, e.g., $1 + X$ for some pattern variable X. Such a matching succeeds on the expression $1 + 2$, but fails on the expression 3. Indeed, our typechecker, following standard practice, compares terms modulo *unification* and not modulo *equality*.

Our solution to ensure proof-preservation in general is to wrap every occurrence of 3 introduced in $\mathcal{E}[3]$ within an identity function that exposes 3 as an expression that, from a logical perspective, returns $1 + 2$. This function, $\mathsf{view}(x)$, requires a ghost argument y and a proof of the form $H : x = y$, and produces **res** $:= y$ as an alias binding.

$$\boxed{\begin{array}{l}\Gamma \circledast [H : 1 + 2 = 3]\\ \mathcal{E}[1 + 2];\end{array}} \longmapsto \boxed{\mathcal{E}[\mathsf{view}(3,\ y := 1 + 2,\ eq := H)];}$$

Technically, we define $\mathsf{view}(x)$ as a function with ghost arguments y and $eq := y = x$ and with the body: **let** $z = x$; **ghost**$([z := x] \longrightarrow [z := y])$; z.

Certain calls to the view function can be subsequently eliminated by means of inlining. Consider for example a specific occurrence of $\mathsf{view}(3)$ from the output program. If we inline this call to view, we obtain: $\{$**let** $z = 3$; **ghost**$([z := 3] \longrightarrow [z := 1 + 2])$; $z\}$. Then, if we further inline **let** $z = 3$, and delete the ghost operation, we obtain 3 at the place where we previously had $1 + 2$. Note, however,

that if the expression view(3) is involved directly or indirectly in a matching against the pattern $1 + X$, then the call to view cannot be so easily eliminated.

Many of the more complex transformations include rewriting subterms with equivalent ones, and need to deal with such proof-preservation concerns.

5.4 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 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 R_i, R_j and R_k, respectively. Recall that a range is of the form $\mathbf{range}(start, stop, step)$. The notation $start..stop$ is a shorthand for $\mathbf{range}(start, stop, 1)$. In particular, $0..n$ describes the range of values from 0 inclusive to n exclusive.

This transformation, like all other loop transformations, needs to produce appropriate contracts for the loops being produced. 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, as materialized before and after the produced loops in the figure. Indeed, the loop on i consumes, in particular, the resource $\bigstar_{i \in R_i} \chi_i$.excl.pre whereas the loop on j consumes instead $\bigstar_{j \in R_j} \bigstar_{k \in R_k} \chi_k$.excl.pre.

5.5 Loop Interchange (a.k.a. Swap)

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. all pairs of iterations i, j and i', j' with $i' > i$ and $j > j'$. We have not yet investigated how loop contracts need to be updated to handle such dependencies. For now, we have implemented automatic generation of valid loop contracts for the cases where at least one of two loops involved is parallelizable. 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 adjustments.

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 fact that our typechecker stores contract instantiations to compute partitions on χ_j.excl.pre and χ_j.shrd, and we rely on the subtraction operation $\boxminus$ to partition χ_j.excl.post.

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

$$\boxed{\begin{array}{l}\textbf{for}\ (i \in R_i)_{\chi_i}\ \{\\ \quad T;\\ \}\end{array}} \longmapsto$$

$$\boxed{\begin{array}{l}
\textbf{ghost}(\star_{i\in R_i}\chi_i.\text{excl.pre} \longrightarrow \star_{j\in R_j}\star_{k\in R_k}\chi_k.\text{excl.pre});\\
\textbf{ghost}(\text{Subst}\{i := 0\}(\chi_i.\text{shrd.inv}) \longrightarrow \text{Subst}\{j := 0\}(\chi_j.\text{shrd.inv}));\\
\textbf{for}\ (j \in R_j)_{\chi_j}\ \{\\
\quad \textbf{for}\ (k \in R_k)_{\chi_k}\ \{\\
\quad\quad \textbf{ghost}(\langle|\rangle \longrightarrow [\text{RecoverIndex}(j,k) \in R_i]);\\
\quad\quad T';\\
\quad\quad \textbf{ghost}(\text{Subst}\{i := \text{RecoverIndex}(j,k)+1\}(\chi_i.\text{shrd.inv}) \longrightarrow\\
\quad\quad\quad \text{Subst}\{k := k+1\}(\chi_k.\text{shrd.inv}));\\
\quad \}\\
\quad \textbf{ghost}(\text{Subst}\{k := R_k.\text{end}\}(\chi_k.\text{shrd.inv}) \longrightarrow\\
\quad\quad \text{Subst}\{j := j + R_j.\text{step}\}(\chi_j.\text{shrd.inv}));\\
\}\\
\textbf{ghost}(\text{Subst}\{j := R_j.\text{end}\}(\chi_j.\text{shrd.inv}) \longrightarrow \text{Subst}\{i := R_i.\text{end}\}(\chi_i.\text{shrd.inv}));\\
\textbf{ghost}(\star_{j\in R_j}\star_{k\in R_k}\chi_k.\text{excl.post} \longrightarrow \star_{i\in R_i}\chi_i.\text{excl.post});
\end{array}}$$

where:

$$T' = \text{Subst}\{i := \text{RecoverIndex}(j,k)\}(T)$$
$$\chi_k = \text{Subst}\{i := \text{RecoverIndex}(j,k)\}(\chi_i)$$
$$\chi_j = \begin{cases}\text{vars} = \chi_i.\text{vars}\\ \text{shrd} = \text{Subst}\{k := R_k.\text{start}\}(\chi_k.\text{shrd})\\ \text{excl} = \{\text{pre} = \star_{k\in R_k}\chi_k.\text{excl.pre};\ \text{post} = \star_{k\in R_k}\chi_k.\text{excl.post}\}\end{cases}$$

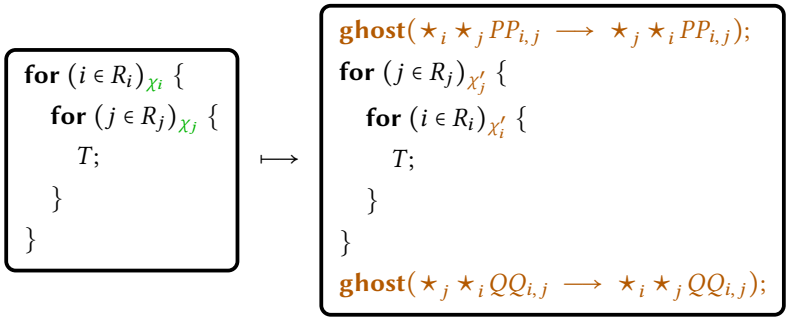
with the following possible instantiations for the ranges:

Range R_i	Range R_j	Range R_k	Formula for recovering i: RecoverIndex(j,k)
$0..(m \times b)$	$0..m$	$0..b$	$j * b + k$
$0..n$ where b divides n	$0..(n/b)$	$0..b$	$j * m + k$
$0..n$ where b divides n	**range**$(0, n, b)$	$j..j+b$	k
$0..n$	**range**$(0, n, b)$	$j..\min(j+b, n)$	k

Fig. 21. The basic transformation `Loop.tile`, with its 4 variants.

5.6 Loop Invariant Code Motion, a.k.a. Loop Hoisting

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 code that first executes T_1, exactly once, then executes a loop with body T_2. The transformation is formalized in Figure 23.



for the case where parallelizable(χ_i), that is, χ_i.shrd.inv = $\varnothing$.

The contracts from the input code are decomposed as follows:

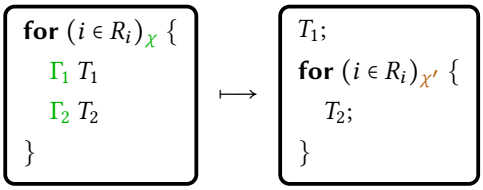
$$\chi_i.\text{shrd} = \{\text{inv} = \varnothing, \text{reads} = (\star_j PR_j \star IR \star RR)\}$$
$$\chi_i.\text{excl} = \{\text{pre} = (\star_j PP_{i,j} \star IP_{i,R_j.\text{start}} \star RP_i), \text{post} = (\star_j QQ_{i,j} \star IP_{i,R_j.\text{end}} \star RP_i)\}$$
$$\chi_j.\text{shrd} = \{\text{inv} = (IP_{i,j} \star IR), \text{reads} = (RP_i \star RR)\}$$
$$\chi_j.\text{excl} = \{\text{pre} = (PP_{i,j} \star PR_j), \text{post} = (QQ_{i,j} \star PR_j)\}$$

The contracts for the output code are built as follows:

$$\chi'_j = \begin{cases} \text{vars} = \chi_i.\text{vars}, \chi_j.\text{vars} \\ \text{shrd} = \{\text{inv} = (\star_i IP_{i,j} \star IR), \text{reads} = (\star_i RP_i \star RR)\} \\ \text{excl} = \{\text{pre} = (\star_i PP_{i,j} \star PR_j), \text{post} = (\star_i QQ_{i,j} \star PR_j)\} \end{cases}$$

$$\chi'_i = \begin{cases} \text{vars} = \chi_i.\text{vars}, \chi_j.\text{vars} \\ \text{shrd} = \{\text{inv} = \varnothing, \text{reads} = (PR_j \star IR \star RR)\} \\ \text{excl} = \{\text{pre} = (PP_{i,j} \star IP_{i,j} \star RP_i), \text{post} = (QQ_{i,j} \star IP_{i,j+R_j.\text{step}} \star RP_i)\} \end{cases}$$

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



where:

$$\_, I' = \Gamma_2 \boxminus \chi.\text{excl.pre}$$
$$\_, I = I' \boxminus \chi.\text{shrd.reads}$$
$$\chi' = \begin{cases} \text{vars} = \chi.\text{vars} \\ \text{shrd} = \{\text{inv} = I,\ \text{reads} = \chi.\text{shrd.reads}\} \\ \text{excl} = \chi.\text{excl} \end{cases}$$

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

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.

5.7 Temporary Alternative Storage

The transformation `Variable.local_name` is the most interesting that we have implemented in terms of operations on plain sequences of instructions. The transformation 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.

$$\boxed{\begin{array}{l}\Gamma_1\\ \mathcal{E}[x, ..., x];\\ \Gamma_2\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } y = \mathsf{stackAlloc}_{\mathsf{Cell}}();\\ T_1;\\ \mathcal{E}[y, ..., y];\\ T_2;\end{array}}$$

where $\mathcal{E}$ is a multi-hole program context with one hole per occurrence of x, and where:

$$T_1 = \begin{cases}\mathsf{set}(y, \mathsf{get}(x)); & \text{if } x \rightsquigarrow v \text{ or } \alpha(x \rightsquigarrow v) \text{ appears in } \Gamma_1\\ \varnothing & \text{if } x \rightsquigarrow \mathsf{UninitCell} \text{ appears in } \Gamma_1\end{cases}$$

$$T_2 = \begin{cases}\mathsf{set}(x, \mathsf{get}(y)); & \text{if } x \rightsquigarrow v \text{ appears in } \Gamma_2\\ \varnothing & \text{if } x \rightsquigarrow \mathsf{UninitCell} \text{ or } \alpha(x \rightsquigarrow \mathsf{Cell}) \text{ appears in } \Gamma_2\end{cases}$$

Fig. 24. The basic transformation `Variable.local_name`.

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 24. There, the group of instructions T is represented as $\mathcal{E}[x, ..., x]$, i.e. as a program context with multiple occurrences of x. The typing context Γ_1 describes the resources available before T, and Γ_2 the resources available after T.

5.8 Transformations on Annotations

Motivating example for transforming annotations. Sometimes ghost instructions or insufficiently precise contracts can prevent the successful application of a transformation. For example, take the following code that could have been generated by tiling the loop over index `i`.

```
for (int j = 0; j < 64; j++) {
  __xwrites("for i in 0..1024 → &A[j, i] ↦ 0");
  __ASSERT(tile_div_check_i, "1024 == 64*16");
  __ghost(tile_divides, "div_check := tile_div_check_i");
  for (int bi = 0; bi < 64; bi++) {
    __xwrites("for i in 0..16 → &A[j, bi*16 + i] ↦ 0");
    // ...
  }
  __ghost(untile_divides, "div_check := tile_div_check_i");
}
```

Let us suppose that the user wants to interchange the loops over `bi` and `j`. In that case, a direct application of the basic transformation `Loop.swap` presented before fails because the input code is not in the right shape consisting of two immediately nested loops. Indeed, there are ghost instructions before and after the loop over `bi`. In order to actually apply such `Loop.swap` transformation, one needs to somehow move the ghost instructions outside the loop over `j`. This operation is non-obvious because some of these ghost instruction depend on the loop index `j`.

