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

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

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

This paper presents OptiTrust, an interactive source-to-source optimization framework that operates on general-purpose C code. The programmer develops a script describing a series of code transformations. The framework provides continuous feedback in the form of human-readable *diff* s over conventional C code. OptiTrust supports advanced code transformations, including transformations exploited by the state-of-the-art DSL tools Halide and TVM, and transformations beyond the reach of existing tools. OptiTrust also supports user-defined transformations, as well as defining complex transformations by composition of simpler transformations. Crucially, to check the validity of code transformations, OptiTrust leverages a *static resource analysis* in a simplified form of Separation Logic. Starting from user-provided annotations on functions and loops, our analysis deduces precise resource usage throughout the code. Through several case studies, we demonstrate how OptiTrust can be employed to produce state-of-the-art, high-performance programs.

1 INTRODUCTION

1.1 Motivation

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

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

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

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

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

the optimized code is hard to maintain through hardware and software evolutions. Third, the rewriting process is error-prone: not only every manual code edition might introduce a bug, but the code complexity also increases, especially when introducing parallelism. These three factors are exacerbated by the fact that optimizations typically make code size grow by an order of magnitude (for example: the optimized code for our following matrix multiplication case study is 7× bigger; 15× bigger for our box blur case study).

In summary, neither fully automatic nor fully manual approaches are satisfying for generating high performance code. Both machine automation and human insight are needed in the optimization process. Before reviewing tools for semi-automatic code optimization, let us introduce a number of qualitative properties on which to evaluate these tools.

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

1.2 Related Work

Halide [Ragan-Kelley et al. 2013] is an industrial-strength domain-specific compiler for image processing, used e.g. to optimize code of Photoshop and YouTube. Halide popularized the idea of separating an *algorithm* describing what to compute from a *schedule* describing how to optimize the computation. This separation makes it easy to try different schedules. TVM [Chen et al. 2018] is a tool directly inspired by Halide, but tuned for machine learning applications. Halide and TVM are inherently limited to their DSLs. They do not support higher-order composition of transformations, and are not extensible [Barham and Isard 2019; Ragan-Kelley 2023]. Moreover, understanding their output is difficult as the applied transformations are not detailed to the user. Interactive scheduling systems have been proposed to mitigate this difficulty [Ikarashi et al. 2021].

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

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

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

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

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

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

Table 1 summarizes the properties of the existing approaches, highlighting their diversity. The table is sorted by increasing generality. For the tools considered, this generality is negatively correlated with expressiveness, i.e., with how advanced the supported transformations are. Regarding generality, only Clava supports operating on general C code, yet provides absolutely no guarantees on semantics preservation. For each property considered, at least two tools show strengths on that property (above half score). However, even if we leave out the ambition of achieving mechanized proofs, each tool considered shows weaknesses on at least two properties (half score or less).

1.3 Contribution

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

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

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

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

To ensure that the user applies only semantic-preserving transformations, OptiTrust performs validity checks that leverage our *static resource analysis*, which concretely takes the form of a type checking algorithm, in a type system featuring linear resources. This type system may be thought of as a variant of the Rust type system, or as a scaled down version of Separation Logic [Reynolds 2002]. Our resource-based system aims to be similar in spirit to RefinedC [Sammler et al. 2021], a Separation Logic-based type system for C code, even though we have not implemented all the features of RefinedC yet.

For type-checking resources, functions and loops need to be equipped with *contracts* describing their resource usage. These contracts may be inserted either directly as no-op annotations in the C source code, or they may be inserted by dedicated commands as part of the transformation script. OptiTrust is able to automatically infer simple loop contracts, thus not all loops need to be annotated manually. Every OptiTrust transformation takes care of updating contracts in order to reflect changes in the code. In other words, a well-typed program remains well-typed after a transformation.

Currently, OptiTrust only automates the application of transformations and the checking of their validity, but we also plan to explore future work to guide the user towards useful optimizations.

1.4 Contents of the Paper

We first present the features of OptiTrust by means of example, in Section 2. Then, we present the construction of OptiTrust in four parts. In Section 3, we describe the overall architecture of the implementation. In Section 4, we explain the resource-based typechecker. In Section 5, we present a set of representative code transformations, illustrating in particular how resource information is exploited to justify correctness, and how loop contracts are maintained through transformations. Finally, we discuss related work in Section 6.

2 OPTITRUST BY EXAMPLE

In this section we demonstrate the features of OptiTrust through three case studies. In the first case study (section 2.1), we leverage OptiTrust to produce an optimized implementation of matrix multiplication, similar to that produced by the specialized compiler TVM. In the second case study (section 2.2), we leverage OptiTrust to produce an optimized implementation of an image processing operation, namely *row-based blur*, similar to the code that had been manually written as part of the OpenCV reference library. In the third case study (section 2.3), we leverage OptiTrust to optimize a particle simulation kernel, demonstrating how a high-level operations on 3D vectors can be refined into idiomatic high performance code.

2.1 Optimizing Matrix Multiplication

The aim of our matrix multiplication case study is to demonstrate the ability of OptiTrust to perform similar optimizations as specialized compilers. Concretely, we aim to produce code similar to that from the TVM matrix multiplication case study. TVM is an industrial-strength domain-specific compiler for machine learning that takes as input programs written in a DSL, and that relies on monolithic compilation passes to apply a given *schedule*, which describes an optimization strategy. The *schedule* for matrix multiplication has been written by an expert, aiming to produce code optimized for Intel CPUs. The schedule is presented in the TVM tutorial[1]. The optimized matrix multiplication code generated by TVM is expressed in LLVM IR. This code is essentially equivalent to the C code shown in Listing 3. Whereas TVM applies monolithic passes on a DSL, OptiTrust applies a series of local, source-to-source transformations, manipulating programs expressed in the general-purpose C language.

Annotated Code. We start from the C code presented in Listing 1: a naive, unoptimized implementation of matrix multiplication. To use OptiTrust, we annotate the code with resource contracts. For example, the `__reads` clauses indicate that the function `mm` reads a matrix `A` of size $m \times p$, as well as a matrix `B` of size $p \times n$. The `__modifies` clause that follows indices that the function may modify the contents of the matrix `C`, of size $m \times n$. The resource `C ⇝ Matrix2(m, n)` not only specifies that the matrix at address `C` in memory has size $m \times n$, it also yields a permission to modify this matrix. This resource is essentially equivalent to the permission to modify each of the cells of the matrix. It therefore corresponds to the resource described using two nested *iterated conjunctions* over *Cell* resources: $\star_{i \in 0..m} \star_{j \in 0..n}$ (`&C[i][j] ⇝ Cell`), where `p ⇝ Cell` describes the *ownership* of the cell at address `p`.

The body of the function `mm` features, as expected for a naive implementation of matrix multiplication, 3 nested loops. The two outer loops, on indices `i` and `j`, carry annotations to guide the resource type-checker. For example, the clause `__xmodifies("&C[i][j] ⇝ Cell")` indicates that `j`-th iteration of the loop on `j` *exclusively* (hence the letter `x`) requires the resource `&C[i][j] ⇝ Cell`, meaning that only the `j`-th iteration accesses `&C[i][j]`. The clause `__xmodifies("for j in 0..n -> &C[i][j] ⇝ Cell")` indicates that the `i`-th iteration of the loop on `i` exlusively requires the `i`-th row of the matrix `C`.

These loop annotations provide partial *loop contracts*; several other clauses are automatically inferred by OptiTrust. For example, OptiTrust infers that the inner loop, on index `k`, requires a clause of the form `__smodifies("sum ⇝ Cell")`. This permission asserts that every iteration of the loop on `k` *sequentially* (hence the letter `s`) requires access to the `sum` cell. OptiTrust also infers that each of the three loops require a read permission on the matrices `A` and `B`. This permission takes the form `__sreads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)")`, indicating that every loop iteration *shares* (hence the letter `s`) read access to the matrices `A` and `B`.

Transformation Script. In OptiTrust, optimizations are dictated by means of a script written in the OCaml programming language. Our script for matrix multiply is displayed in Listing 2. A transformation script generally consists of a series of calls to functions to the OptiTrust library. By convention, the last argument of a transformation always denotes a *target*. As detailed further on, a target provides a way to concisely and robustly refer to one or several code location. Certain transformations, such as `Loop.hoist_expr`, may take additional targets as argument, for example

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

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

Listing 1. Unoptimized code for matrix multiplication, in C code accompanied with resource annotations. The function `mm` multiplies the matrices `A` and `B` and stores the result in `C`. The function `mm1024` specializes input sizes to 1024. In our actual source code, array accesses take the form `A[MINDEX2(m, p, i, k)]` instead of just `A[i][k]`, because we need to keep track of matrix sizes. In the future, we plan to automatically propagate such size information, to ease the work of the programmer.

to describe where an instruction should be moved. For the reader not familiar with OCaml, `f x y` denote the call of `f` on the arguments `x` and `y`; the symbol `~` is used to provide optional (or named) arguments; `[x; y; z]` denotes a list; `(x, y, z)` denotes a tuple; `s1 ^ s2` denotes a string concatenation; and `let f x = e1 in e2` introduces a local function f.

The script from Listing 2 consists of 8 transformation steps. Each step may be executed interactively: with the cursor on a line starting with `!!`, the OptiTrust user can press (e.g.) the "F6" key in the VSCode editor to visualize the *diff* associated with the transformation on that line. All intermediate versions of the code consist of human-readable, executable C code. Only the final processing converts n-dimensional array accesses into the intricate index computations visible in Listing 3. The sole role of the `!!` operator is to delimit steps on which visualizing a *diff* is most relevant. Additionally, OptiTrust can produce a complete execution report in the form of an interactive tree, reporting *diff* not only for the top-level transformations, but also for all the more basic transformations that are leveraged in the process.[2]

Targets. As mentioned earlier, a transformations takes a target as parameters, to describe one or several location where the transformation should be applied. A target consists of a list of constraints (prefixed by "c") that is satisfied by code paths that go through nodes satisfying each constraint, in the given order. For example, `cFunDef "mm"` requires visiting a function definition with the name `"mm"`, and `cFor id` requires visiting a for loop over an index with the name `id`. Targets may also include special modifiers For example, `tBefore` allows targeting the interstice before an instruction. As another example, `cStrict` controls the depth: `[cFunBody "mm1024"; cStrict; cFor ""]` targets all the for-loops that appear immediately within the body of the function `mm1024`, as opposed to being nested within other constructs. Targets may also be given as arguments to constraints, for

[2] A full report for our matrix multiplication script may be found at: https://files.inria.fr/optitrust/soap24/matmul.html

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

Listing 2. OptiTrust script for optimizing `mm1024`.

example, `cFor ~body:[cPlusEq ()] "k"` requires visiting a for loop over an index with the name `"k"`, whose body also contains a `+=` operation.

Transformations. The script from Listing 2 calls transformations from the OptiTrust library. We next describe the key steps of the script. The transformation `Function.inline_def` inlines the definition of `mm` into the `mm1024` function, which specializes the naive matrix multiplication algorithm to the sizes $m = n = p = 1024$. The transformations `Loop.tile`, `Loop.reorder_at` and `Loop.hoist_expr` apply loop transformations whose purpose is to: (1) improve data locality, and (2) exposing additional opportunities for parallelization. Observe in passing how we define a local `tile` function as a shorthand for a specific form of tiling operation, and how we invoke this function multiple times using OCaml's (`List.iter`) operation to iterate over a list. More generally, OptiTrust scripts may leverage any OCaml feature. The transformation `Loop.hoist_expr` is here used to introduce a new temporary matrix, named `pB`, for storing values of matrix `B` using a better layout. Note that such a transformation is out of reach of the TVM compiler; in the TVM case study, the input code is *not* the naive implementation of matrix multiplication from Listing 1, but instead a manually patched code where the intermediate array `pB` needs to be already explicitly exploited. The transformation `Matrix.stack_copy` is used to locally promote an array to the stack (by means of fast `memcpy` operations), in order to subsequently allow operating on that fresh aligned array using SIMD vector registers. The transformations `Omp.simd` and `Omp.parallel_for` introduce OpenMP pragmas for multi-threading and for vectorizing certain loops. The transformation `Loop.unroll` unrolls the 4 iterations of the loop over `k`; doing so helps the downstream C compiler (namely, `gcc`) to recognize opportunity for exploiting SIMD.

Combined Transformations. Our optimization script from Listing 2 consists of only 10 lines, invoking high-level transformations. Yet, internally, these high-level transformations trigger the application of numerous *basic* transformations. (These internal basic transformations may be visualized in the report generated by OptiTrust.[2]) In OptiTrust, a *basic transformation* is one that directly modifies the abstract syntax tree (AST) of the program, and is a transformation whose validity is checked by exploiting resource usage information. All other OptiTrust transformations are called *combined transformations*. Combined transformations are implemented as composition of basic transformations, and their correctness stems from the correctness of the underlying basic transformations.

An example combined transformation is `Loop.reorder_at` (Line 4 of Listing 2). This transformation takes as argument a specific instruction, as well as a description of the desired order for the loops that surround this instruction. 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 involves a total of 4 *loop swaps*, 6 *loop fissions*, and 2 *loop hoist* operations. In particular, the effect of the last hoist operation is to turn local variable named `sum` in Listing 1 into the 2D-array named `sum` in Listing 3.

Validity Checks. As mentioned above, OptiTrust leverages resource typing information to check that *basic* transformations preserve the semantics of the program. For example, consider the transformation `Loop.parallel_for`, which tags a loop with the OpenMP `parallel` directive. This transformation is correct if the contract for this loop does not contain any **`__smodifies`** clause (i.e. a clause describing a resource that the iterations of the loop need to process in sequence). As another example, for swapping two consecutive instructions in a sequence, OptiTrust checks that for any resource that both instructions require, this resource is needed only for a read usage in the two instructions.

Every transformation may need to exploit resource information. Hence, we need the code to typecheck in-between every two transformations.[3] A transformation, to ensure that its output code typechecks, may need to adapt loop fonctions, and possibly to insert *ghost instructions*. In Separation Logic terminology, a ghost instruction is a no-op from the perspective of the semantics, however it reorganizes the view on certain resources. For example, the *Loop.swap* transformation needs to insert, before the outer loop, a ghost operation for swapping two iterated conjunctions; more precisely, for turning the resource: $\star_{i\in 0..m} \star_{j\in 0..n} \left(\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}\right)$ into the resource $\star_{j\in 0..n} \star_{i\in 0..m} \left(\texttt{\&C[i][j]} \rightsquigarrow \texttt{Cell}\right)$. The swap transformation also needs to insert a symmetrical operation after the loops, to revert the view on the resources back to its original form.

The need for ghost instructions is totally standard in Separation Logic frameworks. The novelty here is for ghost instructions to be inserted by program transformations. Moreover, we need to devise program transformations to “move around” ghost instructions, or to “cancel out” pairs of two ghost operations reciprocal of one another. These additional program transformations are called indirectly by our high-level combined transformations, hence they are not visible in the user-level script.

Final Optimized Code. Listing 3 shows the optimized C code produced by our optimization script. By manual inspection, we checked that our code matches the same structural optimizations as visible in the LLVM IR code produced by TVM. Furthermore, by means of executing benchmarks[4], we checked that our code matches the performance of TVM’s code—that is, a 150× speedup over the naive code from Listing 1.

In summary, this first case study shows that OptiTrust, a general-purpose optimization framework, can be used to interactively develop a code competitive with that produced by a state-of-the-art specialized compiler. Unlike specialized compilers, OptiTrust takes as input standard C code, and provides for every transformation feedback in the form of a *diff* over C code. Moreover, our optimization script for that case study is not much longer than that of TVM. To the best of our knowledge, OptiTrust is the first framework to demonstrate the ability to reproduce a case study from a specialized compiler such as TVM inside a general-purpose optimization framework.

---

[3]Our current implementation executes the typechecker on the whole program after every transformations; yet, for obvious performance reason, we have started working on making the typechecker incremental.

[4]The benchmark was performed on a 4-core Intel i7-8665U CPU with AVX2 support, the type of architecture for which TVM’s case study had been optimized for. Besides, our median runtime was very slightly faster than the TVM median runtime. Moreover, our 90th percentile runtime was also slightly faster than the TVM median runtime. Note that the point of this benchmark is not to discuss the performance values in absolute terms, but simply to check that we have been able to reproduce all the optimizations performed by TVM.

```
void mm1024(float* C, float* A, float* B) { // matrix multiply for 1024x1024 matrices
  float* pB = (float*)malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++) {
    for (int bk = 0; bk < 256; bk++) {
      for (int k = 0; k < 4; k++) {
        for (int j = 0; j < 32; j++) {
          pB[32768 * bj + 128 * bk + 32 * k + j] =
            B[1024 * (4 * bk + k) + 32 * bj + j]; }}}}
  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* sum = (float*)malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          sum[32 * i + j] = 0.; }}
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++) { // this loop is for k = 0
            s[j] += A[1024 * (32 * bi + i) + 4 * bk + 0] *
                    pB[32768 * bj + 128 * bk + 32 * 0 + j]; }
          // [...] similar unrolling, not shown, for k = 1, 2, 3
          memcpy(&sum[32 * i], s, sizeof(float[32])); }}
      for (int i = 0; i < 32; i++) {
        for (int j = 0; j < 32; j++) {
          C[1024 * (32*bi + i) + 32*bj + j] = sum[32*i + j]; }}
  // [...] free instructions, not shown
}
```

Listing 3. Optimized C code produced by our OptiTrust script (shown in Listing 2) for the matrix multiplication function `mm1024` (shown in Listing 1). This optimized code has similar structure and achieves similar performance as the reference output of TVM.