This kind of issue with misplaced annotations is very common whenever transformations are chained together like in OptiTrust. Therefore, we developed a set of auxiliary transformations

to deal with such cases, and hide those apparent issues for the user. In practice, for example, the combined transformation version of `Loop.swap` succeeds on the previous example.

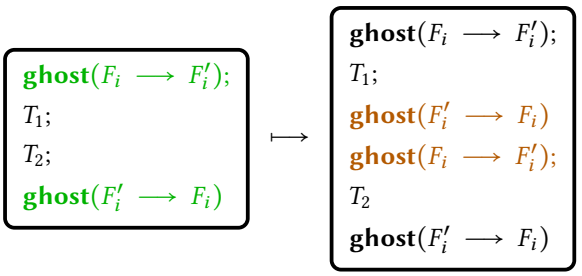
In this particular example, the combined version of `Loop.swap` first applies the basic transformations `Loop.fission` and `Loop.move_out` on the ghost instructions to generate the following intermediate code where the loops over `j` and `bi` can be swapped.

```
__ASSERT(tile_div_check_i, "1024 == 64 * 16");
for (int j = 0; j < 64; j++) {
__xconsumes("for i in 0..1024 → &A[j, i]⇝UninitCell");
__xproduces("for bi in 0..64 → for i in 0..16 → &A[j, bi*16+i]⇝UninitCell");
__ghost(tile_divides, "div_check := tile_div_check_i");
}
for (int j = 0; j < 64; j++) {
  __xwrites("for bi in 0..64 → for i in 0..16 → &A[j, bi*16 + i] ↦ 0");
  for (int bi = 0; bi < 64; bi++) {
    __xwrites("for i in 0..16 → &A[j, bi*16 + i] ↦ 0");
    // ...
  }
}
for (int j = 0; j < 64; j++) {
__xconsumes("for bi in 0..64 → for i in 0..16 → &A[j, bi*16 + i] ↦ 0");
__xproduces("for i in 0..1024 → &A[j, i] ↦ 0");
__ghost(untile_divides, "div_check := tile_div_check_i");
}
```

This example illustrates how transformations on annotations enable combined transformations to apply more basic ones. The rest of this section presents several examples of transformations that modify annotations.

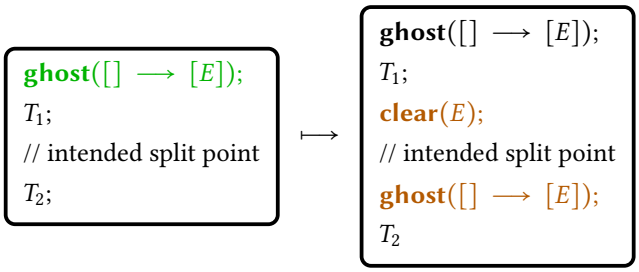
Splitting ghost scopes. Ghost instructions are often used to *temporarily* change the view on a resource, that is, to change it over a given *scope* within a sequence. For instance, if we have a permission to read an entire array, we may want to temporarily extract the permission to read a specific cell, and recompose a permission over the full array later. This *ghost pairs* pattern is materialized by the syntax **`__ghost_begin`** and **`__ghost_end`**. We call *ghost scope* the region delimited by pairs of reciprocal ghost instructions.

To apply certain transformations, it may be necessary to split a ghost scope into two independent ghost scopes. In the example show below, the ghost pair around the instructions T_1 and T_2 can be turned into two independent ghost pairs, one around T_1, and another around T_2. In the latter form, both T_1 and T_2 may be manipulated together with their ghost pair, e.g., for an instruction move, or a loop hoisting operation.



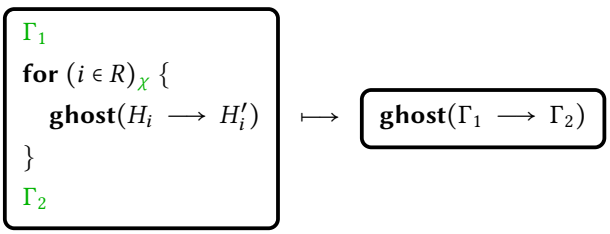
Splitting pure ghost scopes. Certain ghost instructions produce only pure resources—these ghosts correspond to the invocation of lemmas. By default, a pure resource scopes throughout the rest of

the sequence in which it appears. However, it may be interesting to restrict that scope. For example, to perform loop fission, one needs to separate a loop body in two entirely independent parts. We therefore developed a transformation that introduces a clear operation and that duplicates a lemma invocation in order to restrict the scope of pure resources.



Moving and cancelling ghost operations. As explained earlier, to unlock certain transformations, it may be needed to move ghost instructions so that they (logically) execute as early as possible in the program; or, symmetrically, execute as late as possible. In certain situations, moving ghost instructions in such a way may lead to the appearance of *cancellable* pairs of ghost instructions. Indeed, the sequence $\mathbf{ghost}(H \longrightarrow H')$; $\mathbf{ghost}(H' \longrightarrow H)$ is equivalent to a no-op. OptiTrust includes a transformation that attempts to remove pairs of ghost instructions wherever possible.

Loop over ghost instructions. Some transformations may produce loops that only contain ghost instructions. To capture the fact that such loops need not be executed at runtime, we developed a transformation that replaces such loops with a single ghost instruction that has the same effect. The pattern is as follows, where the body of the loop may contain any number of ghost operations.



6 USAGE MAPS

The first part of this section introduces *usage maps*, written Δ. A usage map summarizes which resources are manipulated by a given subterm and how. Usage maps are used in particular for minimizing loop contracts, for checking the irrelevance of the evaluation order of multiple subexpressions, and for guiding certain transformations such as loop fission. Such usage maps can be computed by our typechecker on any subterm. We hence generalize triples from the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$ to the form $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$. Section 6.1 presents the grammar of usage maps. Section 6.2 presents operations on usage maps. Section 6.3 explains how usage maps are computed by our typing algorithm. Section 6.4 introduces notation for usage maps of sequences of instructions.

The second part of this section focuses on useful applications of usage maps. Section 6.5 presents the loop fission transformation, which exploits usage maps to synthesize contracts. Section 6.6 introduces the *triple minimization* operation, which critically exploits usage maps. Section 6.7 describes the *loop contract minimization* procedure, which relies on triple minimization. Section 6.8 presents the proof rule for subexpressions, which also relies on triple minimization. This proof rule applies as a preprocessing before the App rule can be invoked, when function arguments are not simple values (recall Section 4.7).

6.1 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 contains entries for 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 on all cells described by y before any read operation on y, then the usage of y is uninit.
- "y : splittedFrac" can arise when Γ contains a 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
  (2) t produced a resource of the form $(\beta_i - \gamma_1 - ... - \gamma_m)H$
  (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'.

6.2 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.

$$\begin{aligned} \Delta.\mathsf{consumed} &= \Delta.\mathsf{full} \cup \Delta.\mathsf{uninit} \\ \Delta.\mathsf{alter} &= \Delta.\mathsf{consumed} \cup \Delta.\mathsf{produced} \cup \Delta.\mathsf{ensured} \end{aligned}$$

Filtering on contexts. We first define filtering operation on sets of resources and contexts. We write $G \restriction X$, where G is a set of resources (linear or pure) and X is a set of variable names, to compute a set of resources where only the entries from G whose name belongs to the set X are kept. We generalize this operation to contexts: $\langle E \mid F \rangle \restriction X$ is defined as $\langle E \restriction X \mid F \restriction X \rangle$.

Δ_1; Δ_2	∅	required	ensured
∅	∅	required	ensured
required	required	required	⊥
ensured	ensured	ensured	⊥

Δ_1; Δ_2	∅	full	uninit	splittedFrac	joinedFrac	produced
∅	∅	full	uninit	splittedFrac	joinedFrac	produced
full	full	⊥	⊥	⊥	⊥	⊥
uninit	uninit	⊥	⊥	⊥	⊥	⊥
splittedFrac	splittedFrac	full	full	splittedFrac	splittedFrac	⊥
joinedFrac	joinedFrac	full	uninit	splittedFrac	joinedFrac	⊥
produced	produced	∅	∅	produced	produced	⊥

Fig. 25. Tables for sequential composition of two usage maps, for pure and for linear resources. For example, in the second table, the cell on the row "splittedFrac" and on the column "full" expresses that if "x : splittedFrac" is a binding from Δ_1 and "x : full" is a binding from Δ_2, then "x : full" is a binding in $\Delta_1;\Delta_2$. The input or output ∅ corresponds to cases where the usage map contains no binding for the resource name considered. The output ⊥ corresponds to cases that cannot arise according to our typechecking rules.

Intersection of usage maps, and filtering w.r.t. usage maps. We define:

$$\begin{aligned} \Delta_1 \cap \Delta_2 &= \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) \\ \Gamma \mid \cdot \Delta &= \Gamma \mid \cdot \mathrm{dom}(\Delta) \\ \Gamma \setminus \Delta &= \Gamma \mid \cdot (\mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Delta)) \end{aligned}$$

Sequential composition of usage maps. This paragraph 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 "$(\textbf{let}\ r = \mathrm{heapAlloc}_{\mathrm{UninitCell}}())^{\Delta_1};\ \mathrm{set}(r,\ v)^{\Delta_2};\ (\textbf{let}\ k = \mathrm{get}(r))^{\Delta_3};\ \mathrm{free}(r)^{\Delta_4}$", and let us focus on resources that describe the temporary cell r. In Δ_1, we have a binding "y_1 : produced" because the first instruction produces the resource "$y_1 : r \rightsquigarrow \mathrm{UninitCell}$". In Δ_2, we have two bindings "y_1 : uninit" and "y_2 : produced" because the second instruction consumes the resource "$y_1 : r \rightsquigarrow \mathrm{UninitCell}$" and produces a resource "$y_2 : r \rightsquigarrow v$". In Δ_3, we have a binding "y_2 : splittedFrac" because the instruction only reads with the permission y_2 (thus it accepts any subfraction). In Δ_4, we have a binding "y_2 : uninit" because the instruction destroys the resource y without caring about the value of the cell r.

Let us give three examples of compositions. First, the usage map $\Delta_1;\ \Delta_2$ contains a binding "y_2 : produced" because the sequential composition of those two instructions creates the resource y_2. Second, the usage map $\Delta_3;\ \Delta_4$ contains a binding y_2 : full because, taken together, the third and the fourth instructions consume the cell, and read the value that was contained inside. Third, the usage map $\Delta_1;\ \Delta_2;\ \Delta_3;\ \Delta_4$ contains no binding for y_1 or y_2 because the cell r cannot be seen from outside the sequence of those four instructions.

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 in Figure 25 to compute the *combined usage* in case a same resource name is bound both in Δ_1 and Δ_2.

The input or output ∅ 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 abstracts away intermediate resources not present in the final triple.

The output $\bot$ 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 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.

6.3 Computing Usage Maps

Usage of a context subtraction. Each time a proof 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 composed only of uninitialized cells. 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 each newly generated fraction gets an ensured entry.

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 for the merged 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. Therefore, we have to put the usage y_2 : 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 y_1 : 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 proof rules to record the usage information.

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

$$\frac{\Gamma.\mathsf{pure} \vdash t : \tau \qquad t \text{ is a literal or a variable} \qquad \Delta = \{\mathbf{res} : \mathsf{ensured}\}}{\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma \circledast [\mathbf{res} := t : \tau]\}\!\}}\ \textsc{LitOrVar}$$

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

For the rule LitOrVar, the usage map contains a single binding **res** : ensured to account for the alias added in the context. For the rule Let, the typechecker uses the operator $\mathsf{Rename}\{x := x'\}(\Delta)$,

that renames the key x into x' inside the map Δ. This renaming is applied on the usage map of the body to follow the renaming in the context.

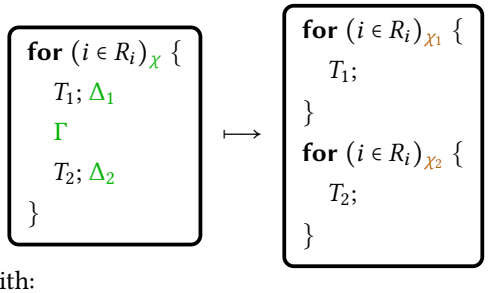
For the interested reader, we now explain how usage maps are computed in practice. Instead of rewriting each proof rule with explicit usage maps, which would be quite verbose, we simply explain how the rules are extended. We reuse the variables names of the rules described in Figure 20.