2.2 Optimizing Box Blur

The aim of our box blur case study is to demonstrate more generality and expressiveness. We aim to produce similar code as a reference code from the handwritten OpenCV library.[5] Our generated code, which shares the same structure, may be found in Listing 6. OpenCV is a popular optimized library for computer vision that is backed by industry. Blurs are essential components of many image processing pipelines, where they are used to remove noise and smoothen images. With a box blur, each pixel in the output image is equal to the average of its neighboring pixels in the input image. The box blur is often separated into a vertical box blur (summing pixels from the same column) and an horizontal box blur (summing pixels from the same row) to improve complexity. The division step can be performed at the very end, or depending on the context may be skipped. For this case study, we focus on deriving optimized code for the horizontal box blur, which already consists of a hundred lines of code.[6] A central aspect of the manually optimized code from OpenCV is that it begins by testing 5 specific input values and, for each test considered, provides code specialized for that input value.

The interest of this case study is that it involves a combination of optimizations that is out of reach of specialized compilers. In particular, Halide does not support the introduction of a sliding window—and there is no plan to lift this limitation.[7] Either the programmer needs to manually refine

[5] https://github.com/opencv/opencv/blob/4.x/modules/imgproc/src/box_filter.simd.hpp#L65

[6] In the future, we could look into deriving the optimized code for the vertical box blur as well, but it would require significant work as it consists of a thousand lines of code and performs SIMD vectorization explicitly using diverse intrinsics.

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

```
void rowSum(const int kn, const T* S, ST* D, const int n, const int cn) {
  __requires("kn >= 0, n >= 1, cn >= 0");
  __reads("S ~> Matrix2(n+kn, cn)");
  __modifies("D ~> Matrix2(n, cn)");
  __ghost(swap_groups, "items := fun i, c -> &D[i][c] ~> Cell");
  for (int c = 0; c < cn; c++) { // foreach channel
    __xmodifies("for i in 0..n -> &D[i][c] ~> Cell");
    for (int i = 0; i < n; i++) { // for each pixel
      __xmodifies("&D[i][c] ~> Cell");
      __ghost(assume, "is_subrange(i..i + kn, 0..n + kn)");
      D[i][c] = reduce_spe1(i, i+kn, S, n+kn, cn, c);
    }
  }
  __ghost(swap_groups_rev, "items := fun i, c -> &D[i][c] ~> Cell");
}
```

Listing 4. Unoptimized implementation for horizontal box blur. The function `rowSum` applies a box blur of size `kn` over a row `S` of size `n + kn` with `cn` channels, and stores the result in `D`.

the code to introduce the sliding window before using Halide; or needs to exploit transformation tools specialized for applying sliding window optimizations [**??**].

Annotated Code. The OpenCV code base does not include an unoptimized C implementation of horizontal box blur. Thus, we had to write one, to use as input to an OptiTrust transformation script. It felt natural to exploit a *reduce* operation, which is a standard high-level construct for high-performance programming. Because we do not yet support first-class functions, we have considered for the moment a version of *reduce* specialized for summing up values.[8] Our unoptimized implementation appears in Listing 4.

As before, we annotate the code with OptiTrust resource contracts. Each iteration of the loops with index `c` and `i` modifies a separate group of cells from the output `D`, as explicited by the `__xmodifies` contract clauses. Our typing algorithm being simple by design, we also need to use *ghost instructions* for the first time. A `__ghost(f, "args");` instruction is an instruction that has no impact on program execution, but instead impacts type-checking by calling a ghost function `f`. Here we call the ghost function `swap_groups` (defined as a regular C function in the OptiTrust library) to explicitly change the view on the memory of `D` from $\star_{i\in 0..n} \star_{c\in 0..cn}$ `&D[i][c] ~> Cell` to $\star_{c\in 0..cn} \star_{i\in 0..n}$ `&D[i][c] ~> Cell`. The contract of the `rowSum` function also includes a precondition on pure resources, described by the `__requires` clause. Contrarily to linear resources that describe ownership of part of the memory, pure resources describe immutable facts and include propositions such as `kn >= 0`.

Transformation Script. We have devised the OptiTrust script to transform the naive code from Listing 4 into a code featuring the exact same optimizations as the OpenCV implementation. Our script, shown in Listing 5, begins by introducing tests for the specific input values, duplicating the naive code in each branch. Concretely, this *multi-versioning* in introduced by means of `Specialize.variable_multi`. For `kn`, which corresponds to the span of the blur, the values 3 and 5 are commonly used by client of the library, For `cn`, which encodes the number of *channels*, the common values are 1 (grayscale), 3 (RBG), and 4 channels (RGBA). We thus need to optimize 5 specialized versions, plus the *generic* version of the code. Several specialized versions share common optimization

[8]Technically, `reduce_spe1` is a special case where the reduction operator is + over values of type `ST` (`uint16_t`), and where input values are read within the first dimension of a 2D matrix with elements of type `T` (`uint8_t`). In the future, we plan to support a generic `reduce` using C++ templates, and replace the call `reduce_spe1(a, b, M, n, m, j)` with `reduce<Add>(a, b, [&](int k){ (ST)M[MINDEX2(n, m, k, j)] })`.

```
bigstep "prepare for specialization";
let mark_then (var, _value) = sprintf "%s" var in
!! Specialize.variable_multi ~mark_then ~mark_else:"generic"
  ["kn", int 3; "kn", int 5; "cn", int 1; "cn", int 3; "cn", int 4]
  [cFunBody "rowSum"; cFor "c"];

bigstep "generic + cn";
!! Reduce.slide ~mark_alloc:"acc" [nbMulti; cMarks ["generic"; "cn"]; cArrayWrite "D"];
!! Reduce.elim [nbMulti; cMark "acc"; cFun "reduce_spe1"];
!! Variable.elim_reuse [nbMulti; cMark "acc"];
!! Reduce.elim ~inline:true [nbMulti; cMarks ["generic"; "cn"]; cFor "i"; cFun "reduce_spe1"];

bigstep "kn";
!! Reduce.elim ~inline:true [nbMulti; cMark "kn"; cFun "reduce_spe1"];
!! Loop.swap [nbMulti; cMark "kn"; cFor "c"];
!! Loop.collapse [nbMulti; cMark "kn"; cFor "i"];

bigstep "cn";
!! Loop.unroll [nbMulti; cMark "cn"; cFor "c"];
!! foreach_target [nbMulti; cMark "cn"] (fun c ->
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "i" ~body:[cArrayWrite "D"]];
  Instr.gather_targets [c; cStrict; cArrayWrite "D"];
  Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor ~stop:[cVar "kn"] "i"];
  Instr.gather_targets [c; cFor "i"; cArrayWrite "D"];
);
```

Listing 5. OptiTrust optimization script for reproducing horizontal blox blur from OpenCV.

strategies—even though the code produced differ. We are able to factorize our OptiTrust script accordingly.

For both the generic code and the channel-specific code, we call `Reduce.slide` to apply a sliding window accumulator optimization that, instead of recomputing sums of neighbors on every iteration over `i`, reuses the previous sum to compute the next one. This algorithmic optimization lowers the complexity of the computation. Then, we expand some of the reduce patterns into for loops using `Reduce.elim` and others into inline expressions using `Reduce.elim ~inline:true`. We eliminate some unnecessary variables using `Variable.elim_reuse`. Note that the script leverages *marks*, for example `Reduce.slide ~mark_alloc:"acc"` will leave a mark on the variable allocations that it generates, allowing later transformations to target these allocations using `cMark "acc"`.

Additionally for the channel-specific code, we unroll the loop over `c` before interleaving the computations from different channels using a combination of fusing loops and moving instructions. Processing all channels at the same time improves memory locality.

For the size-specific code, we expand the reduction pattern into an inlined expression, and swap the loops over `i` and `c` before collapsing them into a single one. The reduced complexity of the sliding window accumulator optimization is not beneficial for such small kernel sizes.

Final Optimized Code. Listing 6 shows an excerpt of the ~100 the optimized C code produced by our script. One difference between the OpenCV code and ours is that OpenCV traverse certain arrays by incrementing pointers, whereas we use explicit array indexing everywhere. Another minor difference is that the OpenCV code leverages templates to be generic in the size of the integers being manipulated in the input and output arrays, whereas for the moment we optimize code that refers to fixed (yet unspecified) integer types. Up to these minor differences, the case study demonstrates the ability of OptiTrust to reproduce a carefully optimized, hand-written implementation from a state-of-the-art image processing library.

2.3 Optimizing a Particle Simulation

WORK IN PROGRESS

3 THE OPTITRUST FRAMEWORK

3.1 Evaluation of OptiTrust

When considering the aforementioned criteria and tools, OptiTrust achieves a unique combination of features.

Generality. OptiTrust is generally applicable to optimizing C code. The code must parse using Clang, the parser of LLVM. The fragments of code that the user wishes to alter must moreover type-check in our resource type system. At the time of writing, we support only core features of the C language: sequences, loops, conditionals, functions, local and global variables, arrays, and structs. There is, however, no inherent limitation: OptiTrust could presumably be extended to support nearly all the C language (we do not plan to handle general goto's). Our resource type system currently only allows describing simple *shapes* of data structures, and does not yet allow specifying the stored values. That said, we have been planing to extend our implementation to a full-featured Separation Logic similar to RefinedC [Sammler et al. 2021]. In summary, OptiTrust in its current form does not yet demonstrate full generality, however it has been designed towards that goal.

Expressiveness. The combination of three ingredients allows OptiTrust's users to generate their desired optimized code: (1) the use of a transformation script for describing a sequence of transformations; (2) the use of a *target* mechanism, allowing to precisely pinpoint where transformations should be applied; (3) the availability of a catalog of general-purpose transformations, whose composition enables altering the code with a lot of flexibility.

Let us summarize the transformations currently supported in OptiTrust. For instruction-level transformations, we support: function inlining, constant propagation, instruction reordering, switching between stack and heap allocation, and basic arithmetic simplifications. For control-flow transformations, we support: loop interchange, loop tiling, loop fission, loop fusion, loop-invariant code motion, loop unrolling, loop deletion and loop splitting. For data layout transformations, we support: interchange of dimensions of an array, and array tiling. There are many more useful transformations for which we are working out sufficient correctness conditions.

Certain transformations may require nontrivial checks. For example, array tiling requires the tile size to divide the array size, and loop splitting requires arithmetic inequalities to hold. OptiTrust currently only validates simple conditions; in the future, more complex conditions could be handled using either SMT solvers or interactive theorem provers.

Control. Transformation scripts in OptiTrust empower the user with very fine-grained control over how the code should be transformed. A challenge is to allow for concise scripts. To that end, OptiTrust provides high-level *combined* transformations, effectively recipes for combining the *basic* transformations provided by OptiTrust. Section 2 presented the example of `Loop.reorder_at`, which attempts, using a combination of fission, hoist, and swap operations, to create a reordered loop nest around a specified instruction. Overall, the use of *combined* transformations allows for reasonably concise transformation scripts, with the user's intention being described at a relatively high level of abstraction. The user stays in control and can freely mix the use of concise abstractions and precise fine-tuning transformations.

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

Listing 6. Optimized C code for horizontal blox blur produced by executing our OptiTrust script from Listing 5 on our naive implementation from Listing 4. This code exploits essentially the same optimizations as in the reference OpenCV code.

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

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

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

Trustworthiness. Compilers are well-known to be incredibly hard to get 100% correct [Yang et al. 2011]. Like compilers, optimization tools are highly subject to bugs. OptiTrust mitigates the risks of producing incorrect code in two ways.

Firstly, we instrumented OptiTrust to generate *reports* when processing transformation scripts. A report takes the form of a standalone HTML page, which contains the diff for every transformation step (and sub-steps). Such a report can be thoroughly scrutinized by a third-party reviewer.

Secondly, we have organized the OptiTrust code base so as to isolate the implementation of the *basic* transformations, which consists of transformations that directly modify the AST. Only basic transformations need to be trusted. We have been careful to systematically minimize the complexity of the interface and of the implementation of our basic transformations. All other transformations—the *combined* transformations—are *not* part of the trusted computing base (TCB).

Performance. OptiTrust aims to support optimization scripts with hundreds of high-level transformations steps, for manipulating code involving thousands of lines of code. A first key ingredient that ensures scalability is the use of *purely functional* abstract syntax trees (ASTs) for representing programs. This representation enables OptiTrust to apply a transformation by only updating the path with affected AST nodes, while at the same time allowing to capture, at no runtime cost, a snapshot of all the intermediate ASTs. These ASTs are exposed in the full reports that the programmer can navigate interactively. A second key ingredient for scalability would be an *incremental typechecker*, providing the ability to re-typecheck only the subterms whose contract or code has changed. As of writing, we have not yet implemented such an incrementality feature for our typechecker. For our case studies, typechecking the whole code after every basic transformation was sufficient, inducing only a few seconds of overheads. We therefore leave to future work the addition of incrementality to our typechecker.

Tradeoff Performance vs TCB. The implementation of OptiTrust also involves a critical tradeoff between the *performance of transformations* and the *size of the trusted computing base*. Let us illustrate this tradeoff with two examples.

The first example consists of the transformation that fuses N adjacent loops. We have chosen to implement the fusion of 2 adjacent loops as a *basic* transformation, and to implement the fusion of N loops as a *combined* transformation that iterates calls to the basic fusion transformation. In doing so, we have maximized the simplicity of the critical code, which involves checking the correctness criteria for loop fusion. However, our combined transformation is slightly less efficient than an implementation that would directly fuse N loops, because it constructs intermediate ASTs that serve no purpose other than helping to validate the correctness criteria.[9]

The second example consists of the transformation that swaps two groups of instructions that appear consecutively within a sequence of instructions. We have chosen to implement, directly as a *basic* transformation, the swapping of a group of N instructions with another group of M instructions. An alternative would have been to support as *basic* transformation only the swapping of two individual instructions, and to view the swapping of two groups of instructions as a *combined* transformation. Our motivation for supporting groups of instructions directly is based on two observations. The first observation is that implementing the correctness criteria for handling groups of instructions requires just one more line of code than implementing the correctness criteria for handling individual instructions. Indeed, our code base already includes a function for computing the resource usage of a group of consecutive instructions. The second observation is that encoding the swapping two groups of size N and M in terms of the swapping of instructions would involve $O(NM)$ operations, whereas a direct implementation only requires $O(N + M)$ operations.

Beyond those two specific examples, our general guideline is to go for the simplest possible basic transformation, except for the cases where it would seriously compromise the scalability of OptiTrust.

3.2 OptiTrust's Internal AST

In OptiTrust, input C programs are encoded into an imperative λ-calculus. All code transformations are performed on that imperative λ-calculus. Then, programs are decoded back into C syntax. Before presenting the encoding in Section 3.4, let us first describe OptiTrust's internal λ-calculus..

Fig. 1 gives the grammar of OptiTrust's AST. In this language, variables are bound by let-bindings and function definitions, and they are always immutable. A benefit is that variables may be substituted with values without concern about occurrences as left- or right-values.

OptiTrust variables may refer to the name of any of the builtin functions. OptiTrust provides functions for allocating memory space without intializing it (alloc), for reading (get), for writing (set) a cell, and for freeing allocated space (free). Moreover, OptiTrust features two additional operations to allocate memory cells for which the corresponding free operation is implicitly performed at the end of the surrounding sequence. The operation $\mathbf{new}(t)$ allocates a memory cell initialized with a specific contents t. The operation $\mathbf{new}(\perp)$ allocates an uninitialized memory cell, that is, a cell in which read operations have undefined behavior. These two operations are meant to occur as part of a let-binding, e.g. $\mathbf{let}\ x = \mathbf{new}(3)$. Besides, the function get_incr, get_decr, get_decr and decr_get are used to encode `u++`, `u--`, `++u` and `--u`, respectively.

[9] If one day additional performance is required, nothing prevents a developer from implementing a correctness criteria for directly handling the fusion of N loops; this "shortcut" implementation could be used to obtain fast feedback, whereas the more trustworthy "basic" implementation could be used on the final production run, to double-check the result.