- For the rule Seq, if each instruction t_i has a usage map Δ_i, the usage map of the sequence Δ is given by:
$$\Delta = \begin{cases} \mathsf{Rename}\{r := \mathbf{res}\}(\Delta_1; ...; \Delta_n) & \text{if } r \neq \varnothing \\ (\Delta_1; ...; \Delta_n) & \text{if } r = \varnothing \end{cases}$$
- 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 $\Gamma_1 \boxminus \gamma.\mathsf{post}$, and S the generated specification hypothesis, then the usage map of the function definition is $((\Delta_1; \Delta_2) \mid \cdot\mathsf{dom}(\Gamma_0.\mathsf{pure})) \cup \{x : \mathsf{required} \mid x \in \mathsf{fv}(\gamma)\} \cup \{\mathbf{res} : \mathsf{ensured}, S : \mathsf{ensured}\}$. 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 $\Delta^{\mathsf{out}}_{\mathsf{pre}}$ is the usage map of the subtraction $\Gamma_0 \ominus \Gamma^{\mathsf{out}}_{\mathsf{pre}}$, $\Delta^{\mathsf{in}}_{\mathsf{body}}$ is the usage of the body of the loop, $\Delta^{\mathsf{in}}_{\mathsf{post}}$ is the usage of the subtraction on $\Gamma^{\mathsf{in}}_{\mathsf{post}}$, $\Delta^{\mathsf{out}}_{\mathsf{post}}$ is a usage map containing one produced (resp. ensured) for each linear (resp. pure) resource in $\Gamma^{\mathsf{out}}_{\mathsf{post}}$, and Δ_r is the usage of the CloseFracs operation, then $(\Delta^{\mathsf{out}}_{\mathsf{pre}}; ((\Delta^{\mathsf{in}}_{\mathsf{body}}; \Delta^{\mathsf{in}}_{\mathsf{post}}) \mid \cdot\mathsf{dom}(\Gamma_0.\mathsf{pure})); \Delta^{\mathsf{out}}_{\mathsf{post}}; \Delta_r)$ is the usage map of the **for** loop. Note that the $((\Delta^{\mathsf{in}}_{\mathsf{body}}; \Delta^{\mathsf{in}}_{\mathsf{post}}) \mid \cdot\mathsf{dom}(\Gamma_0.\mathsf{pure}))$ 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, applied to a conditional $\mathbf{if}_{\Gamma_r}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$, it is always sound (though possibly imprecise) to combine the usage map Δ_0 of the condition expression t_1 to another usage map Δ_1 that gives a full usage to each linear resource in Γ_1 (the output context of t_1) and a usage map Δ_r that contains a produced usage for each linear resource of Γ_r. For the usage of pure resources, we name Δ_2 (resp. Δ_3) the required usage from t_2 (resp. t_3). Then, we take all the pure facts from Γ_r that are not in Γ_1 as ensured in a usage map Δ'_r. In summary, we compute the usage map of the whole conditional as $(\Delta_0; \Delta_1; \Delta_2; \Delta_3; \Delta_r; \Delta'_r)$.

6.4 Notation for Usage Maps

Some 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 $\Gamma_1\ T; \Delta\ \Gamma_2$, where Γ_1 and Γ_2 denotes the initial and final contexts, and Δ denotes the *composition* of the usages from the group of instructions, as defined in Section 6.3.

$$\boxed{\Gamma_0\ T; \Delta\ \Gamma_n} \quad = \quad \boxed{\Gamma_0\ t_1; \Delta_1\ \Gamma_1\ t_2; \Delta_2\ \Gamma_2\ ...\ \Gamma_{n-1}\ t_n; \Delta_n\ \Gamma_n} \qquad \begin{array}{ll} \text{where} & T = t_1; t_2; ...; t_n \\ \text{and} & \Delta = \Delta_1; \Delta_2; ...; \Delta_n \end{array}$$



succeeds if:
(1) i is not free in χ.shrd
(2) $\chi.\text{shrd} \mathbin{!}\cdot(\Delta_1.\text{alter} \cap \Delta_2) = \varnothing$
(3) $\chi.\text{shrd} \mathbin{!}\cdot(\Delta_2.\text{alter} \cap \Delta_1) = \varnothing$

with:

$$\_, F = \Gamma \boxminus \chi.\text{shrd.inv} \qquad \_, F_{\text{cut}} = F \boxminus \text{StackAllocCells}(T_1)$$

$$\chi_1 = \begin{cases} \text{vars} = \chi.\text{vars} \\ \text{shrd} = \chi.\text{shrd} \mathbin{!}\cdot \Delta_1 \\ \text{excl} = \{\text{pre} = \chi.\text{excl.pre},\ \text{post} = F_{\text{cut}}\} \end{cases} \qquad \chi_2 = \begin{cases} \text{vars} = \chi.\text{vars} \\ \text{shrd} = \chi.\text{shrd} \mathbin{!}\cdot \Delta_2 \\ \text{excl} = \{\text{pre} = F_{\text{cut}},\ \text{post} = \chi.\text{excl.post}\} \end{cases}$$

where $\chi.\text{shrd} \mathbin{!}\cdot X$ is a shorthand for $\{\text{inv} = (\chi.\text{shrd.inv} \mathbin{!}\cdot X), \text{reads} = (\chi.\text{shrd.reads} \mathbin{!}\cdot X)\}$.

Fig. 26. The basic transformation `Loop.fission`.

6.5 Loop Fission

A transformation that crucially depends on usage information is `Loop.fission`. In its *basic* version, it breaks a loop with body $T_1; T_2$ into two loops over the same range, a first with body T_1, and a second with body T_2. The transformation is described in Figure 26. As for loop swapping, dependencies can be nontrivial in general. For now, we automatically generate contracts for the case where the resources altered by T_1 at any iteration i do not interfere with the ones altered by T_2 at any other iteration $i' \neq i$.

Let us explain how to synthesize the contracts χ_1 and χ_2 for 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 resources at the cut point, written F_{cut}. The first loop takes the exclusive resources from χ.excl.pre to F_{cut}, whereas the second loop takes the exclusive resources from F_{cut} to χ.excl.post. At a high level, F_{cut} is computed by subtracting the shared resources as well as the local allocations from T_1, described by χ.shrd and StackAllocCells(T_1), from the typing 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 typing context than strictly necessary. Further on (Section 6.7), we describe a procedure for minimizing loop contracts. This procedure is based on a triple minimization operation, which we describe next.

6.6 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(Γ, Γ', Δ) is defined when its input corresponds to a valid triple $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$. The output of the operation is a quadruplet $(E^{\text{fracs}}, \hat{F}, \hat{F}', F^{\text{framed}})$.

- $\hat{F}$ 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.

- F^{framed} is the *maximal frame*: a linear context of resources from Γ.linear that were superfluous in the typechecking of t. It means resources in F^{framed} can be framed during the typechecking of t. Since these resources are not touched by t, they must also occur in Γ'.linear.
- E^{fracs} 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.

Concretely, the result of Minimize is guided by the entries in the usage map Δ, which is computed when typechecking t.

- If t can typecheck without a linear resource H, then H should be put in the frame F^{framed}.
- If t can typecheck with only an uninitialized version of H (because, for instance, it starts by overwriting the data accessible through H), then such uninitialized version of 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 E^{fracs}, the resource αH should be placed in $\hat{F}$, and $(1-\alpha)H$ should remain in F^{framed}.

Detailed examples and an algorithmic description of Minimize can be found in appendix F. From the perspective of establishing soundness results, the following three properties about the quadruplet $(E^{\text{fracs}}, \hat{F}, \hat{F}', F^{\text{framed}})$ are useful:

- $\{\!\{\langle \Gamma.\text{pure}, E^{\text{fracs}} \mid \hat{F}\rangle\}\!\}\ t\ \{\!\{\langle \Gamma'.\text{pure}, E^{\text{fracs}} \mid \hat{F}'\rangle\}\!\}$, which corresponds to the minimized triple.
- $\Gamma \Rightarrow \langle \Gamma.\text{pure}, E^{\text{fracs}} \mid \hat{F} \star F^{\text{framed}}\rangle$, which describes the decomposition of Γ.
- $\langle \Gamma'.\text{pure}, E^{\text{fracs}} \mid \hat{F}' \star F^{\text{framed}}\rangle \Rightarrow \Gamma'$, which describes the decomposition Γ'.

6.7 Minimization of Loop Contracts

Loop transformations often produce resource annotations that are not *minimal*, in the sense that the loop contracts may require more resources than strictly necessary. For example, after the basic loop fission transformation, the contract of the first loop may 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 basic transformation to *minimize* loop contracts, which is systematically called by combined loop transformations.

Loop contract minimization takes as input a loop with contract χ, and updates it to χ', without modifying the code. The procedure depends on the usage map Δ computed for the loop body.

$$\boxed{\textbf{for}\ (i \in R_i)_{\chi}\ \{T; \Delta\}} \longmapsto \boxed{\textbf{for}\ (i \in R_i)_{\chi'}\ \{T\}}$$

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', and can be removed as well. Second, if a resource appears with fraction 1 in χ, yet this resource is marked as splittedFrac in Δ, then this resource is replaced with a read-only version of it. Note that weakening a permission to read-only requires quantifying an additional fraction variable in χ'.

Internally, the implementation of contract minimization reuses the aforementioned *minimization of triple* procedure. Details may be found in appendix G.

6.8 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, and in our subset OptiC, the arguments of a function call may be evaluated in an arbitrary order. The fact that the order is not fixed is interesting because it leaves additional flexibility for optimizations. Our typesystem checks that, for well-typed OptiTrust programs, the order of evaluation is indeed irrelevant. To that end, we consider a sufficient condition: that the arguments can be evaluated in parallel, in the sense that the side effects performed by one argument should not interfere with the evaluation of any other argument.

Remark: there exist OptiC programs that fail to typecheck because our condition is slightly more restrictive than evaluation order irrelevance. For example, the expression $\mathsf{add}(\mathsf{getAndIncr}(x), \mathsf{getAndIncr}(x))$ has an irrelevant order of evaluation but the two occurrences of getAndIncr have conflicting side effects. However, such programs are quite rare and may be easily rewritten to avoid this issue by creating intermediate variables before the function call. Besides, parallel evaluation of function arguments gives even more flexibility for transformations compared to undefined evaluation order.

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 $f(t_1, ..., t_n)$ 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 $\hat{\mathcal{E}}[t_0, ..., t_n]$, where $\hat{\mathcal{E}}$ denotes a *multi-evaluation-context*, and where each t_i denotes a subterm in evaluation position. A multi-evaluation-context is a term with ordered holes that are all in evaluation position. We write $\hat{\mathcal{E}}[t_0, ..., t_n]$ the operation that fills the holes with terms t_0 to t_n. For example, if $\hat{\mathcal{E}}$ denotes the multi-evaluation-context $\square(\square, ..., \square)$, then the application $\hat{\mathcal{E}}[f, t_1, ..., t_n]$ produces the function call $f(t_1, ..., t_n)$.

The goal of the rule Subexpr 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.

Subexpr

$$\frac{\begin{array}{c}\forall i \in [1,n].\quad \{\!\{\Gamma_{i-1}\}\!\}\ t_i^{\Delta_i}\ \{\!\{\Gamma_i'\}\!\} \quad\wedge\quad (E_i^{\mathsf{fracs}}, \hat{F}_i, \hat{F}_i', F_i^{\mathsf{framed}}) = \mathsf{Minimize}(\Gamma_{i-1}, \Gamma_i', \Delta_i) \quad\wedge\quad x_i \text{ fresh} \\ \forall i \in [1,n].\quad \Gamma_i = \langle\, \Gamma_i.\mathsf{pure}, E_i^{\mathsf{fracs}} \mid F_i^{\mathsf{framed}} \,\rangle \quad\wedge\quad \hat{\Gamma}_i' = \langle\, \Gamma_i'.\mathsf{pure} \mid \cdot \mid \Delta_i.\mathsf{ensured} \mid \hat{F}_i' \,\rangle \\ \Gamma_p = \mathsf{CloseFracs}^{\Delta_p}\left(\Gamma_n \circledast \circledast_{i\in[1,n]} \mathsf{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}_i')\right) \\ \{\!\{\Gamma_p\}\!\}\ \hat{\mathcal{E}}[x_1, ..., x_n]^{\Delta_q}\ \{\!\{\Gamma_q\}\!\} \\ \Delta = \mathsf{Rename}\{\mathbf{res} := x_1\}(\Delta_1); ...; \mathsf{Rename}\{\mathbf{res} := x_n\}(\Delta_n); \Delta_p; \Delta_q\end{array}}{\{\!\{\Gamma_0\}\!\}\ \hat{\mathcal{E}}[t_1, ..., t_n]^{\Delta}\ \{\!\{\Gamma_q\}\!\}}$$

Appendix H presents an example application of this rule.

7 RELATED WORK

The most closely related work has been discussed in the introduction, in particular the line of work on proof-carrying code and proof-preserving compilation. In this section, we comment on the remaining related work.