π	:=	$\mid$	$\mathbf{par} \mid \cdot$	"parallel" flag on for-loops
ρ	:=	$\mid$	$\mathbf{rec} \mid \cdot$	"recursive" flag on group of declarations
r	:=	$\mid$	$\mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$	range for simple loops
t	:=	$\mid$	$x \mid \mathbf{res}$	variables, and the special variable **res**
		$\mid$	$b \mid n$	boolean values, and number values
		$\mid$	$\{f_1 = t_1; \ldots; f_n = t_n\} \mid [t_1; \ldots; t_n]$	structure and arrays as values
		$\mid$	$\mathbf{let}\ x = t \mid \mathbf{decl}_\rho\ (t_1; \ldots; t_n)$	declaration, and group of declarations
		$\mid$	$(t_1; \ldots; t_n) \mid t_0(t_1, \ldots, t_n)$	sequence, and function call
		$\mid$	$t_1[t_2] \mid t_1.f$	projection from array/struct values
		$\mid$	$t_1 \boxplus t_2 \mid t_1 \boxdot f$	address computations
		$\mid$	$\mathbf{for}^\pi (i \in r)\, t_1$	simple for-loops, possibly parallel
		$\mid$	$\mathbf{while}\ t_1\ \mathbf{do}\ t_2$	while loops
		$\mid$	$\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$	conditional

Fig. 1. Grammar of OptiTrust's internal λ-calculus.

A special variable, named **res** is used to denote the result value of a function. As we will see, **return** t as final statement of a function is encoded as "**let res** $= t$", and if the return statement appears elsewhere in the function it is additionally followed by an "**exit** l" statement, where l corresponds to a label carried by the sequence associated with the function body. Moreover **res** appears in function contrats to specify the return value.[10]

The metavariable b denotes a boolean value (true or false). The metavariable n denotes an integer. To simplify the presentation, we do not distinguish here between all the possible types of numbers; Our implementation, however, accounts for a diversity of integer and floating point types. Record and array initializers are provided; we will explain further on how their treatment differ between *const* and *non-const* values.

OptiTrust features a special operator sizeof(T), which behaves like a constant, with the only specificity that transformations might affect the type T. The C standard also supports the form sizeof(e), where e is an expression. To support this form, OptiTrust features another primitive function sizeof_expr(e, n), whose semantics is to return the value n. The idea is that sizeof(e) is translated to sizeof_expr$(e,$ sizeof$(T))$, which corresponds to an OptiTrust expression in which transformations on e or T may be applied.

In the OptiTrust AST, the sequence construct is systematically used for describing function bodies, loop bodies, and branches of conditionals—even if the sequence contains zero or a single instruction. The systematic use of sequences is commonly found in the AST representation of C compilers (e.g., Clang), but less common in traditional presentations of the λ-calculus. Our motivation for systematic use of sequences is that is eases the definition of program transformations, in particular for transformations that need to insert or move instructions.

The elements of a sequence consist of: let-bindings, function calls without a binding for the return value, control structures such as loops and conditionals. A C source file is also described as a sequence, which may moreover contain declarations of types, functions, and global variables.

The OptiTrust AST features 4 operations to manipulate structured data. The operation $a[i]$ reads the i-th cell of the array a, provided a denotes a constant value. If, however, a corresponds to a heap-allocated or a mutable stack-allocated array, then the memory address of i-th cell of the array

[10] The use of a dedicated name such as **res** is common practice in program verification tools, e.g. ESC/Java [Flanagan et al. 2002], or Why3 [Filliâtre 2003]. Besides, viewing a return as an assignement instruction appears for example in the Viper program verification tool [Müller et al. 2017].

a can be computed by the operation $t \boxplus i$. This operations corresponds to the C pointer arithmetic operation `t+i`. The contents of that cellmay be retrieved by evaluating $\mathrm{get}(t \boxplus i)$. Likewise, reading the field f of a constant record r is described by the operation $r.f$, whereas the memory address of the field f of a record r allocated in memory is described by the operation $r \boxdot f$. This operation would correspond to the C arithmetic operation `r + offset(typeof(r), f)`.

The construct $\mathbf{for}^{\pi}(i \in \mathbf{range}(t_{\mathrm{start}}, t_{\mathrm{stop}}, t_{\mathrm{step}}))\ t_{\mathrm{body}}$ describes a *simple-for-loop*. In such a loop, the loop range, which consists of the loop bounds and the per-iteration step are evaluted only once at the start. 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 variable i denotes the loop index. It is bound in the loop body as an immutable variable. Optionnally, the loop may be tagged with a *parallel* flag, asserting that the loop may be executed in parallel. This flag corresponds to the directive: `#pragma openmp parallel`.[11]

For sequential C for-loops that do fit the format of our simple-for-loops, we encode them into while-loops. We use an annotation to indicate that they should be printed back as C for-loops. We postpone support for do-while loops, which are seldom used.

3.3 AST Manipulation and Unique Identifiers

The OptiTrust AST corresponds to an immutable tree data structure. A program transformation reads an abstract syntax tree and produces a fresh tree, which may share subtrees with the original tree. This purely functional programming pattern avoids numerous bugs that may arise when modifying data structures in-place. Moreover, it enables us to efficiently store, thanks to sharing, the *trace* that consists of the snapshot of all intermediate ASTs produced by a transformation script.

We maintain the invariant that, within a given AST, every variable binder and every variable occurrence bears a unique identifier (an integer). These unique identifiers not only make variable comparison more efficient, they avoid difficulties that may arise when transformations lead to name clashes. The string representation is used only as a default name for variables when printing out code in text format. Two variables with distinct identifiers may have the same string representation `x`, if the shadowing convention is respected. If, however, our analysis detects that an inner occurrence of a variable named `x` refers to an outer binder on `x`, then it means that one binder needs to be renamed. Required renaming are performed by OptiTrust automatically.

To maintain the invariant of unique identifiers, we need to refresh identifiers whenever a transformation duplicates a subterm. In fact, we maintain an even stronger invariant: a same physical tree node must occur at most once in a given AST. Thus, whenever a transformation needs to duplicate a subterm, it invokes a tree copy function that not only allocates fresh nodes but also freshens the identifiers associated with binders and update the corresponding variable occurrences accordingly.

Maintaining unique occurrence of nodes in ASTs has an additional benefits. We can assign unique identifiers not only to binders, but to every node. Unique identifiers on nodes are helpul for building auxiliary data structures used when performing code analyses. For example, if we build the graph relating functions to their call sites, we may use these unique identifiers to identify the call sites.

The reader may worry about correctness issues in case the implementation of a transformation is missing a copy operation for a duplicated subterm. Such a miss would be immediately caught by a checking procedure that we have implemented, using a hashtable to verify at every step that every

[11] The restrictions imposed by OpenMP on the ranges of parallel for-loops essentially constraint them to fit the format $\mathbf{range}(t_{\mathrm{start}}, t_{\mathrm{stop}}, t_{\mathrm{step}})$, which we use for simple-for-loops. Besides, note that OptiTrust currently does not support the optional arguments on OpenMP's parallel directive, such as `private` variables: for the moment, thread-local variables must be represented using explicit thread-indexed arrays.

node occurs exactly once in the current AST. Therefore, there is no risk in practice of unintentional node sharing.

Overall, the result of these policies of identifiers for linear resources is that, as long as the identifier of a linear resource is unchanged, it is known that the contents of memory associated with that resource is unmodified.

3.4 Principle of a Reversible Translation from C into an Imperative Lambda-Calculus

As mentioned earlier, input C programs are encoded into OptiTrust's internal AST. Crucially, our encoding-decoding scheme is designed for round-trip stability: if a fragment of C code is encoded into our imperative λ-calculus, and if it is not altered by a code transformation, then it is decoded back into the original C code. Importantly, our translation does not depend on our resource typing system. It only assumes that the input code is valid C code. We currently support only a subset of C, as detailed in Section 3.5. As we argue there, the presence of unsupported features in a number of functions from a C source file does *not* prevent OptiTrust to handle the remaining functions.

In order to enable this *stable round-trip* property, our encoding phase leaves a few C-specific annotations in the λ-calculus AST that it produces. For example, we use an annotation to indicate whether an access should be printed as `(*x).f` or `x->f`. These annotations are exploited during the decoding phase. Comments in the source code are currently not preserved, however in the future we could attach them to terms using annotations. Besides, printing details such as spaces, tabulation, and line printing may not preserved with respect to the C code initially provided by the programmer. However, after the code has gone at least once through the round-trip, if the OptiTrust user iterates a number of transformations, the parts of the C code that are not altered by the transformations remains textually unmodified.

The interest of applying transformations not on the C syntax but on a simpler syntax is to allow for less error-prone implementation of transformations. In particular, eliminating local mutable variables and left-values dramatically simplifies the rules for variable substitution. 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, the Common Intermediate Language (CIL) serves as intermediate compilation language for the whole .NET ecosystem [Gough and Gough 2001]; Why3 [Filliâtre and Paskevich 2013] serves as as intermediate verificiation language for C, Java, and Ada programs. Viper [Müller et al. 2017] serves as as intermediate verification language for Java, Rust, Go, OpenCL, etc. We are not aware, however, of any framework that leverages a translation into intermediate language *and* provides a reciprocal translation back to the source language, with the stable round-trip property

As mentioned earlier, OptiTrust's encoding eliminates local mutable variables. A variable `x` is *pure* if there is no assignment operation on `x` and no occurence of `&x`. The C variables that are *pure* are simply mapped to *let-bindings* in OptiTrust's internal λ-calculus. For variables that are not pure, the OptiTrust AST uses heap-allocation. In particular, a variable is not pure if its address is taken in the C code, either explicitly via the address-of operator (`&x`), or implicitly via an assignment (`x = v`) or a compound assignment (`x += v` or `x++`).[12]

A complete technical presentation of our encoding scheme is beyond the scope of the present paper. Moreover, such a presentation will make more sense when OptiTrust covers a larger fragment of the C language. In what follows, we simply aim at given the intuition of the encoding, by means of a few basic examples presented in Fig. 2.

A specific difficulty is the treatment of pre/post-increment/decrement operators (`t++`, `++t`, `t--`, `--t`), whose semantics in the C language is nontrivial due to the possibility of undefined behaviors.

[12]In the expression `&(p->x)`, the variable `p` may be pure, because only a field of `p` is accessed.

`int x = 3;`	$\longleftrightarrow$	$\mathbf{let}_{\texttt{int}}\ x = 3;$	where `x` pure
`f(x);`	$\longleftrightarrow$	$f(x);$	with `void f(int)`
`int* a = malloc(sizeof(int));`	$\longleftrightarrow$	$\mathbf{let}_{(\texttt{int*})}\ a = \text{alloc}_{\texttt{int}}(1);$	where `a` pure
`*a = *a + 2`	$\longleftrightarrow$	$\text{set}(a,\ \text{get}(a) + 2);$	short for $\text{set}_{\texttt{int}}$ and $\text{get}_{\texttt{int}}$
`free(a)`	$\longleftrightarrow$	$\text{free}(a)$	
`int z;`	$\longleftrightarrow$	$\mathbf{let}_{(\texttt{int*})}\ z = \mathbf{new}_{\texttt{int}}(\perp);$	where `z` not pure
`z = 6;`	$\longleftrightarrow$	$\text{set}(z, 6);$	
`int v = z;`	$\longleftrightarrow$	$\mathbf{let}_{\texttt{int}}\ v = \text{get}(z);$	where `v` pure
`int y = 5;`	$\longleftrightarrow$	$\mathbf{let}_{(\texttt{int*})}\ y = \mathbf{new}_{\texttt{int}}(5);$	where `y` not pure
`f(y);`	$\longleftrightarrow$	$f(\text{get}(y));$	
`y = y + 2`	$\longleftrightarrow$	$\text{set}(y,\ \text{get}(y) + 2);$	
`y += 4`	$\longleftrightarrow$	$\text{set\_add}(y, 4)$	
`int* p = &y;`	$\longleftrightarrow$	$\mathbf{let}_{(\texttt{int*})}\ p = y;$	where `p` pure
`*p = *p + 2`	$\longleftrightarrow$	$\text{set}(p,\ \text{get}(p) + 2);$	
`int* q = &y;`	$\longleftrightarrow$	$\mathbf{let}_{(\texttt{int**})}\ q = \mathbf{new}_{(\texttt{int*})}(y);$	where `q` not pure
`q = &z`	$\longleftrightarrow$	$\text{set}(q, z);$	
`*q = *q + 2`	$\longleftrightarrow$	$\text{set}(\text{get}(q),\ \text{get}(\text{get}(q)) + 2);$	

Fig. 2. Example translations from C code into the OptiTrust's internal AST. A variable `x` is *pure* if there is no assignment operation on `x` and no occurence of `&x`.

Interestingly, under the assumption that the input program typechecks in our system, it is correct to view these operations as plain function calls, that is, terms of the form `get_incr(&i)` and `incr_get(&i)`. Indeed, our typing rules enforce the following property: if the order of evaluation of several subexpressions is not specified by the C standard, and if one of the subexpression performs a write effect on a resource, then the other subexpressions cannot read or write that same resource. Remark: in the particular case where a function contains pre/post increment/decrement operators yet does not typecheck in our type system, we need to treat the whole function body as "unsupported by our translation", as detailed in Section 3.5.

3.5 Unsupported C Features and their Handling by OptiTrust

Our translation covers a subset of the C language, as well as the `openmp parallel` pragma on for-loops. We have not considered programming features beyond the C standard, such as intrinsics, inline assembly, or preprocessor macros. We next list common features of the C language that, as of writing, our translation does not support, or supports only partially.

- Function pointers and variadic functions: we believe that there is no specific difficulty, however we have not yet implemented support for them.
- Union types: we have not yet tested programs using this feature.
- Switch-construct: we have not yet considered this feature; we plan to encode them using cascade of if-statements each testing a disjunction of equalities (i.e., allowing to factor several branches of the switch), but excluding code performing arbitrary fall-through.
- Compound literals: handling on-the-fly stack-allocation of data would require an extension to our current treatment of of stack-allocated variables.

- Variable length arrays: they introduce a (weak) form of dependent types, adding some complexity in typechecking and in transformations.
- Mutation of function arguments: the C standards supports reassigning function arguments, however it is generally considered bad practice. The OptiTrust translation checks that function arguments are never reassigned. We leave it to future work to apply an encoding that would introduce local mutable variables to avoid mutating function arguments.
- Abrupt termination: we do not yet handle the control-flow operators **break**, **continue**, and **return** unless at the end of the function body. Their treatment in Separation Logic is well-understood—they are handled, for example, in the VST program verification framework for C programs [Cao et al. 2018]. Yet, their support introduces a fair amount of additional complexity, both with respect to resource typing and with respect to loop transformations. Hence, we have decided to postpone their support.
- Goto's: we have no plan to support general goto's in OptiTrust.
- Low-level atomics: we leave these to future work.

Even though a C function may be supported by OptiTrust's translation scheme, this function may not typecheck in our resource type system, either the programmer has not bothered providing contracts, or because typechecking the function requires contracts with complex logical assertions, which OptiTrust does not yet support. As a result, the C functions that appear in a program fall in one the following three categories.

(1) *Functions are translated and typechecked.* For these definitions, all OptiTrust code transformation are available, and they are guaranteed to preserve the code semantics.
(2) *Functions that translated but not typechecked.* Certain semantic-preserving code transformations can be applied inside those definitions (e.g., creating a specialized version of function). More complex code transformations are either not supported (e.g., instruction delete), or can be applied by the programmer yet without any correctness guaranteed (e.g., loop swap).
(3) *Functions that not translated.* There are two cases.
    (a) *If OptiTrust is able to translate the prototype of the function*, then it produces an AST node for the function definition, and stores the body as plain text. In particular, the user may attach a contract to the function. The contract itself is not verified with respect to the function implementation, however the contract can be exploited for checking code that invokes this function.
    (b) *If OptiTrust is unable to translate the prototype* (e.g., due to variadic functions or variable length arrays), then the whole function definition is stored as plain text in the OptiTrust AST. If such a function, call it F, has an unsupported prototype, and another function G calls F, then the body G cannot be typechecked. However, the function G may be assigned an unverified contract. Thus, it is possible to typecheck other functions that invoke the function G.

In summary, the presence of unsupported features in a C file is not invasive with respect to the ability of OptiTrust to handle the rest of the code.

4 COMPUTING PROGRAM RESOURCES

Resource typing is key to obtaining information that is precise sufficiently for justifying numerous practical code transformations. This section explains the details of our type checking algorithms.

Our algorithm computes, for every statement and every subexpression, the set of resources that it consumes and produces. Moreover, for each resource being consumed, the algorithm records its *usage*, e.g., whether the resource is used for as read-only, as read-write, or as uninitialized resource, or whether it is permanently consumed. All this information is attached to the AST nodes.

Heap predicate	C syntax	Description
$p \rightsquigarrow \mathsf{Cell}_\tau$	`p ~> Cell`	permission to access the cell at address p of type τ
$p \rightsquigarrow \mathsf{Matrix1}_\tau(n)$	`p ~> Matrix1(n)`	permission on an array of length n
$p \rightsquigarrow \mathsf{Matrix2}_\tau(m, n)$	`p ~> Matrix2(m, n)`	permission on a $m \times n$ matrix
$\bigstar_{i \in r} H(i)$	`for i in r -> H(i)`	union of permissions $H(i)$ for each index i in r
αH	`_RO(α, H)`	read-only permission on H with fraction α
$\mathsf{Uninit}(H)$	`_Uninit(H)`	permission on H disallowing reads before write

Fig. 3. Common heap predicates

At a high-level, our typechecking algorithm is a top-down algorithm. This approach has the following benefits:

- Simplicity: we apply typing rules by following the syntax.
- Efficiency: typechecking is performed in a single pass over the AST.
- Explainability: if a type error is reported at a location, then this error depends only on the code and types of what comes before that location.