Source Code Manipulation Frameworks. Numerous tools have been developed to perform source-to-source transformations.

Some of these tools are based on rewrite rules that in essence map a code pattern to another code pattern. 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 [Bravenboer et al. 2008] program transformation language to C++ code, and has been used to turn high-level operations on matrices and vectors into typical high-performance source code. OptiTrust supports, among other transformations, the application of rewrite rules.

Other tools give access to the abstract syntax tree (AST) of the source code. Contrarily to traditional compilers, they are designed to keep synthesizing and manipulating source-level AST, rather than less structured intermediate representations. 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].

Locus [Teixeira et al. 2019] is an optimization framework for optimizing C programs. A fine-grained DSL language is used for describing the search space of possible optimizations (e.g. which tile sizes to try, whether to consider loop interchange, etc.). Then, the framework generates and benchmarks candidate implementations. The applicability of the tool is illustrated on hundreds of loop nests from a variety of HPC kernels.

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 with imperative invocations of transformations. An application article on the Pegasus tool [Pinto et al. 2020] illustrates this approach on loop tiling and interchange operations.

None of the aforementioned frameworks provide formal guarantees on the transformed code. The specificity of OptiTrust is to provide such guarantees by transforming the code together with its separation logic proof.

Reasoning about the Correctness of Program Optimizations. A central question is how to determine whether a program optimization is correct.

Many optimization tools leverage a *functional* language in the first compilation stages to greatly simplify program reasoning. This is the case in particular in the array-based programming languages Halide [Ragan-Kelley et al. 2013], Elevate [Hagedorn et al. 2020b], and ATL [Liu et al. 2022]. The strand of work on *multi-dimensional homomorphisms* also relies on a functional algebraic formalism to represent a wide range of data-parallel computations with only three higher-order functions [Rasch 2024]. However, functional languages are not adapted to express imperative optimization concerns such as reusing memory. Therefore, this approach typically involves an imperative code generation stage and does not provide precise control over that stage.

Other tools aim at optimizing imperative code, leveraging dependency analyses to reason about transformation correctness. In the context of array code, work on the *polyhedral model* [Feautrier 1992] has pioneered transformations on *Static Control Programs* (i.e., programs where the control flow cannot depend on data stored in arrays) with code involving nested loops and *quasi-affine array indexing* (i.e., accesses where the index of is a linear combination of loop variables and constants, moreover allowing for division and modulo by a constant). Clay [Bagnères et al. 2016] is a framework to assist in the optimization of loop nests that can be described in the polyhedral

model. Exo is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. Exo "can be seen as a polyhedral compiler" [Ikarashi et al. 2022]. Because Exo code almost syntactically translates to C code, its code generation stage makes few decisions beyond user control.

A major limitation of such polyhedral analyses is their restriction to static control programs with quasi-affine array indexing, as well as the fact that they are not modular: each function call site is analyzed as if the function was inlined. In contrast, the specificity of OptiTrust is to support reasoning about *less constrained*, *imperative* array code, moreover in a *modular way*.

Another approach to reasoning about data and effect dependencies is to leverage Stateful DataFlow multiGraph (SDFG) [Ben-Nun et al. 2019], which consist of a specialized, extended form of the standard dataflow graphs. This representation allows manipulating explicitly the data movement, the parallelism and the memory hierarchy by means of subgraph rewriting, buffer introduction, and pipeline computations. As far as we know, the existing work on SDFGs does not include formal specifications or invariants. That said, we speculate that SDFGs could accommodate them.

Transformation Scripts. Expressing a series of 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. 2016; 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 the optimization space. Clay [Bagnères et al. 2016] 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.

As mentioned in the introduction, Halide [Ragan-Kelley et al. 2013] is an industrial-strength domain-specific compiler for image processing that 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 many major processor 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.

The strand of work on multi-dimensional homomorphisms also explores the use of a scheduling language, focusing on decisions such as tiling and mapping to the target thread and memory hierarchies [Rasch et al. 2023].

All these tools do not support higher-order composition of transformations, and are not easily 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].

Extensibility of Transformations. 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.

ATL [Liu et al. 2022], already mentioned in the introduction, has the particularity of being embedded into the Rocq proof assistant. ATL programs can be transformed through the application of rewrite rules expressed as Rocq theorems. ATL transformations are inherently accompanied by machine-checked proofs of correctness. Once optimized, ATL programs are then compiled into imperative C code—without formal guarantees, like in OptiTrust. The set of rules includes expressive transformations, some beyond the scope of Halide, and can be extended by the user.

Exo [Ikarashi et al. 2022] pushes the idea of 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 a cursor mechanism for targeting code points [Ikarashi et al. 2025]. As in Elevate, the user can add custom transformations, possibly defined by (higher-order) composition.

MLIR (Multi-Level Intermediate Representation) [Lattner et al. 2021] is a framework for building reusable and extensible compiler infrastructure. MLIR aims in particular at improving compilation for heterogeneous hardware, and at improving support for DSL constructs. To that end, MLIR provides *dialects*, which enables expressing extended language constructs. For example, the *tensor dialect* helps representing multidimensional arrays and operations on them. Recently, a *transform dialect* [Lücke et al. 2024] was added to MLIR to express transformation scripts. Verification dialects have also been added to provide more formal guarantees [Fehr et al. 2025]. These extensions confirm the interest for both finer-grained control and formal validation.

None of the aforementioned tools explore proof-preserving compilation as OptiTrust does. We leave it to future work to explore how the OptiTrust framework could be extended to support user-defined language constructs, possibly taking inspiration from MLIR [Lattner et al. 2021], AnyDSL [Leißa et al. 2018], or MimIR [Leißa et al. 2025]; and to support user-defined hardware backends, as done for example in Exo [Ikarashi et al. 2022].

8 CONCLUSION

We have presented OptiTrust, a framework for optimizing code equipped with formal invariants, with support for advanced transformations whose application can be finely controlled by the user. OptiTrust concretizes the old idea of proof-carrying code for high-level transformations on array-manipulating code. We have demonstrated the applicability of OptiTrust on three realistic case studies, two of which correspond to industrial-quality code featuring nontrivial optimizations.

There are numerous directions for future work. As mentioned in Section 1.3, we wish to consider as case studies programs that manipulate pointers in nontrivial ways, to support concurrent programming patterns, to support the embedding of domain-specific languages, and to establish foundational guarantees for OptiTrust. Because we are building OptiTrust on separation logic, whose expressiveness has been extensively demonstrated, we are confident that our methodology has the potential to be generally applicable.

To improve the achievable performances, we have started to investigate how to support GPU code, as well as more exotic accelerators providing ad-hoc instructions such as matrix-multiplication. Besides, we plan to work on extending OptiTrust to support compilation passes typically found in the back-end of a compiler, e.g., register allocation and layout of stack frames.

To improve usability, we look forward to devise mechanisms to automatically infer some annotations, e.g., the dimensions that appear in array accesses. In particular, we began to work on the automatic inference of focus operations. Going further, we could investigate the inference of certain loop contracts, as illustrated in the literature by means of abstract interpretation Journault

and Miné [2018], of *bi-abduction* [Calcagno et al. 2019; Spies et al. 2024], or of domain-specific techniques, e.g., for stencil computations [Kamil et al. 2016]. The inference of some contracts would reduce the effort required for verifying the unoptimized input code.

Last but not least, we plan to investigate how AI could be used to synthesize annotated code, as well as transformation scripts.

REFERENCES

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

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

Andrew W. Appel. 2001. Foundational Proof-Carrying Code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*. IEEE Computer Society, USA, 247.

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. 2016. 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. 2016. Opening Polyhedral Compiler's Black Box. In *IEEE/ACM International Symp. on Code Generation and Optimization*.

Fabian Bannwart and Peter Müller. 2005. A Program Logic for Bytecode. *Electronic Notes in Theoretical Computer Science* 141, 1 (2005), 255–273. https://doi.org/10.1016/j.entcs.2005.02.026 Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005).

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

Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. 2006. Certificate Translation for Optimizing Compilers. In *Static Analysis*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 301–317.

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

Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. https://doi.org/10.1145/3295500.3356173

Guillaume Bertholon. 2025. *Interactive compilation via trustworthy source-to-source transformations.* Ph. D. Dissertation. https://doi.org/10.70675/c96ce4b6z259cz4adez8d3azfb9185492a3f Thèse de doctorat dirigée par Arthur Charguéraud. Informatique, Strasbourg 2025.

Guillaume Bertholon and Arthur Charguéraud. 2025. Bidirectional Translation between a C-like Language and an Imperative Lambda-calculus. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*. Roiffé, France. https://inria.hal.science/hal-04859522

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

Gary Bradski, Adrian Kaehler, et al. 2000. OpenCV. *Dr. Dobb's journal of software tools* 3, 2 (2000). https://opencv.org/.

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. 2011. Characteristic Formulae for the Verification of Imperative Programs. In *International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/2034773.2034828

Arthur Charguéraud. 2020. 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, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2022. Omnisemantics: Smoother Handling of Nondeterminism. (March 2022). https://hal.inria.fr/hal-03255472 working paper or preprint.

Arthur Charguéraud. 2020. 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

Basile Clément and Albert Cohen. 2022. End-to-end translation validation for the Halide language. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 84 (apr 2022), 30 pages. https://doi.org/10.1145/3527328

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

Thibault Dardinier, Peter Müller, and Alexander J. Summers. 2022. Fractional resources in unbounded separation logic. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 163 (Oct. 2022), 27 pages. https://doi.org/10.1145/3563326

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

Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming* 21, 5 (10 1992), 313–348.

Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1466–1490. https://doi.org/10.1145/3729309

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

Benjamin Goldberg, Lenore Zuck, and Clark Barrett. 2005. Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers. *Electronic Notes in Theoretical Computer Science* 132, 1 (2005), 53–71. https://doi.org/10.1016/j.entcs.2005.01.030 Proceedings of the 3rd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2004).

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.

Tony Hoare. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (Jan. 2003), 63–69. https://doi.org/10.1145/602382.602403

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, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 426–444. https://doi.org/10.1145/3669940.3707218

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

Bart Jacobs and Frank Piessens. 2008. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven. http://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast.pdf

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

Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *SIGPLAN Not.* 51, 6 (June 2016), 711–726. https://doi.org/10.1145/2980983.2908117

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

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

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL* (San Diego, California, USA). Association for Computing Machinery, 13 pages. https://doi.org/10.1145/2535838.2535841

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

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

Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: a partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 119:1–119:30. https://doi.org/10.1145/3276489

Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2025. MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age. *Proc. ACM Program. Lang.* 9, POPL (2025), 95–125. https://doi.org/10.1145/3704840

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

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

Martin Paul Lücke, Oleksandr Zinenko, William S. Moses, Michel Steuwer, and Albert Cohen. 2024. The MLIR Transform Dialect. Your compiler is more powerful than you think. *CoRR* abs/2409.03864 (2024). https://doi.org/10.48550/ARXIV.2409.03864 arXiv:2409.03864

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

Peter Müller and Martin Nordio. 2007. Proof-transforming compilation of programs with abrupt termination. In *Proceedings of the 2007 Conference on Specification and Verification of Component-Based Systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia) *(SAVCBS '07)*. Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/1292316.1292321

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.

George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

Martin Nordio, Peter Müller, and Bertrand Meyer. 2008. Proof-Transforming Compilation of Eiffel Programs. In *Objects, Components, Models and Patterns*, Richard F. Paige and Bertrand Meyer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 316–335.

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

A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

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.

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

Ari Rasch. 2024. (De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms. *ACM Trans. Program. Lang. Syst.* 46, 3, Article 10 (Oct. 2024), 74 pages. https://doi.org/10.1145/3665643

Ari Rasch, Richard Schulze, Denys Shabalin, Anne C. Elster, Sergei Gorlatch, and Mary W. Hall. 2023. (De/Re)-Compositions Expressed Systematically via MDH-Based Schedules. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023, Montréal, QC, Canada, February 25-26, 2023*, Clark Verbrugge, Ondrej Lhoták, and Xipeng Shen (Eds.). ACM, 61–72. https://doi.org/10.1145/3578360.3580269

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 Şakar, 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

Ömer Şakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. 2025. Preserving provability over GPU program optimizations with annotation-aware transformations. *Formal Methods in System Design* (12 2025), 1–57. https://doi.org/10.1007/s10703-025-00480-7

Hanan Samet. 1975. *Automatically proving the correctness of translations involving optimized code - research sponsored by Advanced Research Projects Agency, ARPA order no. 2494*. Ph. D. Dissertation. Stanford University.

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

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/231379.231414

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. 2019. Locus: a system and a language for program optimization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 217–228.

Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. *SIGPLAN Not.* 46, 6 (June 2011), 295–305. https://doi.org/10.1145/1993316.1993533

Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified validation of lazy code motion. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 316–326. https://doi.org/10.1145/1542476.1542512

Jean-Baptiste Tristan and Xavier Leroy. 2010. A simple, verified validator for software pipelining. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 83–92. https://doi.org/10.1145/1706299.1706311

Carsten Varming and Lars Birkedal. 2008. Higher-Order Separation Logic in Isabelle/HOLCF. *Electronic Notes in Theoretical Computer Science* 218 (2008), 371 – 389. https://doi.org/10.1016/j.entcs.2008.10.022 Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIV).

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

Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. 2002. VOC: A Translation Validator for Optimizing Compilers1 1This research was supported in part by NSF grant CCR-0098299, ONR grant N00014-99-1-0131, and the Minerva Center for Verification of Reactive Systems, a gift from Intel, a grant from the German - Israel Foundation for Scientific Research and Development. *Electronic Notes in Theoretical Computer Science* 65, 2 (2002), 2–18. https://doi.org/10.1016/S1571-0661(04)80393-1 COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002).

```
float* const pB = (float*)malloc(MSIZE4(32, 256, 4, 32) * sizeof(float));
#pragma omp parallel for
for (int bj = 0; bj < 32; bj++) {
  __sreads("b ⇝ Matrix2(1024, 1024, B)");
  __xwrites("for bk in 0..256 → for k in 0..4 → for j in 0..32 →
      &pB[MINDEX4(32, 256, 4, 32, bj, bk, k, j)] ↦ B(bk * 4 + k, bj * 32 + j)");
  for (int bk = 0; bk < 256; bk++) {
    __sreads("b ⇝ Matrix2(1024, 1024, B)");
    __xwrites("for k in 0..4 → for j in 0..32 →
      &pB[MINDEX4(32, 256, 4, 32, bj, bk, k, j)] ↦ B(bk * 4 + k, bj * 32 + j)");
    __ASSERT(tile_div_check_j512, "1024 == 32 * 32");
    for (int k = 0; k < 4; k++) {
      __sreads("b ⇝ Matrix2(1024, 1024, B)");
      __xwrites("for j in 0..32 →
        &pB[MINDEX4(32, 256, 4, 32, bj, bk, k, j)] ↦ B(bk * 4 + k, bj * 32 + j)");
      for (int j = 0; j < 32; j++) {
        __sreads("b ⇝ Matrix2(1024, 1024, B)");
        __xwrites("&pB[MINDEX4(32, 256, 4, 32, bj, bk, k, j)] ↦ B(bk * 4 + k, bj * 32 + j)");
        __ASSERT(tile_div_check_k13, "1024 == 256 * 4");
        __ghost(tiled_index_in_range,
            "tile_index := bk, index := k, div_check := tile_div_check_k13");
        __ghost(tiled_index_in_range,
            "tile_index := bj, index := j, div_check := tile_div_check_j512");
        __ghost_begin(__ghost_pair_3, ro_matrix2_focus,
          "matrix := b, i := bk * 4 + k, j := bj * 32 + j");
        pB[MINDEX4(32, 256, 4, 32, bj, bk, k, j)] =
          b[MINDEX2(1024, 1024, bk * 4 + k, bj * 32 + j)];
        __ghost_end(__ghost_pair_3);
} } } }
```

Fig. 27. Excerpt of our optimized code for matrix multiplication with full functional correctness annotations. This excerpt contains annotated generated loops to pre-transpose the matrix b in pB before computation.

APPENDIX

A Matrix-Multiply: Annotated Output Code

The matrix multiplication case study was presented in Section 2.3. Figures 27 and 28 show two excerpts of the annotated code obtained at the last transformation step, just before the (unverified) final extraction to C code. These excerpts include, in particular, the invariants for all the loops produced by the transformations. As the excerpts suggest, a vast amount of annotations is involved for proving the correctness of the optimized code. The OptiTrust transformation-based approach, at the cost of developing a relatively-short transformation script, saves the effort of writing all these annotations by hand.

B Matrix-Multiply: Comparison against TVM

The TVM matrix-multiply case study, as available in v0.19, appears in Figure 29. We only comment on specific aspects and refer to TVM's tutorial for further details. Please also note that the TVM programming API changes significantly after v0.19 as a result of significant refactoring efforts. Some aspects of this comparison may not apply to later TVM developments.

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 Figure 29 would replace the definitions of pB and C with a simpler definition of C:

```
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 code. Likewise, the blocking strategy needs to be hardwired in the source code on the left-hand side of Figure 29. In contrast, our input program for matrix multiplication shown in Figure 11

```
#pragma omp simd
for (int j = 0; j < 32; j++) {
  __sreads("a ~> Matrix2(1024, 1024, A)");
  __xconsumes("&s[MINDEX1(32, j)] ↦
    reduce_sum(0..(bk * 4 + 3), fun k0 → A(bi * 32 + i, k0) * B(k0, bj * 32 + j))");
  __xproduces("&s[MINDEX1(32, j)] ↦
    reduce_sum(0..(bk * 4 + (3 + 1)), fun k0 → A(bi * 32 + i, k0) * B(k0, bj * 32 + j))");
  __xreads("&pB[MINDEX4(32, 256, 4, 32, bj, bk, 3, j)] ↦ B(bk * 4 + 3, bj * 32 + j)");
  __ghost(tiled_index_in_range,"tile_index := bj, index := j, div_check :=
     tile_div_check_j51221");
  __ASSERT(tile_div_check_k1320, "1024 == 256 * 4");
  __ghost(tiled_index_in_range, "tile_index := bk, index := 3, div_check :=
     tile_div_check_k1320");
  __ghost_begin(focusA3, ro_matrix2_focus, "matrix := a, i := bi * 32 + i, j := bk * 4 + 3");
  s[MINDEX1(32, j)] +=
    a[MINDEX2(1024, 1024, bi * 32 + i, bk * 4 + 3)] * pB[MINDEX4(32, 256, 4, 32, bj, bk, 3, j)
     ];
  __ghost_end(focusA3);
  __ghost(in_range_bounds, "x := bk * 4 + 3", "k_gt_021 <- lower_bound");
  __ghost(rewrite_float_linear, "inside := fun v → &s[MINDEX1(32, j)] ↦ v,
    by := reduce_sum_add_right(bk * 4 + 3, fun k → A(bi * 32 + i, k) * B(k, bj * 32 + j),
     k_gt_021)");
 __ghost(rewrite_int_linear, "inside := fun (k: int) → &s[MINDEX1(32, j)] ↦
      reduce_sum(0..k, fun k0 → A(bi * 32 + i, k0) * B(k0, bj * 32 + j)),
    by := add_assoc_right(bk * 4, 3, 1)");
}
```

Fig. 28. Excerpt of our optimized code for matrix multiplication with full functional correctness annotations. This excerpt corresponds to the last of the four inner-loops accumulating in the local variable s.

```
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. 29. 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*). Both use Python syntax.

	median	90th percentile
Naive	1409.0 ms	1409.8 ms
TVM	9.2 ms	9.4 ms
OptiTrust	8.7 ms	9.4 ms

Fig. 30. Benchmark of the performance of a single 1024×1024 matrix multiplication

builds upon familiar C-like syntax and, most importantly, includes no optimization. Starting from totally unoptimized reference code improves readability, trustworthiness, and maintainability. Besides, although our input code for matrix-multiply is currently expressed using explicit loops, in the future we could alternatively express it using higher-order combinators as well.

The right-hand side of Figure 29 shows TVM's optimization script. Our optimization script shown in Figure 13 is not much more verbose than that of TVM. Note that TVM does not produce

C code but directly outputs LLVM IR code. By manual inspection of the code, we checked that the code produced using OptiTrust features the same optimizations. Besides, we ran a benchmark to compare the performance of the two programs.

Figure 30 shows the performance of a single 1024×1024 matrix multiplication over 200 runs. The experiment ran on a 4-core Intel i7-8665U CPU with AVX2 support. The first column shows the median across 200 runs, whereas the second column shows the 90th percentile. Both OptiTrust and TVM reach essentially the same performance, roughly 150× speedup over the naive algorithm.

Finally, let us comment on interactivity. Guided by all the contents from Figure 29, 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, directly manipulating OptiC programs. Moreover, it provides human-readable diffs for every step and every substep involved in the optimization process.

C Semantics of Optiλ

We formalize the semantics of Optiλ using an omni-big-step evaluation ([Charguéraud et al. 2022]) judgment in call by value style. The judgment $t/(s, m) \Downarrow Q$ asserts that the term t, in a program stack s and in a program store m, evaluates to result states that belong to the set Q. The result states in Q are of the form (s', m') where s' is a program stack and m' a program store. Program stacks map program variables to values, and program stores map each location to a mode and a value. Modes in program stores, denoted $\mathcal{M}$, are either RW (read-write) or RO (read-only). A location in mode RO is shared between several threads and therefore cannot be written to. A location in mode RW is exclusively manipulated by the current execution thread and has no such restriction. The values, denoted v, can be logical expressions, locations, function closures of the form $\mathbf{fun}^{s}(x_1, ..., x_n) \mapsto t$, and the special uninitialized value $\bot$.

The operator IntoRO applied on a program store is defined as follows:

$$\mathsf{IntoRO}(m) = \{l \mapsto (\mathsf{RO}, v) \mid \exists \mathcal{M}, m(l) = (\mathcal{M}, v)\}$$

Figure 31 gives the semantic rules of Optiλ. The evaluation contexts consist of function arguments and ranges of **for** loops.

Remark: the rule Seq encodes the fact that a sequence creates a lexical scope by restoring the program stack after its execution. The result value, if any, is bound in the output stacks.

D Details of the Subtraction Algorithm

The core subtraction operation, written $\Gamma \boxminus \Gamma' = (\sigma, F)$, is implemented as follows.

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

The carving subtraction operation, written $\Gamma \boxminus \Gamma' = (E_{\mathsf{frac}}, \sigma, F)$, is implemented by making the following refinements. At step (1), E_{frac} is initialized as an empty environment. 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

$$\frac{}{v/(s,m) \Downarrow \{(s[\mathbf{res} \mapsto v], m)\}}\ \textsc{Val} \qquad \frac{}{x/(s,m) \Downarrow \{(s[\mathbf{res} \mapsto s(x)], m)\}}\ \textsc{Var}$$

$$\frac{}{(\mathbf{fun}(a_1, ..., a_n) \mapsto t_f)/(s,m) \Downarrow (s[\mathbf{res} \mapsto (\mathbf{fun}^s(a_1, ..., a_n) \mapsto t_f)], m)}\ \textsc{Fun}$$

$$\frac{}{(\mathbf{let}\ x = \mathsf{stackAlloc}())/(s,m) \Downarrow \{(s[x \mapsto l], m[l \mapsto (\mathsf{RW}, \bot)]) \mid l \notin \mathsf{dom}(m)\}}\ \textsc{StackAlloc}$$

$$\frac{t/(s,m) \Downarrow Q}{(\mathbf{let}\ x = t)/(s,m) \Downarrow \{(s[x \mapsto s'(\mathbf{res})], m') \mid (s', m') \in Q\}}\ \textsc{Let}$$

$$\frac{t/(s,m) \Downarrow Q' \qquad \forall (s', m') \in Q', \mathcal{E}[s'(\mathbf{res})]/(s,m') \Downarrow Q \qquad \mathcal{E} \text{ is an evaluation context}}{\mathcal{E}[t]/(s,m) \Downarrow Q}\ \textsc{Bind}$$

$$\frac{s_c = s_f[\overline{a_i \mapsto v_i}] \qquad t_f/(s_c, m) \Downarrow Q}{(\mathbf{fun}^{s_f}(a_1, ..., a_n) \mapsto t_f)(v_1, ..., v_n)/(s,m) \Downarrow Q}\ \textsc{Call}$$

$$\frac{\begin{array}{c} t_c/(s,m) \Downarrow Q_c \\ \forall (s_c, m_c) \in Q_c, (s_c(\mathbf{res}) = \mathbf{true} \implies t_t/(s, m_c) \Downarrow Q) \wedge (s_c(\mathbf{res}) = \mathbf{false} \implies t_f/(s, m_c) \Downarrow Q) \end{array}}{(\mathbf{if}\ t_c\ \mathbf{then}\ t_t\ \mathbf{else}\ t_f)/(s,m) \Downarrow Q}\ \textsc{If}$$