4.1 Typing Contexts

Our typechecker computes the set of *resources* available at every program point, and represents them in a *context*, written Γ. These resources consists of *pure resources* and *linear resources*.

Pure resources. The pure part of a typing context contains bindings of the form "$x : \tau$", where τ corresponds either to a C type (for which we used the meta-variable T), or to a *mathematical type*. A mathematical type can be thought of as Coq types (or, equivalently, as types of any other higher order logic). For example, mathematical types include $\mathbb{Z}$, finite and infinite sets. Mathematical types also include propositions: for example "$p : n > 0$" describes a proof p establishing the proposition "$n > 0$". In summary, the pure part of a typing context consists of an interleaving of a traditional program typing context (which binds program variables to C types) and of a Coq context (which binds ghost variables).

Linear resources. The linear part of a typing context contains bindings of the form "$y : H$", where y is a name (used in particular by usage maps) and H is a *heap predicate*. A heap predicate H describes ownership of part of the memory. Fig. 3 summarizes the most common heap predicates, which have already been discussed in Section 2, in particular, $p \rightsquigarrow \mathsf{Matrix1}_T(n)$ is syntactic sugar for $\bigstar_{i \in 0..n}\, p[i] \rightsquigarrow \mathsf{Cell}_T$. Likewise, $p \rightsquigarrow \mathsf{Matrix2}_T(n, m)$ denotes $\bigstar_{i \in 0..n} \bigstar_{j \in 0..m}\, p[i][j] \rightsquigarrow \mathsf{Cell}_T$.

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

Our typechecker carves out subfractions in such a way every time a read-only permission is required by the term at hand. This strategy ensures that we always keep around a fraction of the read-only permission, which may be useful for typing other nearby terms. Our algorithm eagerly merges back βH and $(\alpha - \beta)H$ into the original form αH. Note that these carve-out operations may be performed in cascade, and that merge-back operations can be performed in any order. To peform merge operations in the general case, we introduce the operation $\mathsf{CloseFracs}$, which will be used in our typing rules. This operation CloseFracs repeatedly applies the following rewrite rule:

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

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

Permissions on uninitialized cells. A standard separation logic ensures that the program never reads from an uninitialized memory cell. Traditionally, the specification of a read operation requires a permission of the form $p \rightsquigarrow v$ (or a fraction thereof), with the additional requirement that $v \neq \bot$, where $\bot$ is a special token denoting uninitialized content. We follow a slightly different, yet logically equivalent presentation. Our heap predicate $p \rightsquigarrow \mathsf{Cell}$ denotes not only the ownership of the cell at location p but also the information that its contents is previsouly initialized (i.e., is not $\bot$); we write $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell})$ to denote the ownership of this same resource but without the permission to read its contents.

Furtherfore, we generalize the predicate to the form $\mathsf{Uninit}(H)$ to describe uninitialized arrays and matrices. Concretely, for a matrix, $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Matrix2}(m, n))$ corresponds to $\bigstar_{i \in 0..n} \bigstar_{j \in 0..m} p[i][j] \rightsquigarrow \bot$. At this time, we do not attempt to provide a definition of $\mathsf{Uninit}(H)$ for arbitrary H, but only for those built as iterations over cells.

When our typechecker encounters a term that requires $\mathsf{Uninit}(H)$ in a context where the plain resource H is available, it weakens H into $\mathsf{Uninit}(H)$ on-the-fly.

Notations for contexts. Recall that a context consists of *pure resources* and *linear resources*. In this paper, we use the notation $\langle x_0 : \tau_0, \ldots, x_n : \tau_n \mid y_0 : H_0, \ldots, y_n : H_n \rangle$ to denote a resource set where x_i are pure resources of type τ_i, and y_i are linear resources with predicate H_i. The pure part is a telescope: this means that x_i may occur in any τ_j where $i < j$. The pure variables x_i also scope over the linear formulas H_j. The order of the linear resources y_j is essentially irrelevant. (It only affects the execution of the entailment algorithm on certain instances, for example if two resources describe a read-only permission over the same cell.)

Moreover, certain bindings "$x_i : \tau_i$" from the pure part of the contexts may be *alias definitions* of the form "$x_i : \tau_i := v_i$". Such an alias corresponds to a local definition in Coq; it may also be interpreted as a binding from x_i to the singleton type whose sole inhabitant is v_i. In practice, we simply write "$x_i := v_i$" because τ_i can be inferred from v_i. In presence of an alias of the form "$x_i : \tau_i := v_i$", our typechecker eagerly replaces x_i with v_i during internal unification operations.

Following the practice of proof assistants, resources names that are nowhere mentioned may be hidden. For example the context, $\langle p : \mathsf{ptr}, n : \mathsf{int}, n > 0 \mid p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}} \rangle$ contains two anonymous resources: $n > 0$ and $p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}}$.

As syntactic sugar, we define $[x_0 : \tau_0, \ldots, x_n : \tau_n]$ as $\langle x_0 : \tau_0, \ldots, x_n : \tau_n \mid \varnothing \rangle$.

Besides, we define $\alpha(y_0 : H_0, \ldots, y_n : H_n)$ as $(y_0 : \alpha H_0, \ldots, y_n : \alpha H_n)$ to distribute a fraction over a list of linear resources.

4.2 Triples and Usage Maps

Triples. Our typing judgement takes the form $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, capturing the fact that, in context Γ the term t is well typed and produces a context Γ' with a *usage map* Δ. We will come back later on to the details of this usage map. First, let us explain the bindings of the special variable **res**. If the term t yields a return value, then, by convention, this value is described in Γ' under the name **res**. If, moreover, this return value can be expressed by a simple logical expression, then **res** is bound as an alias in Γ'. This pattern will be illustrated for example in the typing rule for values.

In a triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, the contexts Γ and Γ' are *closed*, meaning that each of them only refers to variables that they bind. The postcondition Γ' repeats all the pure entries of the precondition Γ.

The pure bindings that appear in Γ' but not in Γ may correspond: (1) to the binding for **res**, which denotes the result value produced by t, and (2) to a number of ghost variables that correspond to existentially quantified variables of the postcondition of t. The linear bindings of Γ' may be arbitrarily modified compared with those in Γ, reflecting on the side-effects performed by t. Linear resources that are bound with the same name in Γ' as in Γ necessarily correspond to resources that have not been modified by t. As explained further, all of these effects performed by t are summarized in the usage map Δ.

As a shorthand, we omit Δ and write $\{\Gamma\}\ t\ \{\Gamma'\}$ when the explanations need not focus on the usage map. Internally, however, usage maps are systematically computed: when typechecking an AST node t, the value of Δ is attached to this AST node, together with Γ and Γ'.

Definition of usage maps. A *usage map* is an association map that binds resource names to *usage kinds*, which we detail below. For a *pure* resource name, there are 2 possible usage kinds: required and ensured. For a *linear* resource name, there are 5 possible usage kinds: full, uninit, splittedFrac, joinedFrac and produced. In a triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, the usage map Δ binds names of resources that can be bound in Γ or Γ', or possibly in both. Besides, Δ binds only names of resources that are effectively manipulated by t. (In separation logic terminology, we would say that 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 a name fresh from 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 H, this new occurence of H is assigned a fresh name, distinct from y.
- "y : uninit" is similar to "y : full" but moreover captures the information that t needs not read the original contents of the memory cells associated with the resource named y. In particular, if t performs a write operation in a cell y before any read operation on y, then the usage of y is uninit.
- "y : splittedFrac" can arise when Γ contains a splittable linear resource "$y : H$", for some predicate H. The usage "y : splittedFrac" means that t uses an unspecified subfraction of this resource. In such a situation, the name y is bound both in Γ and in Γ'. It may be the case, however, that the resource named y carries different fractions in Γ and Γ'.
- "y : joinedFrac" can arise when Γ contains a linear resource of the form "$y : (\alpha - \beta_1 - \ldots - \beta_n)H$". The usage "$y$: joinedFrac" means that: (1) the linear resource named y is not used by t, and (2) t produced a resource of the form $(\beta_i - \gamma_1 - \ldots - \gamma_m)H$, and (3) these two resources are merged and the result appears in Γ' under the name y. As explained when describing the CloseFracs operation in Section 4.1, the resulting resource is $y : (\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H$.
- "y : produced" means that the linear resource y has been produced by t. In this case, y is a name fresh from Γ, and is bound 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'.

The naming policy of linear resources inside contexts is directly driven by usage maps. If a linear resource is entirely consumed, its name disappears. However, each time there is a remaining subfraction in the context, it keeps the initial resource name. This allow to detect that if t_1 uses a resource y as read only and then t_2 use the resource to modify its contents, the usage map $t_1; t_2$ has a usage map containing y : full. The full details for computing usage maps in sequences will be detailed in section **??**.

Operators on 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.

$$\begin{array}{rcl} \Delta.\mathsf{dom} & = & \mathsf{dom}(\Delta) \\ \Delta.\mathsf{consumed} & = & \Delta.\mathsf{full} \cup \Delta.\mathsf{uninit} \\ \Delta.\mathsf{usedRO} & = & \Delta.\mathsf{splittedFrac} \cup \Delta.\mathsf{joinedFrac} \\ \Delta.\mathsf{notRO} & = & \mathsf{dom}(\Delta) \setminus \Delta.\mathsf{usedRO} \\ \Delta_1 \Cap \Delta_2 & = & \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) \end{array}$$

4.3 Operators on Contexts

In general, a context Γ takes the form $\langle E \mid F \rangle$.

We define the projections $\Gamma.\mathsf{pure} = E$ and $\Gamma.\mathsf{linear} = F$.

We define $\Gamma \Vdash X$, where X is a set of names (of pure or linear resources), as the context made by keeping from Γ only the resources with names that belong to X. Furthermore, we let $\Gamma \Vdash \Delta$ be a shorthand for $\Gamma \Vdash \mathsf{dom}(\Delta)$ where Δ is a usage map.

Substitutions, specialization and renaming in contexts. First, we let $\mathsf{Subst}\{\sigma\}(X)$ denote the substitution of the bindings σ, inside the entity X. Each binding in σ maps a variable name to a value (possibly another variable name). For example, $\mathsf{Subst}\{x := v\}([y : \mathsf{int}, P : y = x])$ evaluates to $[y : \mathsf{int}, P : y = v]$. As explained in the previous section, our use of variable identifiers means that we do not need to deal with shadowing. We therefore consider to be an error to evaluate $\mathsf{Subst}\{\sigma\}(X)$ in case a key of σ occurs as a binding name in X.

Second, we introduce the operation $\mathsf{Specialize}_{E_0}\{\sigma\}(\Gamma)$ to eliminate certain bindings from Γ, substituting the corresponding occurrences with specified values, and checking that these values have the correct type in a pure context E_0. This operation assumes $\mathsf{dom}(\sigma)$ to be included in set of keys of Γ.pure. Concretely, $\mathsf{Specialize}_{E_0}\{x := v\}(\langle E_1, x : \tau, E_2 \mid F \rangle)$ where τ is not bound in E_1 evaluates to $\langle E_1, \mathsf{Subst}\{x := v\}(E_2) \mid \mathsf{Subst}\{x := v\}(F) \rangle$ and checks that $E_0 \vdash v : \tau$. Sometimes, instantiating variables from σ also forces the instantiation of other pure variables in Γ because of typing constraints. For example, $\mathsf{Specialize}_{x:\mathsf{int}}\{x_A := x\}([A : \mathsf{Type}, x_A : A, x'_A : A, B : \mathsf{Type}, x_B : B])$ evaluates to $[x'_A : \mathsf{int}, B : \mathsf{Type}, x_B : B]$. More generally,

$$\begin{array}{rcl} \mathsf{Specialize}_{E_0}\{\sigma\}(\Gamma) & := & \mathsf{Specialize}'_{E_0}\{\sigma\}(\varnothing, \Gamma) \\ \mathsf{Specialize}'_{E_0}\{\sigma\}(E_1, \langle x : \tau, E_2 \mid F \rangle) & := & \begin{cases} \mathsf{Specialize}'_{E_0}\{\sigma'\}(& \sigma = \{x := v\} \uplus \sigma' \\ \quad \mathsf{Specialize}_{E_0}\{\sigma''\}([E_1]).\mathsf{pure}, & \text{when } \mathsf{dom}(\sigma'') \subset \mathsf{dom}(E_1) \\ \quad \mathsf{Subst}\{\sigma'', x := v\}(\langle E_2 \mid F \rangle)) & E_0 \vdash v : \mathsf{Subst}\{\sigma''\}(\tau) \\ \mathsf{Specialize}'_{E_0}\{\sigma\}(E_1, x : \tau, \langle E_2 \mid F \rangle) & \text{when } x \notin \mathsf{dom}(\sigma) \end{cases} \\ \mathsf{Specialize}'_{E_0}\{\varnothing\}(E_1, \langle E_2 \mid F \rangle) & := & \langle E_1, E_2 \mid F \rangle \end{array}$$

We write $\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ as a short form for $\mathsf{Specialize}_{\Gamma_0.\mathsf{pure}}\{\sigma\}(\Gamma)$.

Third, we define $\mathsf{Rename}\{\rho\}(\Gamma)$ to rename certain keys from Γ. Here, ρ denotes a map from certain variable names bound by Γ to distinct fresh variables. For example, $\mathsf{Rename}\{x := x'\}(\langle E_1, x : \tau, E_2 \mid F \rangle)$ evaluates to $\langle E_1, x' : \tau, \mathsf{Subst}\{x := x'\}(E_2) \mid \mathsf{Subst}\{x := x'\}(F) \rangle$. Rename can also be used to rename the linear resources: for example $\mathsf{Rename}\{y := y'\}(\langle E \mid F_1, y : H, F_2 \rangle)$ evaluates to $\langle E \mid F_1, y' : H, F_2 \rangle$.

Technically, as explained earlier, contexts include a third component storing *existential fractions*. The substitution, specialization, renaming and refreshing operators apply in this component as well.

Separating conjunction of contexts. We define $\Gamma_1 \star \Gamma_2$ as $\langle \Gamma_1.\mathsf{pure}, \Gamma_2.\mathsf{pure} \mid \Gamma_1.\mathsf{linear}, \Gamma_2.\mathsf{linear} \rangle$, where the comma indicates list concatenation.

We also define iterated conjunction, which is used in particular in the typing rule for for-loops. We define $\bigstar_{k \in r} \Gamma$ where k occurs in Γ. Essentially this formula builds the separating conjunction of the linear resources, and replaces the pure variables of Γ with variables denoting indexed families. For example, in first approximation, if x of type bool appears in Γ, then x of type int $\rightarrow$ bool appears in $\bigstar_{k \in r} \Gamma$. More generally, if x of type τ appears in Γ, then x of type $\forall k \in r, \tau$ appears in $\bigstar_{k \in r} \Gamma$. Formally, $\bigstar_{k \in r} \Gamma$ is defined as:

$$\bigstar_{k \in r} \langle x_0 : \tau_0, \ldots, x_n : \tau_n \mid y_0 : H_0, \ldots, y_n : H_n \rangle \quad := \quad \langle x_0 : \tau'_0, \ldots, x_n : \tau'_n \mid y_0 : H'_0, \ldots, y_n : H'_n \rangle$$

$$\text{where} \begin{cases} \tau'_i & := \quad \forall k \in r.\ \mathsf{Subst}\{x_j := x_j(k) \mid j < i\}(\tau_i) \\ H'_i & := \quad \bigstar_{k \in r} \mathsf{Subst}\{x_j := x_j(k)\}(H_i) \end{cases}$$

4.4 Contracts

Certain terms like functions, loops, and certains conditionals, carry a user-provided contract that guides the typechecker, providing information that would be hard or costly to infer.

Function contracts. A function definitions annotated with a contract γ takes the form $\mathbf{fun}(a_1, \ldots, a_n)_\gamma \mapsto t$. Here γ consists of two contexts, one for the pre-condition, one for the post-condition. Formally, we write it $\{\mathsf{pre} = \Gamma_{pre}\ ;\ \mathsf{post} = \Gamma_{post}\}$. The pre-condition Γ_{pre} may refer to the formal parameters a_i, as well as the surrounding context. The post-condition Γ_{post} may refer not only to the same variables as the pre-condition, but also the pure variables bound in the pre-condition.

Loop contracts. A for-loop annotated with a contract χ takes the form $\mathbf{for}\ (i \in r)_\chi\{t\}$. Here χ consists of a structured record that binds per-iteration resources γ, shared reads F, sequential invariants Γ, as well as a set of variables E that scope over those three entities. The resource set γ has the same type as a function contract. F should contain only splittable resources—in practice, only read-only resources. Γ corresponds to a standard loop invariant in sequential separation logic.

$$\begin{cases} \mathsf{vars} = E & \text{Pure variables, common between all loop contract fields} \\ \mathsf{excl} = \gamma & \text{Function contract for resources used exclusively at one iteration} \\ \mathsf{shrd} = \begin{cases} \mathsf{reads} = F & \text{Read only resources shared between iterations} \\ \mathsf{inv} = \Gamma & \text{Sequential invariant (may depend on the loop index)} \end{cases} \end{cases}$$

As we will see later in typing rule, the loop body is typechecked in a context that binds i of type int, an hypothesis of type $i \in r$, the variables of E, the resources $\gamma.\mathsf{pre}$, (subfractions of) the resources in F and Γ. The loop body needs to produce the resources $\gamma.\mathsf{post}$, it needs to give back the resources from F that it recieved, and produce the resources $\mathsf{Subst}\{i := i + 1\}(\Gamma)$. The latter corresponds to the invariant at the begining of the next iteration.

A loop is parallelizable if and only if it admits a loop contract χ with an empty sequential invariant (that is $\chi.\mathsf{shrd.inv} = \varnothing$). We write $\mathsf{parallelizable}(\chi)$ in this case.

$$\textsc{Var}\ \frac{v : \tau \in E}{E \vdash v : \tau} \qquad \textsc{IntLit}\ \frac{}{E \vdash n : \mathsf{int}} \qquad \textsc{BoolLit}\ \frac{}{E \vdash b : \mathsf{bool}} \qquad \textsc{PureApp}\ \frac{E \vdash \cdot \boxplus \cdot : \tau_1 \to \tau_2 \to \tau \qquad E \vdash v_1 : \tau_1 \qquad E \vdash v_2 : \tau_2}{E \vdash v_1 \boxplus v_2 : \tau}$$

Fig. 4. Rules for typing pure values

4.5 Entailment and On-the-Fly Casts

We write $\Gamma \Rightarrow \Gamma'$ the *entailment* relation, which is standard concept in separation logic. Formally, $\Gamma \Rightarrow \Gamma'$ holds if there exist a map σ with a binding $x := v$ for each pure variable $x : \tau$ in Γ' where v has type τ in Γ such that there is a bijection between linear resources of Γ and linear resources of $\mathrm{Specialize}_\Gamma\{\sigma\}(\Gamma')$ that can be unified together. We denote $\Gamma \Leftrightarrow \Gamma'$ the property $(\Gamma \Rightarrow \Gamma') \wedge (\Gamma' \Rightarrow \Gamma)$.

In addition to entailment, we define a subtraction operation, which is also commonly used in sepatation logic frameworks. The subtraction operation $\Gamma \ominus \Gamma'$ fails (i.e. returns **None**) if a resource in Γ' cannot be found in Γ and returns a result of the form **Some** (σ, F) otherwise. There, σ is a map from pure variables of Γ' to instantiation values constructed in the context Γ, and F is the subset of linear resources from Γ that are left after intantiating all linear resources from Γ'. More formally, if **Some** $(\sigma, F) = \Gamma \ominus \Gamma'$, then F is one of the strongest linear contexts such that $\mathrm{dom}(\sigma) = \Gamma.\mathrm{pure}$ and that $\Gamma \Rightarrow \mathrm{Specialize}_\Gamma\{\sigma\}(\Gamma') \star F$.

The entailment algorithm that checks an entailment $\Gamma \Rightarrow \Gamma'$ is a particular case of a subtraction operation (with splitting of read-only resources disabled). At a high level, a subtraction operation $\Gamma \ominus \Gamma'$ is structured as follows. As usual in separation logic, the pure variables of Γ' are viewed as existential variables; we instantiate them with fresh unification variable. Also, each linear resource from Γ' is cancelled against a corresponding resource from Γ. If a resource of the form $\mathsf{Uninit}(H)$ appears in Γ' and the resource H appears in Γ, then our algorithm applies a weakening on-the-fly. If a resource of the form αH appears in Γ', and α is an unconstrained fraction variable, then our algorithm looks for a resource of the form βH in Γ, and splits this resource. A subsequent CloseFracs operation will merge back the two parts.

4.6 Typechecking of Terms

Typing rules. In this section we discuss the typing rules of our system. We choose here an algorithmic presentation, where the frame computation is explicit. Therefore, the choice of the rule to apply is entirely driven by the structure of the program. Algorithmically, when we check a triple $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$, Γ and t are inputs whereas Γ' and Δ are outputs. This section focusses on checking triples, we discuss the computation of usage maps (Δ) in section **??**.

Pure values. The simplest typing rule is the rule for pure values. Pure values consist of program variables and constant literals. Pure values can also be constructed from ghost variables and by the application of a pure operator, but these never appear directly in the program source code. When typing such expression, we simply remember an alias from **res** to the value itself.

Note that reading the value of a mutable program variable x is not a pure value, since it is encoded as the call $get(x)$.

$$\begin{array}{rlll} v ::= & x & & \text{Variable} \\ | & n \ | & b & \text{Integer or boolean literal} \\ | & v \boxplus v & & \text{Pure operation} \end{array}$$

Rule for let-bindings. A let-binding **let** $x = t$ stores the result of the expression t in a variable called x. Since inside the result of t is defined as a binding of the special variable **res**, we only have to rename this special variable to the intended name x. The postcondition of the let-binding itself

Val

$$\frac{\Gamma.\mathrm{pure} \vdash v : \tau}{\{\Gamma\}\ v\ \{\Gamma \star [\mathbf{res} : \tau := v]\}}$$

Let

$$\frac{\{\Gamma_0\}\ t\ \{\Gamma_1\} \qquad \Gamma_2 = \mathrm{Rename}\{\mathbf{res} := x\}(\Gamma_1)}{\{\Gamma_0\}\ \mathbf{let}\ x = t\ \{\Gamma_2\}}$$

Seq

$$\frac{\begin{array}{c}\forall i \in [1, n].\quad x_i \text{ fresh} \quad\wedge\quad \{\Gamma_{i-1}\}\ t_i\ \{\Gamma'_i\} \quad\wedge\quad \Gamma_i = \mathrm{Rename}\{\mathbf{res} := x_i\}(\Gamma'_i) \\ \Gamma_r = \begin{cases}\mathrm{Rename}\{x_i := \mathbf{res}\}(\Gamma_n) & \text{if } t_i \text{ is of the form ``}\mathbf{let\ res} = t'_i\text{''} \\ \Gamma_n & \text{otherwise}\end{cases}\end{array}}{\{\Gamma_0\}\ (t_1; \ldots; t_n)\ \{\Gamma_r\}}$$

Block

$$\frac{\{\Gamma_0\}\ (t_1; \ldots; t_n)\ \{\Gamma_r\} \qquad \mathbf{Some}\ (\varnothing, \Gamma_q) = \Gamma_r \ominus \mathrm{StackAllocCells}(t_1, \ldots, t_n)}{\{\Gamma_0\}\ \{t_1; \ldots; t_n\}\ \{\Gamma_q\}}$$

Fun

$$\frac{\begin{array}{c}\{[\Gamma_0.\mathrm{pure}] \star \gamma.\mathrm{pre}\}\ t\ \{\Gamma_1\} \qquad \Gamma_1 \Rrightarrow \gamma.\mathrm{post} \\ T_f = \ulcorner (T_1, \ldots, T_n) \to \mathit{typeof}(\mathbf{res}, \gamma.\mathrm{post}) \urcorner\end{array}}{\{\Gamma_0\}\ (\mathbf{fun}(a_1 : T_1, \ldots, a_n : T_n)_\gamma \mapsto t)\ \{\Gamma_0 \star [\mathbf{res} : T_f, \mathrm{Spec}(\mathbf{res}, \gamma)]\}}$$

Subexpr

$$\frac{\begin{array}{c}x_i \text{ fresh} \qquad \forall i \in [0, n].\quad \{\Gamma_i\}\ t_i^{\Delta_i}\ \{\Gamma'_i\} \quad\wedge\quad (\hat{E}_i, \hat{F}_i, \hat{F}'_i, \bar{F}_i) = \mathrm{Minimize}(\Gamma_i, \Gamma'_i, \Delta_i) \\ \forall i \in [0, n].\quad \Gamma_{i+1} = \langle \Gamma_i.\mathrm{pure}, \hat{E}_i \mid \bar{F}_i \rangle \quad\wedge\quad \hat{\Gamma}'_i = \langle \Gamma'_i.\mathrm{pure} \Vdash \Delta_i.\mathrm{ensured} \mid \hat{F}'_i \rangle \\ \Gamma_p = \mathrm{CloseFracs}(\Gamma_{n+1} \star \bigstar_{i \in [0,n]} \mathrm{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}'_i)) \qquad \{\Gamma_p\}\ E[x_0, \ldots, x_n]\ \{\Gamma_q\}\end{array}}{\{\Gamma_0\}\ E[t_0, \ldots, t_n]\ \{\Gamma_q\}}$$

App

$$\frac{\begin{array}{c}\Gamma_0 \ni \mathrm{Spec}(x_0, \gamma) \qquad [a_1, \ldots, a_n] = \mathrm{Args}(\gamma) \\ \mathbf{Some}\ (\sigma', \Gamma_f) = \Gamma_0 \ominus \mathrm{Specialize}_{\Gamma_0}\{\overline{a_i := x_i}^{i \in [1,n]}, \sigma\}(\gamma.\mathrm{pre}) \\ \mathrm{dom}(\rho) = \mathrm{dom}(\gamma.\mathrm{post}) \qquad \mathrm{im}(\rho) \cap \mathrm{dom}(\Gamma_0) = \varnothing \\ \Gamma_q = \mathrm{CloseFracs}(\Gamma_f \star \mathrm{Rename}\{\rho\}(\mathrm{Subst}\{\overline{a_i := x_i}^{i \in [1,n]}, \sigma, \sigma'\}(\gamma.\mathrm{post})))\end{array}}{\{\Gamma_0\}\ x_0(x_1, \ldots, x_n)_{\sigma,\rho}\ \{\Gamma_q\}}$$

For

$$\frac{\begin{array}{c}\Gamma_p = [\chi.\mathrm{vars}] \star (\bigstar_{i \in r}\ \chi.\mathrm{excl.pre}) \star \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{first}\}(\chi.\mathrm{shrd.inv}) \\ \mathbf{Some}\ (\sigma', \Gamma_f) = \Gamma_0 \ominus \mathrm{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma_p) \\ \Gamma'_p = [i : \mathrm{int}, i \in r] \star [\chi.\mathrm{vars}] \star \chi.\mathrm{excl.pre} \star \frac{1}{r.\mathit{len}} \chi.\mathrm{shrd.reads} \star \chi.\mathrm{shrd.inv} \\ \{\Gamma'_p\}\ t_b\ \{\Gamma'_q\} \qquad \Gamma'_q \Rrightarrow \chi.\mathrm{excl.post} \star \frac{1}{r.\mathit{len}} \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{next}(i)\}(\chi.\mathrm{shrd.inv}) \\ \Gamma_q = (\bigstar_{i \in r}\ \chi.\mathrm{excl.post}) \star \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{last}\}(\chi.\mathrm{shrd.inv}) \\ \Gamma_r = \mathrm{CloseFracs}(\Gamma_f \star \mathrm{Rename}\{\rho\}(\mathrm{Subst}\{\sigma, \sigma'\}(\Gamma_q))) \qquad (\pi = \mathrm{parallel}) \to \mathrm{parallelizable}(\chi)\end{array}}{\{\Gamma_0\}\ \mathbf{for}^{\pi}(i \in r)_{\chi,\sigma,\rho}\ t_b\ \{\Gamma_r\}}$$

If

$$\frac{\begin{array}{c}\{\Gamma_0\}\ t_0\ \{\Gamma'_0\} \qquad \{\mathrm{Learn}\{\mathbf{res} = \mathrm{true}\}(\Gamma'_0)\}\ t_1\ \{\Gamma_1\} \qquad \{\mathrm{Learn}\{\mathbf{res} = \mathrm{false}\}(\Gamma'_0)\}\ t_2\ \{\Gamma_2\} \\ (\Gamma_3 \text{ synthetized by another algorithm}) \qquad \Gamma_1 \Rrightarrow \Gamma_3 \qquad \Gamma_2 \Rrightarrow \Gamma_3\end{array}}{\{\Gamma_0\}\ \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \{\Gamma_3\}}$$

Fig. 5. Rules of our typesystem

$$\begin{array}{rcl}
\{[b : \ulcorner T \urcorner]\} & \mathsf{ref}(b) & \{[\mathbf{res} : \mathsf{ptr}] \star \mathbf{res} \rightsquigarrow \mathsf{Cell}_T\} \\
\{\} & \mathsf{ref_uninit}() & \{[\mathbf{res} : \mathsf{ptr}] \star \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_T)\} \\
\{\} & \mathsf{alloc}() & \{[\mathbf{res} : \mathsf{ptr}] \star \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_T)\} \\
\{[a : \mathsf{ptr}] \star a \rightsquigarrow \mathsf{Cell}_T\} & \mathsf{get}(a) & \{[\mathbf{res} : \ulcorner T \urcorner] \star a \rightsquigarrow \mathsf{Cell}_T\} \\
\{[a : \mathsf{ptr}, b : \ulcorner T \urcorner] \star \mathsf{Uninit}(a \rightsquigarrow \mathsf{Cell}_T)\} & \mathsf{set}(a, b) & \{a \rightsquigarrow \mathsf{Cell}_T\} \\
\{[a : \mathsf{ptr}] \star \mathsf{Uninit}(a \rightsquigarrow \mathsf{Cell}_T)\} & \mathsf{free}(a) & \{\}
\end{array}$$

Fig. 6. Contracts of built-in functions. Recall that function contracts are expressed on functions applied to formal parameters (i.e. variable names). Recall that H can be coerced on-the-fly into $\mathsf{Uninit}(H)$.

does not mention **res** anymore, and this is normal since the let-binding itself does not have a return value. Seeing a let-binding as an instruction in a sequence is unusal in a functional setting, but our sequences containing let-bindings are isomorphic to let-in chains. Note that let-bindings do not manage scopes by themselves, as scopes are managed by the typing rule for sequence.

Sequence of instructions. In the C language, each sequence is also a scope block. Here we will treat scope blocks and sequences as two different typing rules. This presentation with two rules is practical since it allows us to manage multideclarations such as `int x = 3, y = x;` as a sequence without scope of two let bindings. This paragraph focusses on the rule Seq that typechecks a sequence without scope block. This rule Seq embeds the fact that instructions are executed one after each other by threading a context through the instructions. Since each instruction might have an ignored return value if it is not a let-binding, we replace it by a ghost value of the same type by renaming the return value placeholder **res** with a fresh variable name. If the sequence contains an instruction of the form "**let res** = t_i", this instruction defines the return value of the sequence. Therefore, at the end of the sequence, we need to restore the name of the result of t_i (temporarily set to be x_i) to **res**.

Scope blocks. The scope block part of a sequence is handled by the rule Block. 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. This is a no-op in practice since all the program variables are already in the context. This approach ensures that we never lose information that may be needed later in the resource computation. However, this policy of never forgetting any variable tends to blow up pure context size, and we should apply some context filtering in future work.

Contrary to the pure context, the linear context is affected by the Block rule. Exiting a scope block means that the stack allocated variables that go out of scope disappear. We reflect that in our typing rule by consuming their cells at the end of the sequence. The operator $\mathsf{StackAllocCells}(t_1, \ldots, t_n)$ returns the resource set of cells that were allocated on the stack by this sequence. Formally,

$$\begin{array}{rcl}
\mathsf{StackAllocCells}(t_1, ..., t_n) & := & \mathsf{StackAllocCell}(t_1) \star \cdots \star \mathsf{StackAllocCell}(t_n) \\
\mathsf{StackAllocCell}(t) & := & \begin{cases} p \rightsquigarrow \mathsf{Cell}_\tau & \text{if } t \text{ is of the form "}\mathbf{let}\ p = \mathsf{stackalloc}(\tau)\text{"} \\ \varnothing & \text{otherwise} \end{cases}
\end{array}$$

The definition of StackAllocCells can be extended if we want to allow stack allocation in the middle of expressions such as `f(&(struct point){0, 0})`.

Function abstraction. When typing a function abstraction, the typechecker leverages the user-provided function contract γ and checks that it is respected by the function body t. The body itself it typed in a context capturing all the pure resources from the outside context, adding the function arguments and the pure precondition of the contract. There is no implicit capture of the linear context, therefore the linear resources available for typing the body t only consist of the linear resources of the precondition γ.pre. After typing the body of the function, the type-checker verifies that the output context entails the postcondition γ.post. The function abstraction itself is a pure operation that simply adds a binding for **res** as a function of spec γ. The syntax $\{\gamma.\text{pre}\}\ \mathbf{res}(a_1, \ldots, a_n)\ \{\gamma.\text{post}\}$ defines a binding for **res** as a function with contract γ. In the rule we made explicit the fact that the function contracts stored in the typing context always include all the arguments, but in the user annotation the function arguments are implicitly bound.

Function calls are splitted in two typing rules. In C, function calls evaluates their arguments in an arbitrary order. In our typing rules, we chose to separate the unordered evaluation of function arguments in the rule Subexpr and the actual function call App performed right after.