$$\frac{\begin{array}{c} Q_0 = \{(s_0, m_0)\} \qquad \forall i \in [1,n], \forall (s,m) \in Q_{i-1},\ t_i/(s,m) \Downarrow Q_i \\ Q_A = \{(s, m \setminus A(s)) \mid (s,m) \in Q_n\} \text{ where } A(s) = \{s(x_i) \mid t_i \text{ is of the form ``}\mathbf{let}\ x_i = \mathsf{stackAlloc}()\text{''}\} \\ Q = \begin{cases} \{(s_0, m) \mid (s,m) \in Q_A\} & \text{if } r = \varnothing \\ \{(s_0[\mathbf{res} \mapsto s(x)], m) \mid (s,m) \in Q_A\} & \text{if } r = x \end{cases} \end{array}}{\{t_1; ...; t_n; r\}/(s_0, m_0) \Downarrow Q}\ \textsc{Seq}$$

$$\frac{m(l) = (\mathcal{M}, v) \qquad v \neq \bot}{\mathsf{get}(l)/(s,m) \Downarrow \{s[\mathbf{res} \mapsto v], m\}}\ \textsc{Get} \qquad \frac{m(l) = (\mathsf{RW}, v')}{\mathsf{set}(l, v)/(s,m) \Downarrow \{(s, m[l \mapsto (\mathsf{RW}, v)])\}}\ \textsc{Set}$$

$$\frac{}{\mathsf{ignore}(v)/(s,m) \Downarrow \{(s \setminus \mathbf{res}, m)\}}\ \textsc{Ignore} \qquad \frac{}{\mathsf{add}(v_1, v_2)/(s,m) \Downarrow \{(s[\mathbf{res} \mapsto v_1 + v_2], m)\}}\ \textsc{Add}$$

$$\frac{m(l_1) = (\mathsf{RW}, v_1) \qquad v_1 \neq \bot}{\mathsf{inplaceAdd}(l_1, v_2)/(s,m) \Downarrow \{(s, m[l_1 \mapsto (\mathsf{RW}, v_1 + v_2)])\}}\ \textsc{InplaceAdd}$$

$$\frac{}{\mathsf{heapAlloc}()/(s,m) \Downarrow \{(s[\mathbf{res} \mapsto l], m[l \mapsto (\mathsf{RW}, \bot)]) \mid l \notin \mathsf{dom}(m)\}}\ \textsc{HeapAlloc}$$

$$\frac{m(l) = (\mathsf{RW}, v)}{\mathsf{free}(l)/(s,m) \Downarrow \{(s, m \setminus l)\}}\ \textsc{Free}$$

$$\frac{\begin{array}{c} n_{\text{start}} \leq n_{\text{stop}} \qquad t/(s[i \mapsto n_{\text{start}}], m) \Downarrow Q_1 \\ \forall (s_1, m_1) \in Q_1, (\mathbf{for}^{\mathbf{seq}}\ (i \in \mathbf{range}(n_{\text{start}} + n_{\text{step}}, n_{\text{stop}}, n_{\text{step}}))\ t)/(s_1, m_1) \Downarrow Q \end{array}}{(\mathbf{for}^{\mathbf{seq}}\ (i \in \mathbf{range}(n_{\text{start}}, n_{\text{stop}}, n_{\text{step}}))\ t)/(s,m) \Downarrow Q}\ \textsc{ForIter}$$

$$\frac{n_{\text{start}} > n_{\text{stop}}}{(\mathbf{for}^{\mathbf{seq}}\ (i \in \mathbf{range}(n_{\text{start}}, n_{\text{stop}}, n_{\text{step}}))\ t)/(s,m) \Downarrow \{(s,m)\}}\ \textsc{ForEnd}$$

$$\frac{m = m_{\text{RO}} \uplus \biguplus_{i \in R} m_i \qquad \forall i \in R,\ t/(s, \mathsf{IntoRO}(m_{\text{RO}}) \uplus m_i) \Downarrow Q_i}{(\mathbf{for}^{\mathbf{par}}\ (i \in R)\ t)/(s,m) \Downarrow \left\{(s,m') \;\middle|\; \exists \overline{s_i'}, \exists \overline{m_i'}, \begin{array}{c} m' = m_{\text{RO}} \uplus \biguplus_{i \in R} m_i' \\ \wedge\ \forall i \in R, (s_i, \mathsf{IntoRO}(m_{\text{RO}}) \uplus m_i') \in Q_i \end{array}\right\}}\ \textsc{ForPar}$$

Fig. 31. Semantics of the Optiλ internal language in omni-big-step style as explained in appendix C. Other arithmetic built-in functions follow the pattern of Add or InplaceAdd.

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 fresh α' added to E_{frac}. Our algorithm then adds the binding $\alpha := \alpha'$ into σ. The interest of extracting a carved fraction from βH rather than consuming the whole read-only permission βH is that the left-over fraction remains available in Γ, allowing to match other resources of the form $\alpha'' H$ that might appear in the other elements from Γ'.

E Details of Context Specialization

In Section 4.7, we introduced the operator $\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ to adapt a function contract for a specific call. We here present its (fairly technical) definition.

We formally define $\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ by using an auxiliary recursive function of the form $\mathsf{Specialize}'_{E_0}\{\sigma\}(E_1, \Gamma)$, where E_0 denotes the pure part of Γ_0, and where E_1 denotes an accumulator. The operation assumes $\mathsf{dom}(\sigma)$ to be included in set of keys of $\Gamma.\mathsf{pure}$. The definition of $\mathsf{Specialize}'$ relies itself on two auxiliary operators. The operator $\mathsf{TypeOf}(v, E)$ returns the unique type τ such that the pure expression v has type τ in the pure context E (this corresponds to the judgment written $E \vdash v : \tau$ detailed in Section 4.4). The operator $\mathsf{Unify}(E, \tau, \tau')$ tries to unify the types τ and τ' by treating variables in E as unification variables in τ'. If it succeeds, it returns a map between the resolved unification variables to their values found by unification. Some variables from E may remain unresolved and do not appear in this map.

$$
\begin{aligned}
\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma) &= \mathsf{Specialize}'_{\Gamma_0.\mathsf{pure}}\{\sigma\}(\varnothing, \Gamma) \\
\mathsf{Specialize}'_{E_0}\{\varnothing\}(E_1, \langle E_2 \mid F\rangle) &= \langle (E_1, E_2) \mid F\rangle \\
\mathsf{Specialize}'_{E_0}\{\sigma\}(E_1, \langle x : \tau, E_2 \mid F\rangle) &= \begin{cases}
\text{if } x \notin \mathsf{dom}(\sigma): \\
\quad \mathsf{Specialize}'_{E_0}\{\sigma\}((E_1, x : \tau), \langle E_2 \mid F\rangle) \\
\text{otherwise } \sigma \text{ decomposes as } (x := v) \uplus \sigma': \\
\quad \text{let } \tau_0 = \mathsf{TypeOf}(v, E_0) \text{ in} \\
\quad \text{let } \sigma'' = \mathsf{Unify}(E_1, \tau_0, \tau) \text{ in} \\
\quad \text{let } E_1' = \mathsf{Specialize}_{E_0}\{\sigma''\}([E_1]).\mathsf{pure} \text{ in} \\
\quad \text{let } \Gamma' = \mathsf{Subst}\{\sigma'' \uplus (x := v)\}(\langle E_2 \mid F\rangle) \text{ in} \\
\quad \mathsf{Specialize}'_{E_0}\{\sigma'\}(E_1', \Gamma')
\end{cases}
\end{aligned}
$$

F Details of Triple Minimization

In Section 6.6, we gave high-level properties of the triple minimization operator. We here give implementation details.

Minimize is computed by looking at the usage of each resource:

- For a resource H that appear as uninit in the usage map, we put $\mathsf{Uninit}(H)$ in $\hat{F}$.
- For resources that appear as splittedFrac in the usage map, we can give an arbitrarily small fraction to t, and keep the rest in the frame.
- For resources that appear as joinedFrac in the usage map, we can completely place them in the frame.

These two last points, some care is needed for the minimized postcondition $\hat{F}'$, because new subfractions might have been created by t and were immediately merged into a resource that is now not given anymore. You can find below some examples of minimization made by our version

of Minimize:

$\Gamma(y)$	$\Gamma'(y)$	$\Delta(y)$	E^{fracs}	$\hat{F}$	$\hat{F}'$	F^{framed}
H	H	$y \notin \Delta$	$\varnothing$	$\varnothing$	$\varnothing$	$y : H$
H	$\varnothing$	full	$\varnothing$	$y : H$	$\varnothing$	$\varnothing$
H	$\varnothing$	uninit	$\varnothing$	$y : \mathsf{Uninit}(H)$	$\varnothing$	$\varnothing$
H	H	splittedFrac	$\alpha : \text{frac}$	$y' : \alpha H$	$y' : \alpha H$	$y : (1-\alpha)H$
αH	αH	splittedFrac	$\beta : \text{frac}$	$y' : \beta H$	$y' : \beta H$	$y : (\alpha-\beta)H$
$(\alpha-\beta)H$	αH	splittedFrac	$\gamma : \text{frac}$	$y' : \gamma H$	$y' : \gamma H,\ y_\beta : \beta H$	$y : (\alpha-\beta-\gamma)H$
αH	$(\alpha-\beta)H$	splittedFrac	$\gamma : \text{frac}$	$y' : \gamma H$	$y' : (\gamma-\beta)H$	$y : (\alpha-\gamma)H$
$(\alpha-\beta_1-\beta_2)H$	$(\alpha-\beta_1-\beta_3)H$	splittedFrac	$\gamma : \text{frac}$	$y' : \gamma H$	$y' : (\gamma-\beta_3)H,\ y_2 : \beta_2 H$	$y : (\alpha-\gamma)H$
$(\alpha-\beta)H$	αH	joinedFrac	$\varnothing$	$\varnothing$	$y' : \beta H$	$y : (\alpha-\beta)H$
$(\alpha-\beta_1-\beta_2-\beta_3)H$	$(\alpha-\beta_2)H$	joinedFrac	$\varnothing$	$\varnothing$	$y_1 : \beta_1 H,\ y_3 : \beta_3 H$	$y : (\alpha-\beta_1-\beta_2-\beta_3)H$

Algorithmically, Minimize can be defined by iterating over its first argument.
Start with $E^{\text{fracs}} = \varnothing$, $\hat{F} = \varnothing$, $\hat{F}' = \Gamma'$.linear and $F^{\text{framed}} = \varnothing$.
For each binding $y : H$ in Γ, look up y in Δ:

- If y is not in Δ, add $y : H$ in F^{framed} and remove it from $\hat{F}'$ (it must exist there by the invariants of triples).
- If y : full is in Δ, add $y : H$ in $\hat{F}$.
- If y : uninit is in Δ, add $y : \mathsf{Uninit}(H)$ in $\hat{F}$.
- If y : splittedFrac is in Δ, decompose H as $(\alpha - \beta_1 - \ldots - \beta_n)H'$. It is always possible since $\alpha = 1$ is allowed and the list of β_i can be empty. Create a fresh fraction γ and place it in E^{fracs}. Add $y' : \gamma H'$ in $\hat{F}$ and $y : (\alpha - \beta_1 - \ldots - \beta_n - \gamma)H'$ in F^{framed}. Replace the binding $y : (\alpha - \delta_1 - \ldots - \delta_m)H'$ in $\hat{F}'$ by the following: try to pair δ_i with a matching β_j. For each unmatched β_i, add a binding $y_i'' : \beta_i H'$ to $\hat{F}'$. These correspond to the subfractions that were created by t and merged into y. Let the unmatched δ_i form the list of $\tilde{\delta}_i$. These correspond to the subfractions consumed by t and not given back. Add the binding $y' : (\gamma - \tilde{\delta}_1 - \ldots - \tilde{\delta}_{\tilde{m}})H$ to $\hat{F}'$.
- If y : joinedFrac is in Δ, decompose H as $(\alpha - \beta_1 - \ldots - \beta_n)H'$. Add $y : H$ in F^{framed}. Remove the binding $y : (\alpha - \delta_1 - \ldots - \delta_m)H'$ in $\hat{F}'$. Given that joinedFrac usage are only created by CloseFracs and given how the CloseFracs algorithm works, each δ_i will necessarily match one of the β_j, however there will be some β_i that are not matched. For each unmatched β_i, add the binding $y_i' : \beta_i H'$ in $\hat{F}'$.

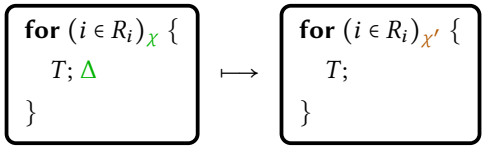
The next two sections give details for two applications of this Minimize operator: typechecking subexpressions and loop contract minimization.