The parallel subexpression rule. Subexpr evaluates arguments subexpressions in parallel ensuring there is no interference between them. In this rule $E[t_0, \ldots, t_n]$ is a multi evaluation context where all the t_i are in position of evaluation. For function calls, each t_i is one of the arguments that needs to be evaluated and is replaced by a simple variable x_i to enable using the App rule.

To be more precise the Subexpr rule is an algorithmic version of the equivalent more standard rule Subexpr' defined below. As written in Subexpr', in principle, to type in parallel multiple subexpressions, we need to find a way to split the linear resources available such that each subexpression can be typed with a separate set of resources. Then we can merge the postconditions of all subexpression with leftover resources that were not used by any subexpression, before typing the surrounding function call.

Subexpr'

$$\frac{\Gamma_0 \Rrightarrow \left(\bigstar_{i\in[0,n]}\hat{\Gamma}_i\right) \star \hat{\Gamma}_r \qquad \forall i \in [0,n].\ \{\hat{\Gamma}_i\}\ t_i^{\Delta_i}\ \{\hat{\Gamma}_i''\} \qquad \hat{\Gamma}_i' = \langle \hat{\Gamma}_i''.\text{pure} \Vdash \Delta_i.\text{ensured} \mid \hat{\Gamma}_i''.\text{linear}\rangle \\ \Gamma_c = \text{CloseFracs}(\hat{\Gamma}_r \star \bigstar_{i\in[0,n]} \text{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}_i')) \qquad \{\Gamma_c\}\ E[x_0, \ldots, x_n]\ \{\Gamma_p\}}{\{\Gamma_0\}\ E[t_0, \ldots, t_n]\ \{\Gamma_p\}}$$

In practice, we do not know in advance how to split the resources between subexpressions. Therefore, the algorithmic rule Subexpr leverages the usage maps Δ_i to decide how to split resources while typing the subexpressions.

To make this decision about splitting, we introduce the operation $\text{Minimize}(\Gamma, \Gamma', \Delta)$ on a precondition Γ, a postcondition Γ' and a usage map Δ. These three arguments must come from a valid typing judgement $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$. In this case, Minimize returns a quadruplet $(\hat{E}, \hat{F}, \hat{F}', \bar{F})$ such that $\{\langle\Gamma.\text{pure}, \hat{E} \mid \hat{F}\rangle\}\ t\ \{\langle\Gamma'.\text{pure}, \hat{E} \mid \hat{F}'\rangle\}$ holds, $\Gamma \Leftrightarrow \langle\Gamma.\text{pure}, \hat{E} \mid \hat{F} \star \bar{F}\rangle$ holds, $\Gamma' \Leftrightarrow \langle\Gamma'.\text{pure}, \hat{E} \mid \hat{F}' \star \bar{F}\rangle$ holds, and intuitively $\bar{F}$ is a "maximal" frame removed from both Γ and Γ' and unused by t. We can see the Minimize operator as a way to find the minimal footprint of a term. The concrete algorithm to compute Minimize will be discussed in section **??**.

In the Subexpr rule, the first subexpression t_0 gets typed in the full context Γ_0, however while typing it, we discover its usage map Δ_0. By applying the Minimize operator on the triple found for t_0, our typecheker learns that only linear resources from $\hat{F}_i$ are needed to typecheck t_0, leaving $\bar{F}_i$ for other subexpressions. Therefore, we can type the next subexpression in a context Γ_1 with $\bar{F}_0$ as its linear resources. Γ_1 also contains all the pure resources from Γ_0 because they can be freely duplicated, and the fresh fractions generated by the Minimize operation $\hat{E}_i$ since those appear in $\bar{F}_i$.

Then, iteratively, all subexpressions are typed in a shrinking context Γ_i, until they are all typed. This suffices to guarantee that all t_i can be run in parallel but does not immediately give the postcondition after this parallel execution. The postconditions Γ_i' found in the recursive typing step contain too much resources: intuitively, all the resources in $\hat{F}_i$ are present, and each Γ_i' contains its own copy of all the pure facts in Γ_0. The postcondition of the parallel execution is instead composed of the separated conjunction of the leftover resources Γ_{n+1} which implicitly contains all the pure facts from Γ_0 and from the $\hat{E}_i$ along with all the *minimized postconditions* $\hat{\Gamma}_i'$. These minimized postconditions only contain pure facts generated by t_i and the linear facts of Γ_i' actually used by t_i given by $\hat{F}_i'$.

Note that in the rule Subexpr, two subexpressions can still share a read only permission of the same resource H. This is possible because the first subexpression t_i will only keep a subfraction αH of the resource (for any positive α) in its minimized precondition $\hat{\Gamma}_i$ and leave $(1-\alpha)H$ in Γ_{i+1} as a resource available for subsequent subexpressions. The second subexpression t_j using H will then keep βH in $\hat{\Gamma}_j$ and leave $(1-\alpha-\beta)H$ in Γ_{j+1}.

Theorem 4.1 (Subexpr $\leftrightarrow$ Subexpr'). *the algorithmic rule Subexpr is equivalent to the rule Subexpr' that splits resources before parallel execution.*

Proof. Suppose that the typing rule Subexpr holds. To instantiate the Subexpr', we choose $\hat{\Gamma}_i = \langle \Gamma_i.\text{pure}, \hat{E}_i \mid \hat{F}_i \rangle$, and $\hat{\Gamma}_r = \Gamma_{n+1}$. By definition of the Minimize operator, it follows that: (1) $\{\hat{\Gamma}_i\}\ t_i^{\Delta_i}\ \{\hat{\Gamma}_i''\}$ holds, and (2) $\Gamma_i \Rightarrow \langle \Gamma_i.\text{pure}, \hat{E}_i \mid \hat{F}_i \star \bar{F}_i \rangle$ Then, by duplicating pure facts, we get the property (3) $\Gamma_i \Rightarrow \hat{\Gamma}_i \star \Gamma_{i+1}$. Finally, by iterating the entailment (3), we can conclude $\Gamma_0 \Rightarrow \left(\bigstar_{i\in[0,n]} \hat{\Gamma}_i\right) \star \hat{\Gamma}_r$. All the remaining premisces are the same for both rules. Therefore, the rule Subexpr' is applicable whenever Subexpr is.

Reciprocally, since the frame rule holds in our separation logic, Subexpr is applicable whenever Subexpr' holds. □

Typing function applications. The rule App for the function application is used after its arguments are evaluated. To use this rule, the typechecker searches in the context Γ_0 a specification γ for the function x_0. It finds the formal arguments names of γ (a_i) and specialize them with the effective arguments of the call. By definition of Specialize, this step checks that effective arguments are well typed. Then, it instantiates the precondition of the function γ.pre by finding a pure variable substitution σ', and consuming pure resources in Γ_0 thus creating the frame context Γ_f. To build the context after the call, the typechecker adds the instantiated post-condition to Γ_f and try to close fractions.

The user or the transformations may provide two additional annotations σ and ρ that influence this step.

σ is a partial instantiation context for pure variables in the contract. It can be seen additional optional ghost arguments. Each binding $x := v$, where x is a pure variable of the function precondition forces to instantiate x with v insead of searching a value for x by unification. For some functions, it may be mandatory to give at least some of the pure arguments in σ because they cannot be found by unification.

ρ is a renaming map for the arguments that come from the function postcondition γ.post. It must contain one binding for each resource in γ.post (pure and linear). Each one of the new names must be unique and fresh in Γ_0. In all our applications, this renaming map is always autogenerated but it could be useful to manually specify some of its bindings to force a name for ghost values generated by a function.

As we saw in earlier examples, annotated code also features calls to ghost function that transform the resources available without performing any computation. As far as the typechecker is concerned, these ghost calls can be seen as regular function calls without a return value. Ghost calls are considered to have zero program argument, all their arguments are given as ghost arguments in σ. They are typed using the same rule as any other function.

For-loops. The typing rule for simple for-loops is almost entirely driven by the loop contract annotation χ. Before entering the loop, we check that the precondition of the loop contract can be instantiated from the context Γ_0. From outside the loop, the loop contract precondition is composed of the pure variables χ.vars, an iterated separating disjunction over the range of the loop r of the resources in χ.excl.pre, the shared read only resources χ.shrd.reads, and the shared sequential invariant χ.shrd.inv at the first iteration of the loop. This gives a frame Γ_F and an instantiation map σ.

After the loop we combine the frame Γ_F with the loop contract postcondition. From outside the loop, the loop contract postcondition is composed of the iterated separated conjunction of the resources in χ.excl.post, the same shared read only resources as in the precondition χ.shrd.reads, and the shared sequential invariant χ.shrd.inv but this time at the last effective value of i in the loop. Note that this last effective iteration can be different from the end bound of the range. For instance, $\mathbf{range}(0, 3, 2).last = 4$ and $\mathbf{range}(0, -1, 1).last = 0$. Like with function contracts, we try to close the fractions after adding new resources in the context.

Independantly, the typechecker verifies that the body of the loop t_b can be typed in a context with the loop index i of type int, an hypothesis that restricts i to be in the range of the loop r, the variables from χ.vars, the resources of χ.excl.pre, a subfraction $\frac{1}{r.len}$ of every resource in χ.shrd.reads and the sequential invariant χ.shrd.inv. Because the context is a telescope and contains an hypothesis $i \in r$ that implies $r.len > 0$, the fraction $\frac{1}{r.len}$ is always well defined.

After evaluating the body we check that it fulfills the loop contract postcondition. It consists of the resources of χ.excl.post, the same fractions of χ.shrd.reads that were given as a precondition of the body, and the sequential invariant at the next iteration.

If the loop is declared to be executed in parallel, we also check that the contract is indeed parallelizable (i.e. that it contains nothing in the sequential invariant).

Similarly to function application, the parameter σ allows to control how the external precondition is instantiated and the parameter ρ describes how the resources from the external postcondition are renamed. Renaming of the resources from the post-condition is not strictly necessary for loop contracts since each loop occurs exactly once in the code, but we chose to keep it for symmetry with function calls.

Conditionals. When typing a conditional instruction, we first start by typing the condition expression yielding a context Γ_0'. Then in both branches t_1 and t_2, we remember that the result of the condition (i.e. the variable **res** in Γ_0') is true or false respectively, and use this as the typing context for t_1 and t_2. In practice, the **res** variable after evaluating t_0 will often be an alias, therefore a mere substitution would lose information. Instead we use a new operator $\text{Learn}\{\mathbf{res} = b\}(\Gamma)$ that (1) convert the alias $\mathbf{res} : \text{bool} := v$ into $\mathbf{res} : \text{bool}, \mathbf{res} = v$ if such alias exists in Γ, (2) specialize the variable **res** with b, (3) simplifies generated boolean equalities. This third simplification step is useful because a boolean value will often yields to properties of the form $\text{true} = (x_1 == x_2)$, or $\text{false} = (x_1 < x_2)$ which can be respectively lifted to the more directly usable $x_1 = x2$ and $x_1 \geq x_2$ and used while typing the branches.

After typing t_1 and t_2 we obtain two contexts Γ_1 and Γ_2 that need to be joined. In the rule we leave the choice of Γ_3 to an oracle. In practice, this oracle should try to directly use a postcondition of a contract higher in the AST, and try a reasonable default if such contract is unavailable. This reasonable default could be to put all the newly generated pure facts in a disjunction and keep the linear context of one of the two branches. No matter how the oracle resolves this Γ_3, the typing rule should enforce that both Γ_1 and Γ_2 entails Γ_3.

5 JUSTIFYING TRANSFORMATION CORRECTNESS

In this section, we explain how OptiTrust leverages resource typing information to check the correctness of the transformations requested by the programmer. Recall that we only need to check the correctness of *basic* transformations, because *combined* transformations are defined as composition of basic transformations. We will cover several transformations supported by OptiTrust, focusing on those that leverage the resource information in an interesting way. A few other transformations are discussed in the appendix. Like in the previous section, our formalism is based on OptiTrust's internal λ-calculus.

For every transformation, we present a generally applicable, *sufficient condition* for the transformation to be correct. There could be situations where our criterias fall short of recognizing a transformation to preserve the semantics—in most cases, this situation arises because our Separation Logic is currently limited to expressing shapes of data structures. Besides the correctness criteria, recall that transformations may update loop contracts and may insert ghost instructions, to ensure that the output code typechecks. We will explicitly describe those key aspects.

Certain transformations operate on groups of instructions. Thus, depending on the transformation, the meta-variable T may denote either a single instruction, or a group of consecutive instructions. The typing information associated with a single instruction T is written in our formalism in the form $\boxed{\Gamma_1\ T; \Delta\ \Gamma_2}$, where Γ_1 denotes the initial set of resource, Γ_2 denotes the final set of resources, and Δ denotes the *usage* of the term T, as formalized in the previous section. The resource usage Δ of a group is obtained by computing the *composition of their usage*, as defined in ??. In other words, if Δ^i denotes the usage of the term T^i, then:

$$\boxed{\Gamma_1\ T; \Delta\ \Gamma_2} \quad \equiv \quad \boxed{\Gamma_1\ T^1; \Delta^1 \ldots T^n; \Delta^n\ \Gamma_2} \quad \text{where } T \equiv T^1; ..; T^n \;\text{ and }\; \Delta \equiv \Delta^1; \ldots; \Delta^n$$

5.1 Transformations on Sequences of Instructions

Moving Instructions. The transformation `Instr.move` allows to move a group of instructions to a given destination within the same sequence. Doing so amounts to swapping a group of instructions T_1 with an adjacent group of instructions T_2. Thus, the move transformation applies to a program of the form $E[T_1; T_2]$, where E denotes a context.[13] The transformation is formalized in Figure 7. The variables Δ_1 and Δ_2 denote the usage associated with T_1 and T_2. The result of the transformation is $E[T_2; T_1]$. The correctness criteria is stated on the right-hand-side of the figure; it is explained next.

The expression Δ_1.notRO essentially denotes the resources that T_1 modifies (Technically, Δ_1.notRO denotes the resources that it does not *only* reads; keep in mind that T_1 may also consume or produce resources, and is not limited to modifying resources.) The empty intersection $\Delta_1.\mathrm{notRO} \Cap \Delta_2 = \varnothing$ captures the idea that if a resource is modified by T_1, then T_2 must not use it, otherwise swapping T_1 and T_2 might not be correct. (The resource intersection operator $\Cap$ was defined in section 4.2.) The second empty intersection captures the symmetrical property: if a resource is modified by T_2, then T_1 must not use it. When both conditions are met, the only resources that both T_1 and T_2 depend on are accessed in read-only mode. In such a situation, the groups of instructions T_1 and T_2 may be safely swapped, without impact on the result of their evaluation.

Deleting Instructions. The transformation `Instr.delete` allows deleting a group of instructions T from a sequence. It therefore maps a program $E'[T_0; T; T_2]$ towards a program $E'[T_0; T_2]$, for a

[13] We allow contexts to capture the surrounding elements in the sequence. Concretely, we write $E[T_1; T_2]$ to mean a program of the form $E'[\{T_0; T_1; T_2; T_3\}]$, where each T_i denotes a group of instructions, and where E' denotes a context whose hole contains a sequence of instructions, as denoted by the braces.

$$\boxed{E\begin{bmatrix}T_1;\Delta_1\\ T_2;\Delta_2\end{bmatrix}} \longmapsto \boxed{E\begin{bmatrix}T_2;\\ T_1;\end{bmatrix}} \qquad \text{correct if: } \begin{cases}\Delta_1.\mathrm{notRO} \cap \Delta_2 = \varnothing\\ \Delta_2.\mathrm{notRO} \cap \Delta_1 = \varnothing\end{cases}$$

Fig. 7. Description of the basic transformation `Instr.move`.

$$\boxed{E\,[\Gamma\ T;\Delta]} \longmapsto \boxed{E\,[\varnothing]}$$

correct if $E\,[G;\,]$ typechecks, where G is a ghost instruction satisfying the triple $\{\Gamma_m\}\ G\ \{\mathrm{IntoUninit}(\Gamma_m)\}$, and where $\Gamma_m \equiv \Gamma \mathbin{|\!\cdot} \Delta.\mathrm{notRO}$.

Fig. 8. Description of the basic transformation `Instr.delete`.

$$\boxed{E\,[\varnothing]} \longmapsto \boxed{E\,[T;\,]}$$

correct if: (1) the program $E[T;\,]$ typechecks as $E[\Gamma\ T;\Delta]$ for some Γ and Δ, and (2) for those values of Γ and Δ, the program $E\,[G;\,]$ typechecks, where G is defined like in Figure 8.

Fig. 9. Description of the basic transformation `Instr.insert`.

context E'. Following the same convention as for instruction swap, we describe the transformation as mapping $E[T]$ to $E[\varnothing]$, for a context E.

The correctness criteria appears in Figure 8. Intuitively, the deletion operation preserves the semantics of the program if the contents of the resources modified by T is not observed by the rest of the program. To test this hypothesis, we build an auxiliary program, written $E[G]$, in which we replace the group of instructions T with a ghost instruction G that forces to cast the resources used by T into their corresponding "uninitialized form". In other word, if H is a resource modified by T, then the ghost operation[14] G consumes H and produces $\mathrm{Uninit}(H)$. The set of resources modified by T, written Γ_m in Figure 8, is computed by the filtering operation $\Gamma \mathbin{|\!\cdot} \Delta.\mathrm{notRO}$, which was defined in section 4.3.

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

Inserting Instructions. The transformation `Instr.insert` refines a program from $E[\varnothing]$ to $E[T]$, where T denotes the group of inserted instructions. The correctness criteria, described in Figure 9, is essentially the same as that for instruction deletion. Indeed, for $E[T]$ to admit the same semantics as $E[\varnothing]$, it suffices that $E[\varnothing]$ admits the same semantics as $E[T]$, i.e., to check that the deletion of T is correct.

Duplicating Instructions. In the C standard, an expression is said to be *reproducible* if its evaluation does not perform any visible (i.e. non-local) side-effect, and if evaluating this expression multiple time produces the same results.[15] If an instruction T is reproducible, then after a first instruction T, a second instruction T may be inserted or removed without affecting the semantics.

Thereafter, we omit the context surrounding the code snippet (previously written E).

$$\boxed{\begin{matrix}T;\\ \end{matrix}} \leftrightarrow \boxed{\begin{matrix}T;\\ T;\end{matrix}} \qquad \text{where } T \text{ is a reproducible instruction.}$$

Likewise, if an expression e is reproducible, then after the instruction **let** $x = e$, an instruction **let** $y = e$ may be inserted or removed.

[14] Technically, G is implemented not as a single ghost instruction, but as a group of ghost instructions, to facilitate further manipulation of the individual ghost instructions.

[15] We leave it to future work to capture the notion of reproducible using Separation Logic permissions. Doing so would involve, in particular, the introduction of permissions to invoke random-number generators, and of permissions to execute concurrent operations whose outcode depend on the scheduling of threads.

$$\boxed{\begin{array}{l}\textbf{let } x = e;\\ \end{array}} \leftrightarrow \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\end{array}} \quad \text{where } e \text{ is a reproducible expression.}$$

5.2 Transformations on Ghost Code

Updates to contracts and ghost instructions. The semantics of a program is fully determined by its actual C code: it does not depend in any way on the *ghost* code nor on the function and loop contracts. Therefore, the OptiTrust user may freely modify loop contracts, and may freely insert, delete, or modify ghost instructions. The requirement is to reach, after one or several updates, a set of annotations for which the typechecking of the same C code succeeds.

Introduction of equalities. From the semantics of certain operations, the OptiTrust user can deduce equalities between variables and/or pure values. These equalities take the form **assert**$(v = v')$. When facing such a construct, the typechecker augments the current typing context with this pure assertion. In fact, these assertions can be viewed as particular form of ghost instructions. As we explain further, such equalities may be exploited to perform rewriting operations on both C instructions and ghost instructions.

In what follows, we present 3 transformations that introduce ghost assertions. We will explain later how combined transformations leverage them to implement classical program optimizations.

Read after read. The first transformation asserts that two immediately successive reads in the same memory location yield the same result. In the version of Separation Logic that we consider, this assertion always holds; If we were to consider a more expressive concurrent separation logic featuring invariants, we would need to refine our criteria to check that the read-only permission used for the read operations is kept locally between the two read operations.

$$\boxed{\begin{array}{l}\textbf{let } x = \mathrm{get}(p);\\ \textbf{let } y = \mathrm{get}(p);\\ \end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = \mathrm{get}(p);\\ \textbf{let } y = \mathrm{get}(p);\\ \textbf{assert}(x = y);\end{array}}$$

Read after write. The second transformation asserts that reading immediately after a write yields the value that was written. Below, v denotes a pure value.

$$\boxed{\begin{array}{l}\mathrm{set}(p, v);\\ \textbf{let } x = \mathrm{get}(p);\\ \end{array}} \longmapsto \boxed{\begin{array}{l}\mathrm{set}(p, v);\\ \textbf{let } x = \mathrm{get}(p);\\ \textbf{assert}(x = v);\end{array}}$$

Reproducible expressions. The third transformation asserts that if we evaluate twice the same reproducible expression e (here again, in the sense of the C standard), then we get equal results.

$$\boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \textbf{assert}(x = y);\end{array}} \quad \begin{array}{l}\text{where } e \text{ is reproducible}\\ \text{(in the sense of the C standard).}\end{array}$$

Rewriting using equalities. The basic transformation `Rewrite.eq` allows to exploit an equality of the form **assert**$(v = v')$ to replace, anywhere in the rest of the sequence containing the assertion, an occurrence of v with an occurrence of v'.

$$\boxed{\begin{array}{l}\textbf{assert}(v = v');\\ E[v]\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{assert}(v = v');\\ E[v']\end{array}}$$

5.3 Transformations on Bindings

Inlining and binding for pure expressions. The basic tranformation `Variable.inline_pure` eliminates a binding of the form **let** $x = v$, where v is a pure expression. This transformation can always be applied without further check. The reciprocal transformation, `Variable.bind_pure`, introduces a binding for one or several occurrences of a pure expression v. Likewise, it requires no further check.

Inlining in the next instruction, for a single occurrence. The basic transformation `Variable.inline_one` eliminates a binding **let** $x = e$ in programs where x has exactly one occurrence, and this occurence is contained in the immediately succeeding instruction, and this instruction consists only of variables and function calls (in particular, it does not involve control-flow construct such as if-statements of for-loops.) The correctness of this transformation critically relies on the fact that our typing rules enforce the property that the order of evaluation of subexpressions is irrelevant (recall Figure 5).

$\boxed{\begin{array}{l}\textbf{let } x = e;\\ E_{instr}[x];\end{array}} \longmapsto \boxed{E_{instr}[e];}$ correct if E_{instr} has no control flow construct, and no occurence of x.

Inlining in the next instruction, for multiple occurrences. The combined transformation `Variable.inline_dup` expands a binding **let** $x = e$ at one of its occurrences, without removing the binding. Here again, we consider an occurrence in the immediately succeeding instruction, moreover assuming that this instruction does not contain control flow constructs. This combined transformation can be obtained by duplicating the let-binding, then applying `Variable.inline_one`.

$\boxed{\begin{array}{l}\textbf{let } x = e;\\ E_{instr}[x];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \textbf{assert}(x = y)\\ E_{instr}[x];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ E_{instr}[y];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ E_{instr}[e];\end{array}}$ correct if E_{instr} has no control flow construct, and e is reproducible.

Inlining in the scope of a sequence. The combined transformation `Variable.inline` eliminates a binding **let** $x = e$ in the general case. If e is a pure expression, then `Variable.inline_pure` is invoked. Otherwise, the transformation proceeds as follows. If there are no occurences of x, it invokes the transformation `Instr.delete`. If there is exactly one occurence of x, it attempts to move, using `Instr.swap`, the binding on x just in front of this binding, then invoke `Variable.inline_one`. If there are several occurrences of x in the sequence, then it moves the binding to the front of the first instruction that contains occurrences of x; then it applies the transformation `Variable.inline_dup`; then it repeats the process until reaching the last occurrence of x. We leave to future work the support, in a combined transformation, of more complex patterns where occurrences of a non-pure binding appear in depth below control flow constructs. We show below an example decomposition of `Variable.inline`.

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

Binding of one immediate sub-expression. The basic transformation `Variable.bind_one` is essentially the reciprocal of `Variable.inline_one`. Given a sub-expression e that appears inside an instruction of the form $E_{instr}[e]$ (without control-flow constructs), the `Variable.bind` transformation introduces a binding **let** $x = e$ and turns the instruction into $E_{instr}[x]$.

$$\boxed{E_{instr}[e];} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ E_{instr}[x];\end{array}} \quad \text{correct if } E_{instr} \text{ has no control flow construct.}$$

Folding for additional occurrences. The combined transformation `Variable.bind_dup` is essentially the reciprocal of `Variable.inline_dup`. In the scope of a binding **let** $x = e$, this transformation turns $E_{instr}[e]$ into $E_{instr}[x]$. We next detail how it can be decomposed as a sequence of basic transformations (`Variable.bind_one`, `Instr.swap`, `Ghost.reproducible`, `Rewrite.eq`, and `Instr.delete`).

$$\boxed{\begin{array}{l}\textbf{let } x = e;\\ T;\\ E_{instr}[e];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ T;\\ \textbf{let } y = e;\\ E_{instr}[y];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ T;\\ E_{instr}[y];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \textbf{assert}(x = y);\\ T;\\ E_{instr}[y];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ \textbf{let } y = e;\\ \textbf{assert}(x = y);\\ T;\\ E_{instr}[x];\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ T;\\ E_{instr}[x];\end{array}}$$

Common sub-expression elimination. The combined transformation `Variable.bind` exploits the transformations `Variable.bind_one` and `Variable.bind_dup` to introduce a binding to factorize the evaluation of common subexpressions. Let us illustrate this combined transformation for the particular case of two occurrences appearing in function calls in a sequence.

$$\boxed{\begin{array}{l}f(e);\\ T;\\ g(e);\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ f(x);\\ T;\\ g(e);\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } x = e;\\ f(x);\\ T;\\ g(x);\end{array}}$$

In the future, we plan to implement a high-level combined transformation for common sub-expression elimination that would automatically identify all the redundant expressions, then attempt to apply the transformation `Variable.bind` in order to introduce the relevant bindings. The OptiTrust user would thereby keep control of the scope of the program over which to search for common sub-expressions, and of where the freshly created bindings should be placed; however, the system would save the user the burden of targeting explicitly the sub-expressions to be factorized.

5.4 Transformations on Storage

Recall from Section 3 that a pure variable, e.g., `const int x = 3;` is represented in the OptiTrust as **let** $x = 3$, that a non-pure stack-allocated variable, e.g., `int x = 3;` is represented in the OptiTrust AST as **let** $x = \textbf{new}(3)$, and that an uninitialized variable `int x;` is represented as **let** $x = \textbf{new}(\perp)$. The permissions produced by **new**() are automatically reclaimed at the end of the scope (recall the rule *Seq* from Figure 5). Heap-allocated data is handled in OptiTrust like in C, via calls to the functions `malloc` and `free`.

The purpose of this section is to present transformations for introducing, eliminating, and converting between various forms of storage, with or without initialization. Our implementation also supports `calloc` for allocating zero-initialized memory cells, as well as `alloca` for allocating variable-sized arrays on the stack[16], but we omit them from the discussion for the interest of space. We also do not discuss straightforward transformations such as `Variable.rename`, whose purpose is simply to rename a variable, updating all its occurrences accordingly.

Separating declaration from initialization. For a stack-allocated variable, the basic transformation `Variable.init_detach` separates its declaration from its initialization. The basic transformation `Variable.init_attach` applies the reciprocal operation.

$$\boxed{\textbf{let } x = \textbf{new}(e);} \quad \leftrightarrow \quad \boxed{\textbf{let } x = \textbf{new}(\bot); \text{set}(x, e);}$$

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

$$\boxed{\{T_1;\ \textbf{let } x = \textbf{new}(\bot);\ T_2;\ T_2\}} \quad \leftrightarrow \quad \boxed{\{T_1;\ \textbf{let } x = \texttt{malloc}(n); T_2;\ \texttt{free}(x);\ T_3\}}$$

Our implementation supports the generalization of this conversion between stack and heap allocation to also handle arrays and N-dimensional matrices.

Converting a storage into a pure-binding. The basic transformation `Variable.to_const` applies to a stack-allocated storage named x, initialized at the moment of its declaration, and such as the only operation performed on x is reading the contents of x. The transformation replaces x with a pure binding. In the formal statement below, T denotes a group of instructions in the sequence before the binding on x, E denotes the group of instructions in the sequence after the binding on x, and the assumption is that all the occurrences of x are captured by the holes of the context E.

$$\boxed{\{T;\ \textbf{let } x = \textbf{new}(e);\ E[\text{get}(x), \ldots, \text{get}(x)]\}} \quad \leftrightarrow \quad \boxed{\{T;\ \textbf{let } x = e;\ E[x, \ldots, x]\}}$$

Removal of unused storage. If a stack-allocated storage is never used, it may be removed by means of the operation `Instr.delete`. Concretely, the instruction $\textbf{let } x = \textbf{new}(\bot)$ may be deleted if x has no occurrences, and the instruction $\textbf{let } x = \textbf{new}(e)$ may be deleted if moreover the effects performed by e are irrelevant to the rest of the program.

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

$$\boxed{\begin{array}{l}\textbf{let } x = \texttt{malloc}(e);\\ T; \text{// x not used}\\ \texttt{free}(x);\end{array}} \quad \longmapsto \quad \boxed{\begin{array}{l}\textbf{let } x = \texttt{malloc}(e);\\ \texttt{free}(x);\\ T;\end{array}} \quad \longmapsto \quad \boxed{T}$$

[16]Modern versions of the C-standard allow for variable size stack-allocation, however at this stage we prefer to avoid the introduction of dependent types.

Temporary alternative storage. The transformation `Variable.local_name` over a user-specified group of instructions, say T, for a user-specified storage, say x. Over this scope, a fresh storage, say y, is allocated. Just before executing T, the contents of x is copied into y. All instructions from T are updated to use y in place of x. Just after these instructions, the possibly-updated contents of y is copied into x. Depending on the context, the initial copy from x to y, or the final copy from y into x might be unnecessary.

The transformation applies to both a stack and heap allocated variable x, and the user may choose between stack and heap allocation for y. Moreover, our implementation supports the general case where x is not just a variable but an array of an N-dimensional matrix. In case where x is a matrix, y may corresponds to only a subset (i.e., a tile) of the matrix. The interest of the `local_name` transformation is to enable a computation kernel to operate on a local, cache-friendly copy of a 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 (as in Section 2.1), or to enable vectorization.

For the transformation `local_name`, the resource typing information is used not only for checking the correctness criteria, but also for guiding the generation of the output code. The transformation is described in Figure 10, where the group of instructions T is represented as $E[x, .., x]$, i.e., as a context with multiple occurrences of x. The context Γ_1 describes the resources available before T, and Γ_2 the resources available after T.

An essential aspect of the correctness criteria is to check that, during the execution of T, the full permission on x is *frozen* (i.e., made unavailable) in order to ensure that no operation may be performed on x via potential aliases of this pointer. A standard technique for enforcing such a *freeze* in Separation Logic is to introduce a magic wand operator, guarded by an abstract heap predicate, named H in the figure. This heap predicate serves the role of a *key* for unfreezing the resource x at the desired point—here the end of the scope on which y is used in place of x. The construct $\mathbf{ghost}(\Gamma \longrightarrow \Gamma')$ denotes a ghost instruction that consumes Γ and produces Γ'.

5.5 Loop Transformations

Recall that loop contracts are written χ (definition in section 4.4), and that arbitrary jumps are not yet allowed (no **`break`** or **`continue`**).

Tiling Loops. The `Loop.tile` transformation allows tiling (or strip-mining) a loop, transforming it into two nested loops, as shown in Figure 11. The transformation is generic over the functions RangeOuter, RangeInner and RecoverIndex, abstracting over different ways to compute ranges and indices. Table 2 shows examples of ranges and indices. The transformation is always correct, assuming that (RangeOuter, RangeInner, RecoverIndex) triples are correctly defined (they are part of the TCB): the same iterations should be performed in the same order.

The new inner loop contract is synthesized by substituting the new index value. The new outer loop contract is synthesized by substituting the appropriate index in the shrd resources, and by tiling the excl resources to match the new loop structure. Note that we use this shorthand notation in the contract definition:

$$\mathop{\bigstar}_{i \in r} \gamma = \{\text{pre} \equiv \mathop{\bigstar}_{i \in r} \gamma.\text{pre},\ \text{post} \equiv \mathop{\bigstar}_{i \in r} \gamma.\text{post}\}$$

Because tiled resources are now consumed and produced, ghost operations are also added to make sure that the new code will type in the same context as before. Indeed, the resources consumed by the initial loop are:

$$\mathop{\bigstar}_{i \in ri} \chi.\text{excl.pre} \star \text{Subst}\{i := ri.\text{start}\}(\chi.\text{shrd.inv}) \star \chi.\text{shrd.reads}$$

$$\boxed{\begin{array}{l} \hfill \Gamma_1 \\ E[x, .., x]; \\ \hfill \Gamma_2 \end{array}} \longmapsto \boxed{\begin{array}{l} \textbf{let } y = \textbf{new}(\bot); \\ T_1; \\ E[y, .., y]; \\ T_2; \end{array}}$$

where:

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

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

correct if: $\boxed{\begin{array}{l} \textbf{let } y = \textbf{new}(\bot); \\ \textbf{let\_ghost } (H : \text{resource}); \\ T_1; \\ \textbf{ghost}(R \longrightarrow H \star (H \mathrel{-\!\!\star} R)) \\ E[y, .., y]; \\ \textbf{ghost}(H \star (H \mathrel{-\!\!\star} R) \longrightarrow R) \\ T_2; \end{array}}$ typechecks successfully, where R is one of:

- $(x \rightsquigarrow \text{Cell})$
- $\alpha(x \rightsquigarrow \text{Cell})$
- $\text{Uninit}(x \rightsquigarrow \text{Cell})$

depending on how "$x \rightsquigarrow$ Cell" appears in Γ_1.