G Details of Loop Minimization

Section 6.7 introduced the procedure for minimization of loop contracts. The implementation details appear in Figure 32, where the operator IntoRO converts resources to their read-only form (for a formal definition, we refer to [Bertholon 2025, Appendix A]). Intuitively, the procedure relies on the Minimize operator to minimize the exclusive part of the loop contract, and it tries to reduce the footprint of the shared part of the loop contract by taking arbitrary subfractions and using shared reads whenever possible.

H Example Typechecking of Subexpressions

This section presents an example application of the Subexpr from Section 6.8 and repeated below:



with:

$$(E^{\text{fracs}}, \hat{F}, \hat{F}', \_) = \text{Minimize}(\chi.\text{excl.pre}, \text{Subst}\{\varsigma^{-1}\}(\chi.\text{excl.post}), \Delta)$$

$$\langle E^{\text{RO}} \mid F^{\text{RO}} \rangle = \text{IntoRO}((\chi.\text{shrd.inv.linear} \restriction \cdot\Delta.\text{splittedFrac}) \star (\chi.\text{shrd.reads} \restriction \cdot\Delta.\text{splittedFrac}))$$

$$\chi' = \begin{cases} \text{shrd.reads} = F^{\text{RO}} \star (\chi.\text{shrd.reads} \restriction \cdot\Delta.\text{alter}) \\ \text{shrd.inv} = \langle \chi.\text{shrd.inv.pure} \restriction \cdot\Delta.\text{required} \mid \chi.\text{shrd.inv.linear} \restriction \cdot\Delta.\text{alter} \rangle \\ \text{excl.pre} = \langle \chi.\text{excl.pre.pure} \restriction \cdot\Delta.\text{required} \mid \hat{F} \rangle \\ \text{excl.post} = \text{Subst}\{\varsigma\}(\langle \chi.\text{excl.post.pure} \mid \hat{F}' \rangle) \\ \text{vars} = \chi.\text{vars} \restriction \cdot(\text{usedVars}(\chi'.\text{shrd}) \cup \text{usedVars}(\chi'.\text{excl}) \cup \Delta.\text{required}), E^{\text{RO}}, E^{\text{fracs}} \end{cases}$$

Fig. 32. The basic transformation `Loop.minimize`. The Minimize operation is that defined for triples in Section 6.6. usedVars(X) is the set of all variables used in X at least once.

i	0	1	2	3
Γ_i.pure	p, q, c : ptr(int)	p, q, c : ptr(int)	p, q, c : ptr(int)	p, q, c : ptr(int) α : frac
Γ_i.linear	$y_p : p \rightsquigarrow$ Cell $y_q : q \rightsquigarrow$ Cell $y_c : c \rightsquigarrow$ Cell	$y_p : p \rightsquigarrow$ Cell $y_q : q \rightsquigarrow$ Cell $y_c : c \rightsquigarrow$ Cell	$y_p : p \rightsquigarrow$ Cell $y_q : q \rightsquigarrow$ Cell	$y_p : (1-\alpha)(p \rightsquigarrow$ Cell) $y_q : q \rightsquigarrow$ Cell
t_i	q	get_incr(c)	get(p)	get(p)
Δ_i	q : required **res** : ensured	c : required y_c : full y_c' : produced **res** : ensured	p : required y_p : splittedFrac **res** : ensured	p : required y_p : splittedFrac **res** : ensured
E_i^{fracs}	$\varnothing$	$\varnothing$	α : frac	β : frac
$\hat{F}_i$	$\varnothing$	$y_c : c \rightsquigarrow$ Cell	$\alpha(p \rightsquigarrow$ Cell)	$\beta(p \rightsquigarrow$ Cell)
$\hat{F}_i'$	$\varnothing$	$y_c' : c \rightsquigarrow$ Cell	$\alpha(p \rightsquigarrow$ Cell)	$\beta(p \rightsquigarrow$ Cell)
F_i^{framed}	$y_p : p \rightsquigarrow$ Cell $y_q : q \rightsquigarrow$ Cell $y_c : c \rightsquigarrow$ Cell	$y_p : p \rightsquigarrow$ Cell $y_q : q \rightsquigarrow$ Cell	$y_p : (1-\alpha)(p \rightsquigarrow$ Cell) $y_q : q \rightsquigarrow$ Cell	$y_p : (1-\alpha-\beta)(p \rightsquigarrow$ Cell) $y_q : q \rightsquigarrow$ Cell
$\hat{\Gamma}_i'$.pure	**res** $:= q$: ptr(int)	**res** : int	**res** : int	**res** : int
Γ_p.pure	p, q, c : ptr(int), $x_0 := q$: ptr(int), x_1, x_2, x_3 : int			
Γ_p.linear	$y_p : p \rightsquigarrow$ Cell, $y_q : q \rightsquigarrow$ Cell, $y_c' : c \rightsquigarrow$ Cell			

Fig. 33. Example of an application of the SUBEXPR rule on an expression $\hat{\mathcal{E}}[q, \text{get\_incr(c)}, \text{get}(p), \text{get}(p)]$, in a context $\langle p, q, c : \text{ptr(int)} \mid y_p : p \rightsquigarrow \text{Cell}, y_q : q \rightsquigarrow \text{Cell}, y_c : c \rightsquigarrow \text{Cell} \rangle$.

$$\frac{\begin{array}{c}\textsc{Subexpr}\\ \forall i \in [1,n].\quad \{\Gamma_{i-1}\}\ t_i^{\Delta_i}\ \{\Gamma_i'\} \quad\wedge\quad (E_i^{\text{fracs}}, \hat{F}_i, \hat{F}_i', F_i^{\text{framed}}) = \text{Minimize}(\Gamma_{i-1}, \Gamma_i', \Delta_i) \quad\wedge\quad x_i \text{ fresh}\\ \forall i \in [1,n].\quad \Gamma_i = \langle\, \Gamma_i.\text{pure}, E_i^{\text{fracs}} \mid F_i^{\text{framed}} \,\rangle \quad\wedge\quad \hat{\Gamma}_i' = \langle\, \Gamma_i'.\text{pure} \mid \cdot \Delta_i.\text{ensured} \mid \hat{F}_i' \,\rangle\\ \Gamma_p = \text{CloseFracs}^{\Delta_p}(\Gamma_n \circledast \circledast_{i\in[1,n]} \text{Rename}\{\textbf{res} := x_i\}(\hat{\Gamma}_i'))\\ \{\Gamma_p\}\ \hat{\mathcal{E}}[x_1, ..., x_n]^{\Delta_q}\ \{\Gamma_q\}\\ \Delta = \text{Rename}\{\textbf{res} := x_1\}(\Delta_1); ...; \text{Rename}\{\textbf{res} := x_n\}(\Delta_n); \Delta_p; \Delta_q\end{array}}{\{\Gamma_0\}\ \hat{\mathcal{E}}[t_1, ..., t_n]^{\Delta}\ \{\Gamma_q\}}$$

The example consists of a multi-evaluation-context $\hat{\mathcal{E}}$, which could be a function call, featuring 4 subexpression holes: $\hat{\mathcal{E}}[q, \text{get\_incr(c)}, \text{get}(p), \text{get}(p)]$. This expression is typechecked in a typing context: $\langle p, q, c : \text{ptr(int)} \mid y_p : p \rightsquigarrow \text{Cell}, y_q : q \rightsquigarrow \text{Cell}, y_c : c \rightsquigarrow \text{Cell}\rangle$.

Figure 33 shows the typechecking steps. The figure includes 4 columns, describing the steps associated with each of the 4 subterms. The rows explain how the metavariables from the rule Subexpr are instantiated. In particular, observe how the two subexpressions $\text{get}(p)$ both have read-only access to the same resource $H = p \rightsquigarrow \text{Cell}$. As the details in the figure show, the first $\text{get}(p)$, 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 $\text{get}(p)$.

I Soundness of the Algorithmic Rule for Typechecking for Loops

This section focuses on the correctness argument for our algorithmic typing rule for **for** loops. This rule is the most remote compared with known presentations of separation logic.

Let us recall this algorithmic typing rule for handling **for** loops:

$$\frac{\begin{array}{c}\Gamma_{\text{pre}}^{\text{out}} = [\chi.\text{vars}] \circledast (\circledast_{i\in R}\, \chi.\text{excl.pre}) \circledast \chi.\text{shrd.reads} \circledast \text{Subst}\{i := R.\text{start}\}(\chi.\text{shrd.inv})\\ (E_{\text{frac}}, \sigma^{\text{out}}, F) = \Gamma_0 \ominus \Gamma_{\text{pre}}^{\text{out}}\\ \Gamma_{\text{pre}}^{\text{in}} = [\chi.\text{vars}] \circledast \chi.\text{excl.pre} \circledast \tfrac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv}\\ \{\!\{[\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \Gamma_{\text{pre}}^{\text{in}}\}\!\}\ t\ \{\!\{\Gamma_{\text{post}}^{\text{in}}\}\!\}\\ (\sigma_{\text{post}}^{\text{in}}, \varnothing) = \Gamma_{\text{post}}^{\text{in}} \boxminus \chi.\text{excl.post} \circledast \tfrac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \text{Subst}\{i := i + R.\text{step}\}(\chi.\text{shrd.inv})\\ \Gamma_{\text{post}}^{\text{out}} = \text{Subst}\{\sigma^{\text{out}}\}((\circledast_{i\in R}\, \chi.\text{excl.post}) \circledast \chi.\text{shrd.reads} \circledast \text{Subst}\{i := R.\text{end}\}(\chi.\text{shrd.inv}))\\ \Gamma_r = \text{CloseFracs}([\Gamma_0.\text{pure}, E_{\text{frac}}] \circledast F \circledast \Gamma_{\text{post}}^{\text{out}})\\ \pi = \textbf{par} \implies \text{parallelizable}(\chi)\end{array}}{\{\!\{\Gamma_0\}\!\}\ \textbf{for}^{\pi}\ (i \in R)_{\chi}\ t\ \{\!\{\Gamma_r\}\!\}}\ \textsc{For}$$

Let us show the soundness of this For rule supposing that the two common separation logic typing rules for sequential and parallel **for** loops shown below hold.

$$\frac{\{[E_0, i : \text{int}, i \in R] \circledast \Gamma_{\text{inv}}(i)\}\ t\ \{\Gamma_{\text{inv}}(i + R.\text{step})\}}{\{[E_0] \circledast \Gamma_{\text{inv}}(R.\text{start})\}\ \textbf{for}^{\textbf{seq}}\ (i \in R)\ t\ \{\Gamma_{\text{inv}}(R.\text{end})\}}\ \textsc{ForSeq}$$

$$\frac{\{[E_0, i : \text{int}, i \in R] \circledast \Gamma_{\text{pre}}(i)\}\ t\ \{\Gamma_{\text{post}}(i)\}}{\{[E_0] \circledast \circledast_{i\in R}\, \Gamma_{\text{pre}}(i)\}\ \textbf{for}^{\textbf{par}}\ (i \in R)\ t\ \{\circledast_{i\in R}\, \Gamma_{\text{post}}(i)\}}\ \textsc{ForPar}$$

Before starting the proof let us state two technical lemmas. The first lemma states that we can always specialize a logical triple into a more specific triple by partially instantiating its precondition. This lemma is analogous to the specialization of a logical theorem on specific parameters.

$$\{\Gamma_0 \circledast \Gamma_1\}\ t\ \{\Gamma_2\} \implies \{\Gamma_0 \circledast \text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma_1)\}\ t\ \{\text{Subst}\{\sigma\}(\Gamma_2)\}$$

The second lemma states that the specialized version of a context entails itself.

$$\text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma) \Rightarrow \Gamma$$

Now let us prove the main result of this section: the algorithmic typing rule For can be derived from the standard separation logic rules ForSeq and ForPar. We perform a case analysis on whether the loop is parallel or not which corresponds to the component π:

- If π = **seq** (the loop is sequential), our For rule can be derived from the rule ForSeq by placing in Γ_{inv} all the exclusive per-iteration resources produced by iterations strictly before j, all the exclusive per-iterations resources consumed by iterations after and including j, and all the resources shared across iterations. More formally, we use the instantiations:

$$\begin{aligned}
E_0 &= \Gamma_0.\text{pure} \\
\Gamma_{\text{inv}} &= \mathbf{fun}(j) \mapsto \text{Subst}\{\sigma^{\text{inv}}\}(\\
&\qquad \circledast_{i \in \mathbf{range}(R.\text{start}, j, R.\text{step})}\ \chi.\text{excl.post} \circledast \\
&\qquad \bigstar_{i \in \mathbf{range}(j, R.\text{stop}, R.\text{step})}\ \chi.\text{excl.pre.linear} \circledast \\
&\qquad \chi.\text{shrd.reads} \circledast \\
&\qquad \text{Subst}\{i := j\}(\chi.\text{shrd.inv})) \\
\sigma^{\text{inv}} &= \sigma^{\text{out}} \setminus \text{dom}(\chi.\text{shrd.inv.pure})
\end{aligned}$$

From our rule For, we know that $\{[\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \Gamma^{\text{in}}_{\text{pre}}\}\ t\ \{\Gamma^{\text{in}}_{\text{post}}\}$. By the triple-specialization lemma applied with σ^{inv}, we therefore know that $\{[\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}})\}\ t\ \{\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}})\}$. By adding a frame $\Gamma^{\text{in}}_{\text{frame}}$, defined as:

$$\begin{aligned}
\Gamma^{\text{in}}_{\text{frame}} = \text{Subst}\{\sigma^{\text{inv}}\}(& \\
&\circledast_{i \in \mathbf{range}(R.\text{start}, j, R.\text{step})}\ \chi.\text{excl.post} \circledast \\
&\bigstar_{i \in \mathbf{range}(j + R.\text{step}, R.\text{stop}, R.\text{step})}\ \chi.\text{excl.pre.linear} \circledast \\
&\tfrac{R.\text{len}-1}{R.\text{len}}\chi.\text{shrd.reads})
\end{aligned}$$

we obtain $\{[\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) \circledast \Gamma^{\text{in}}_{\text{frame}}\}\ t\ \{\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}}) \circledast \Gamma^{\text{in}}_{\text{frame}}\}$. Then, by the consequence rule, we can deduce the required hypothesis $\{[E_0, i : \text{int}, i \in R] \circledast \Gamma_{\text{inv}}(i)\}\ t\ \{\Gamma_{\text{inv}}(i + R.\text{step})\}$ for applying ForSeq by proving two entailments:
  - $[E_0, i : \text{int}, i \in R] \circledast \Gamma_{\text{inv}}(i) \Rightarrow [\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) \circledast \Gamma^{\text{in}}_{\text{frame}}$
    We know that $\Gamma^{\text{in}}_{\text{pre}} = [\chi.\text{vars}] \circledast \chi.\text{excl.pre} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv}$. By definition of Specialize and σ^{inv}, we have $\text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) = \text{Subst}\{\sigma^{\text{inv}}\}(\chi.\text{excl.pre.linear} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv})$.
    Then, we can conclude by distributing Subst over separated conjunctions, splitting the fraction $\frac{1}{R.\text{len}}\chi.\text{shrd.reads}$, and by using the following equality:

$$\bigstar_{i \in \mathbf{range}(j, R.\text{stop}, R.\text{step})}\ \chi.\text{excl.pre.linear} = \text{Subst}\{i := j\}(\chi.\text{excl.pre.linear}) \star \bigstar_{i \in \mathbf{range}(j + R.\text{step}, R.\text{stop}, R.\text{step})}\ \chi.\text{excl.pre.linear}$$

  - $\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}}) \circledast \Gamma^{\text{in}}_{\text{frame}} \Rightarrow \Gamma_{\text{inv}}(i + R.\text{step})$
    To prove this entailment we use the fact that we know from the hypothesis $(\sigma^{\text{in}}_{\text{post}}, \varnothing) = \Gamma^{\text{in}}_{\text{post}} \boxminus \chi.\text{excl.post} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \text{Subst}\{i := i + R.\text{step}\}(\chi.\text{shrd.inv})$ that $\Gamma^{\text{in}}_{\text{post}} \Rightarrow \chi.\text{excl.post} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \text{Subst}\{i := i + R.\text{step}\}(\chi.\text{shrd.inv})$. Thus, $\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}}) \Rightarrow \text{Subst}\{\sigma^{\text{inv}}\}(\chi.\text{excl.post} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \text{Subst}\{i := i + R.\text{step}\}(\chi.\text{shrd.inv}))$.

Then, we can conclude by distributing Subst over separated conjunctions, merging the fraction $\frac{1}{R.\mathsf{len}}\chi.\mathsf{shrd.reads}$, and by using the following equality:

$$\circledast_{i\in\mathbf{range}(R.\mathsf{start},\,j,\,R.\mathsf{step})}\ \chi.\mathsf{excl.post} \circledast \mathsf{Subst}\{i := j + R.\mathsf{step}\}(\chi.\mathsf{excl.post}) = \circledast_{i\in\mathbf{range}(R.\mathsf{start},\,j+R.\mathsf{step},\,R.\mathsf{step})}\ \chi.\mathsf{excl.post}$$

Now, we know by the rule ForSeq that $\{[E_0] \circledast \Gamma_{\mathsf{inv}}(R.\mathsf{start})\}\ \mathbf{for}^{\mathbf{seq}}\ (i \in R)\ t\ \{\Gamma_{\mathsf{inv}}(R.\mathsf{end})\}$.
To conclude, we frame the linear resource set F (that comes from the rule For itself) and prove the following entailments:

- $\Gamma_0 \Rightarrow [E_0] \circledast \Gamma_{\mathsf{inv}}(R.\mathsf{start}) \circledast F$
  We know by the subtraction $\Gamma_0 \ominus \Gamma^{\mathsf{out}}_{\mathsf{pre}}$ that $\Gamma_0 \Rightarrow [E_{\mathsf{frac}}] \circledast \mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\Gamma^{\mathsf{out}}_{\mathsf{pre}}) \circledast F$.
  As $\Gamma^{\mathsf{out}}_{\mathsf{pre}} = [\chi.\mathsf{vars}] \circledast (\circledast_{i\in R}\ \chi.\mathsf{excl.pre}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})$, by definition of Specialize and σ^{inv} and since we know σ^{out} covers all the pure bindings of $\Gamma^{\mathsf{out}}_{\mathsf{pre}}$, we have: $\mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\Gamma^{\mathsf{out}}_{\mathsf{pre}}) = \mathsf{Subst}\{\sigma^{\mathsf{inv}}\}((\bigstar_{i\in R}\ \chi.\mathsf{excl.pre.linear}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}} \restriction \mathsf{dom}(\chi.\mathsf{shrd.inv.pure})\}(\mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})))$.
  By the property that a specialized context entails itself, $\mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}} \restriction \mathsf{dom}(\chi.\mathsf{shrd.inv.pure})\}(\mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})) \Rightarrow \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})$.
  Therefore, $\Gamma_0 \Rightarrow [\Gamma_0.\mathsf{pure}] \circledast \mathsf{Subst}\{\sigma^{\mathsf{inv}}\}((\bigstar_{i\in R}\ \chi.\mathsf{excl.pre.linear}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})) \circledast F$.
  To conclude, we can introduce $\chi.\mathsf{excl.post}$ and find $\Gamma_{\mathsf{inv}}(R.\mathsf{start})$ using the equality $\varnothing = \circledast_{i\in\mathbf{range}(R.\mathsf{start},R.\mathsf{start},R.\mathsf{step})}\ \chi.\mathsf{excl.post}$.
- $\Gamma_{\mathsf{inv}}(R.\mathsf{end}) \circledast F \Rightarrow \Gamma_r$
  We know that $\Gamma_r = \mathsf{CloseFracs}([\Gamma_0.\mathsf{pure}, E_{\mathsf{frac}}] \circledast F \circledast \mathsf{Subst}\{\sigma^{\mathsf{out}}\}((\circledast_{i\in R}\ \chi.\mathsf{excl.post}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{end}\}(\chi.\mathsf{shrd.inv})))$. By scoping rules of the loop contracts, bindings of $\chi.\mathsf{shrd.inv}$ cannot occur in other elements of χ. Therefore, in this context σ^{out} and σ^{inv} produce the same substitution.
  To conclude, we use the fact that the range $\mathbf{range}(R.\mathsf{stop}, R.\mathsf{stop}, R.\mathsf{step})$ of the iterated separating conjunction over $\chi.\mathsf{excl.pre.linear}$ from $\Gamma_{\mathsf{inv}}(R.\mathsf{stop})$ is empty and notice that for any context Γ, $\Gamma \Leftrightarrow \mathsf{CloseFracs}(\Gamma)$.

- If $\pi = \mathbf{par}$ (the loop is parallel), then the rule For is obtained from the rule ForPar. Since $\pi = \mathbf{par}$, we know from the rule that $\chi.\mathsf{shrd.inv}$ must be empty. Therefore we can use the following instantiations for E_0, Γ_{pre} and Γ_{post}:

$$\begin{aligned} E_0 &= \Gamma_0.\mathsf{pure} \\ \Gamma_{\mathsf{pre}} &= \mathbf{fun}(i) \mapsto [\chi.\mathsf{vars}] \circledast \chi.\mathsf{excl.pre} \circledast \tfrac{1}{R.\mathsf{len}}\chi.\mathsf{shrd.reads} \\ \Gamma_{\mathsf{post}} &= \mathbf{fun}(i) \mapsto \chi.\mathsf{excl.post} \circledast \tfrac{1}{R.\mathsf{len}}\chi.\mathsf{shrd.reads} \end{aligned}$$

  We directly have $\Gamma^{\mathsf{in}}_{\mathsf{pre}} = \Gamma_{\mathsf{pre}}(i)$. From the hypothesis $(\sigma^{\mathsf{in}}_{\mathsf{post}}, \varnothing) = \Gamma^{\mathsf{in}}_{\mathsf{post}} \boxminus \chi.\mathsf{excl.post} \circledast \frac{1}{R.\mathsf{len}}\chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := i + R.\mathsf{step}\}(\chi.\mathsf{shrd.inv})$, we obtain $\Gamma^{\mathsf{in}}_{\mathsf{post}} \Rightarrow \Gamma_{\mathsf{post}}(i)$. From the hypothesis $\{[\Gamma_0.\mathsf{pure}, i : \mathsf{int}, i \in R] \circledast \Gamma^{\mathsf{in}}_{\mathsf{pre}}\}\ t\ \{\Gamma^{\mathsf{in}}_{\mathsf{post}}\}$, we can therefore deduce $\{[E_0, i : \mathsf{int}, i \in R] \circledast \Gamma_{\mathsf{pre}}(i)\}\ t\ \{\Gamma_{\mathsf{post}}(i)\}$ and apply the rule ForPar.
  Now, we know by the rule ForPar that $\{[E_0] \circledast \circledast_{i\in R}\Gamma_{\mathsf{pre}}(i)\}\ \mathbf{for}^{\mathbf{par}}\ (i \in R)\ t\ \{\circledast_{i\in R}\Gamma_{\mathsf{post}}(i)\}$.
  By the triple-specialization lemma, we have:
  $\{[E_0] \circledast \circledast_{i\in R}\mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\Gamma_{\mathsf{pre}}(i))\}\ \mathbf{for}^{\mathbf{par}}\ (i \in R)\ t\ \{\mathsf{Subst}\{\sigma^{\mathsf{out}}\}(\circledast_{i\in R}\Gamma_{\mathsf{post}}(i))\}$.
  To conclude, we frame the linear resource set F (that comes from the rule For itself) and prove the following entailments:
  - $\Gamma_0 \Rightarrow [E_0] \circledast \mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\circledast_{i\in R}\Gamma_{\mathsf{pre}}(i)) \circledast F$

We know by the subtraction $\Gamma_0 \ominus \Gamma_{\text{pre}}^{\text{out}}$ that $\Gamma_0 \Rightarrow [E_{\text{frac}}] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{out}}\}(\Gamma_{\text{pre}}^{\text{out}}) \circledast F$. As $\Gamma_{\text{pre}}^{\text{out}} = [\chi.\text{vars}] \circledast (\circledast_{i \in R}\, \chi.\text{excl.pre}) \circledast \chi.\text{shrd.reads}$, we can conclude by splitting $\chi.\text{shrd.reads}$ into $\bigstar_{i \in R} \frac{1}{R.\text{len}} \chi.\text{shrd.reads}$, and logically reordering separated conjunctions.

– $\text{Subst}\{\sigma^{\text{out}}\}(\circledast_{i \in R}\, \Gamma_{\text{post}}(i)) \circledast F \Rightarrow \Gamma_r$
We know that $\Gamma_r = \text{CloseFracs}([\Gamma_0.\text{pure}, E_{\text{frac}}] \circledast F \circledast \text{Subst}\{\sigma^{\text{out}}\}((\circledast_{i \in R}\, \chi.\text{excl.post}) \circledast \chi.\text{shrd.reads}))$. We can directly conclude by merging $\bigstar_{i \in R} \frac{1}{R.\text{len}} \chi.\text{shrd.reads}$ from $\bigstar_{i \in R} \Gamma_{\text{post}}(i)$ into $\chi.\text{shrd.reads}$, and using the fact that for all context Γ, $\Gamma \Leftrightarrow \text{CloseFracs}(\Gamma)$.