Fig. 10. Description of the basic transformation `Variable.local_name`. The construct $\textbf{ghost}(\Gamma \longrightarrow \Gamma')$ denotes a ghost instruction that consumes Γ and produces Γ'.

$$\boxed{\begin{array}{l} \textbf{for}_{\ \chi}\ i \in ri\ \{ \\ \quad T; \\ \} \end{array}} \longmapsto \boxed{\begin{array}{l} G_1; \\ \textbf{for}_{\ \chi_j}\ j \in rj\ \{ \\ \quad \textbf{for}_{\ \chi_k}\ k \in rk\ \{ \\ \qquad \text{Subst}\{i := \text{RecoverIndex}(j, k)\}(T); \\ \quad \} \\ \} \\ G_2; \end{array}} \quad \text{always correct.}$$

where:

$rj \equiv \text{RangeOuter}(ri) \qquad rk \equiv \text{RangeInner}(ri, j)$

G_1 and G_2 are groups of ghosts with contract triples:

$$\{\bigstar_{i \in ri}\ \chi.\text{excl.pre}\}\ G_1\ \{\bigstar_{j \in rj}\ \bigstar_{k \in rk}\ \chi_k.\text{excl.pre}\} \qquad \{\bigstar_{j \in rj}\ \bigstar_{k \in rk}\ \chi_k.\text{excl.post}\}\ G_2\ \{\bigstar_{i \in ri}\ \chi.\text{excl.post}\}$$

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

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

Fig. 11. The effect of `Loop.tile` and its correctness condition.

ri	RangeOuter(ri)	RangeInner(ri, j)	RecoverIndex(j, k)
$0..(n \times m)$	$0..n$	$0..m$	$j * m + k$
$0..n$ with $m \mid n$	$0..(n/m)$	$0..m$	$j * m + k$
$0..n$	$\mathbf{range}(0, n, m)$	$j..\min(j + m, n)$	k

Table 2. Example ranges and indices for the Loop.tile transformation.

$$\boxed{\begin{array}{l} \textbf{for}_{\ \chi_i}\ i \in r_i\ \{ \\ \quad \textbf{for}_{\ \chi_j}\ j \in r_j\ \{ \\ \qquad T; \\ \quad \} \\ \} \end{array}} \longmapsto \boxed{\begin{array}{l} G_1; \\ \textbf{for}_{\ \chi'_j}\ j \in r_j\ \{ \\ \quad \textbf{for}_{\ \chi'_i}\ i \in r_i\ \{ \\ \qquad T; \\ \quad \} \\ \} \\ G_2; \end{array}} \quad \begin{array}{l} \text{correct if:} \\ \chi_i.\text{shrd.inv} = \varnothing \lor \chi_j.\text{shrd.inv} = \varnothing \end{array}$$

Fig. 12. The effect of Loop.swap and its two correctness conditions. For contracts and ghosts see Figure 13.

While the resources consumed the produced loop nest are:

$$\bigstar_{j \in rj} \bigstar_{k \in rk} \chi_k.\text{excl.pre} \star \text{Subst}\{i := ri.\text{start}\}(\chi.\text{shrd.inv}) \star \chi.\text{shrd.reads}$$

Interchanging Loops. The Loop.swap transformation allows interchanging two loops, which changes the execution order of loop iterations as depicted on Figure 12. Intuitively, it is correct when the relevant loop iterations can be safely swapped. There exists a flexible correctness condition for this transformation[17], however it requires reasoning over logical resource formulas with quantified variables constrained by inequalities, a difficult task which we leave for future work. Instead, we focus on two simpler conditions that have many practical applications: the transformation is correct when either loop is parallelizable.

Interchanging Loops (Parallel Outer Loop). When the outer loop over i is parallelizable, pairs of iterations T and $\text{Subst}\{i := i', j := j'\}(T)$ commute when $i' \neq i$, as they share resources exclusively in read-only.[18] Sequential dependencies are restricted to the j dimension, and swapping i and j dimensions does not change the order of execution on the j dimension.

Figure 13 details the produced loop contracts and ghosts for this particular case. We seek to preserve the typing environment of the body T, and to preserve the typing environment of the context surrounding the loops. In order to achieve this, we first 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 (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. Then, we assign these resource partitions to the right place in the produced contracts. Observe that χ_j and χ'_j are extremely similar, except that necessary $\bigstar_i$ where added on the resources that are exclusive to i iterations. In particular, the generated loop over i is still parallelizable. Similarly, χ_i and χ'_i are alike, except that $\bigstar_j$ where removed on the resources that are exclusive to j iterations and that indices are substituted in the resources that come from the invariant of j iterations.

[17] T and $\text{Subst}\{i := i', j := j'\}(T)$ must share resources exclusively in read-only when $i' > i \land j > j'$ relative to range order

[18] this holds in particular when $i' > i$ and $j > j'$

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

G_1 and G_2 are groups of ghosts with contract triples:

$$\{\bigstar_i \bigstar_j PP_{i,j}\}\ G_1\ \{\bigstar_j \bigstar_i PP_{i,j}\} \qquad \{\bigstar_j \bigstar_i QQ_{i,j}\}\ G_2\ \{\bigstar_i \bigstar_j QQ_{i,j}\}$$

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

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

Fig. 13. The contracts and ghosts for Loop.swap (Figure 12) when $\chi_i.\mathsf{shrd.inv} = \varnothing$.

A concrete way to compute the partitions is by using the $\mathsf{PartialSub}(\Gamma_1, \Gamma_2)$ operator similar to the context subtraction operator from section 4.5. Instead of failing like $\Gamma_1 \ominus \Gamma_2$ when a resource in Γ_2 cannot be found in Γ_1, $\mathsf{PartialSub}(\Gamma_1, \Gamma_2)$ returns both the resources that were found, and the ones that could not be found, including pure resources. More formally, if $\sigma, F, \Gamma_1', \Gamma_2' = \mathsf{PartialSub}(\Gamma_1, \Gamma_2)$, then:

- F is one of the strongest linear contexts, and Γ_2' is one of the weakest linear contexts, such that $\Gamma_1 \star \mathsf{Specialize}_{\Gamma_1}\{\sigma\}(\Gamma_2') \Rightarrow \mathsf{Specialize}_{\Gamma_1}\{\sigma\}(\Gamma_2) \star F$
- equivalently, **Some** $(\sigma, F) = \Gamma_1 \ominus F_2 \wedge$ **Some** $(\varnothing, F_2) = \Gamma_2 \ominus \Gamma_2'$
- additionnaly, **Some** $(\varnothing, \Gamma_1') = \Gamma_1 \ominus F$

Using this operator, our example partition can be computed as follows:

$$\_, \_, R, R_i \equiv \mathsf{PartialSub}(\chi_i.\mathsf{shrd.reads}, \chi_j.\mathsf{shrd.reads})$$

In practice, we avoid this computation altogether in our implementation by instead leveraging information left by our type checker regarding realized contract instantations.

Ghost operations are added to make sure that the new code will type in the same context as before, swapping nested groups of exclusive resources to match the new loop nests.

Interchanging Loops (Parallel Inner Loop). When the inner loop over j is parallelizable, pairs of iterations T and $\mathsf{Subst}\{i := i', j := j'\}(T)$ commute when $i' \neq i$ and $j \neq j'$, as they share resources exclusively in read-only.[19] This holds less obviously than in the previous case, as one could worry that different iterations of i could shuffle resources between different iterations of j. Crucially here, our type system does not implicitly shuffle resources between different groups ($\bigstar$), requiring explicit use of ghost operations. Syntactically, there is only one instruction in the outer loop (the inner loop), and no ghosts operations. Therefore, sequential dependencies are restricted to the i dimension, and swapping i and j does not change the order of execution on the i dimension.

[19] again, this holds in particular when $i' > i$ and $j > j'$

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

with:

$$
\_, F, \_, \varnothing \equiv \mathsf{PartialSub}(\Gamma, \chi.\mathsf{shrd.inv}) \qquad R \equiv \mathsf{cleanup}(F)
$$

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

Fig. 14. The effect of `Loop.fission` and its correctness condition.

We elide the full definition of the transformation as it almost corresponds to reversing the arrow on the parallel outer loop case. The main detail breaking symmetry is that instead of matching ghosts G_1 and G_2, dual ghosts are generated. Modulo the ghosts G_1 and G_2, swapping with an outer parallel loop produces an inner parallel loop, and swapping again gives back the initial code. Finally, if both inner and outer loops are parallel, applying either criteria leads to the same outcome.

Fissioning Loops. The `Loop.fission` transformation breaks a loop into two loops over the same index range, as shown in Figure 14. The first loop performs the first instructions from the body (T_1), and the second loop performs the remaining instructions (T_2). This transformation changes the execution order of the instructions, and is correct in general when the relevant instructions can be safely swapped. As before, there exists a flexible correctness condition[20] that we leave for future work, for this paper we focus instead on a simple and practical condition.

In particular, loop fission is correct if the resources modified by T_1 and T_2 do not interfere across iterations for any i and i'. For χ.excl resources, there is no interference because each iteration is independent. For χ.shrd resources, we check for interference using Δ_1 and Δ_2: if T_1 modifies one resource from χ.shrd, then T_2 must not use this same resource; symmetrically, if T_2 modifies a resource, then T_1 must not use it. Note, however, that T_1 and T_2 may both read the same resource. In the figure, we use the following notation:

$$
\chi.\mathsf{shrd} \Vdash X \equiv \{\mathsf{inv} = \chi.\mathsf{shrd.inv} \Vdash X,\ \mathsf{reads} = \chi.\mathsf{shrd.reads} \Vdash X\}
$$

On top of checking for the correctness condition, the fission transformation must also synthesize new loop contracts. For shrd resources, we simply project the subsets of χ.shrd resources used by T_1 and T_2. For excl resources, we preserve the previous pre- and post-conditions, but need to synthesize a new middle-point (R) corresponding to the iteration-exclusive resources available between T_1 and T_2. R is computed by subtracting the shared resources (χ.shrd) from the resources available between T_1 and T_2 (Γ), and for technical scoping reasons, performing a final "cleanup".

Fusing Loops. The `Loop.fusion` transformation is the reverse of the `Loop.fission` transformation, merging two loops over the same index range into one as shown in Figure 15. It exhibits a similar

[20] T_1 and $\mathsf{Subst}\{i := i'\}(T_2)$ must share resources exclusively in read-only when $i' < i$ relative to range order

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

with:

$$
\_, \_, Q_1, P_2 \equiv \text{PartialSub}(\chi_1.\text{excl.post},\ \chi_2.\text{excl.pre})
\qquad
\chi \equiv \begin{cases}
\text{vars} \equiv \chi_1.\text{vars},\ \chi_2.\text{vars} \\
\text{shrd} \equiv \chi_1.\text{shrd} \star \chi_2.\text{shrd} \\
\text{excl.pre} \equiv \chi_1.\text{excl.pre} \star P_2 \\
\text{excl.post} \equiv \chi_2.\text{excl.post} \star Q_1
\end{cases}
$$

Fig. 15. The effect of `Loop.fusion` and its correctness condition.

$$
\boxed{\begin{array}{l}
\textbf{for}_{\ \chi}\ i \in r_i\ \{ \\
\quad \Gamma_1\ T_1; \Delta_1 \\
\quad \Gamma_2\ T_2; \Delta_2 \\
\}
\end{array}}
\longmapsto
\boxed{\begin{array}{l}
T_1; \\
\textbf{for}_{\ \chi'}\ i \in r_i\ \{ \\
\quad T_2; \\
\} \\
G_3;
\end{array}}
\quad
\begin{array}{l}
\text{correct if:} \\
\begin{cases}
i \text{ not free in } T_1 \text{ (constant code)} \\
T_1; T_1 \ \leftrightarrow\ T_1 : \\
\quad \begin{cases}
\_, \varnothing, \_, \varnothing = \text{PartialSub}(\Gamma_2 \Vdash \Delta_1.\text{produced},\ \Gamma_1 \Vdash \Delta_1.\text{uninit}) \\
\Delta_1.\text{full} = \varnothing
\end{cases} \\
\forall i.\ T_1; T_2; T_1 \ \leftrightarrow\ T_1; T_2 : \\
\quad \Delta_1 \Cap \Delta_2.\text{notRO} = \varnothing \\
r_i \text{ is nonempty, or the whole program typechecks with:} \\
\quad \{\Gamma_2 \Vdash \Delta_1.\text{produced}\}\ G_3\ \{\text{IntoUninit}(\Gamma_2 \Vdash \Delta_1.\text{produced})\}
\end{cases}
\end{array}
$$

$$
\text{with:}\quad
\begin{array}{l}
\text{in the final result: } G_3 \equiv \varnothing \\
\_, I', \_, \varnothing \equiv \text{PartialSub}(\Gamma_2,\ \chi.\text{excl.pre}) \\
\_, I, \_, \varnothing \equiv \text{PartialSub}(I',\ \chi.\text{shrd.reads})
\end{array}
\qquad
\chi' \equiv \begin{cases}
\text{vars} \equiv \chi.\text{vars} \\
\text{shrd.inv} \equiv I \\
\text{shrd.reads} \equiv \chi.\text{shrd.reads} \\
\text{excl} \equiv \chi.\text{excl}
\end{cases}
$$

Fig. 16. The effect of `Loop.move_out` and its correctness condition.

correctness condition. Regarding the synthesized contracts, the two sets of shrd resources are merged together to create the new set of shrd resources. For excl resources, we preserve the previous pre- and post- conditions, threading through any resources that might have been framed between the two loops: the Q_1 resources from γ_1.excl.post that are not consumed by γ_2.excl.pre are threaded through the new postcondition, and the P_2 resources from γ_2.excl.pre that are not produced by γ_1.excl.post are taken as input in the new precondition.

Hoisting an Instruction. The `Loop.move_out` transformation allows to hoist a group of instructions outside of a loop as shown in Figure 16, and is akin to loop invariant code motion.

This transformation is correct if the hoisted instructions T_1 are the same for every iteration i, can be safely de-duplicated and do not interfere with the other loop instructions T_2. Additionally, to avoid wrapping T_1 in a conditional, r_i must not be empty, or none of the contents from the resources modified by T_1 must be observed after the loop (similar condition as for `Instr.delete`).

Other Loop Transformations. Let us mention a few other transformations supported by OptiTrust.

- `Loop.collapse`: collapses two nested loops into one (reverse of tiling).
- `Loop.hoist`: hoist a multi-dimensional array allocation outside of a loop.
- `Loop.shift`: shifts the range of a loop, applying the reverse shift to the index occurences.
- `Loop.extend_range`: that extends the range of a loop by wrapping its body in a conditional.
- `Loop.unroll`: unrolls a loop whose range is statically known.
- `Loop.rename_index`: that renames a loop index.
- `Loop.parallel`: to assign a parallel flag to a loop.
- `Loop.to_memcpy`: replaces a loop with a *memcpy* operation.

6 RELATED WORK

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

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

Guidance in general-purpose compilers. To introduce human guidance in general-purpose compilers, a common approach is to insert *pragmas* into the code. For example, Scout [Krzikalla et al. 2011] is a pragma-based tool for guiding source-to-source transformations that introduce vector instructions. The main limitation of pragmas is that they are ill-suited for describing sequences of optimizations. Indeed, there is no easy way to attach a pragma to a line of code that is generated by a first optimization. Kruse and Finkel [Kruse and Finkel 2018] suggest the possibility to stack up pragmas, by providing labels as additional pragma arguments: a second pragma may refer to the labels introduced by the transformation described in a first pragma. This approach does not scale up well beyond a handful of successive transformations. OptiTrust, in contrast, supports chains of dozens of transformations.

Domain-specific compilers. Another possible approach to overcome the limitations of general-purpose compilers is to leverage *domain specific languages* (DSL), such as Halide [Ragan-Kelley et al. 2013], TVM [Chen et al. 2018], or Boast [Videau et al. 2018]. Specialized compilers can benefit from carefully tuned heuristics. Yet, even for programs expressed in a specific DSL, the optimization search space remains vast, hence programmer guidance is key to achieve good performance. In Halide and TVM, for example, the script that guides the compilation strategy is called a *schedule*.

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

In contrast to DSLs, OptiTrust sticks to a standard, general-purpose language. The correctness criteria for each transformation is expressed with respect to the semantics and our resource typing for the C language. As we have seen with the example of the *reduce* function in the OpenCV example, OptiTrust neverthess can manipulate domain-specific operations, and exploit transformations that are specific to these operations. At any point in the transformation script, an occurrence of a domain-specific operation may lowered into standard C code, thereby enabling further lower-level optimizations.

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

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

Source code manipulation frameworks. Frameworks that offer more expressiveness than rewrite rules generally give access to the abstract syntax tree (AST) of the source code. Traditional compilers employ an AST, but they are not designed for synthesizing pieces of AST at the source level. Moreover, traditional compilers operate on intermediate representations, and lose the structure of the original code. These two limitations of general-purpose compilers have motivated the development of frameworks that are specifically designed to support code transformations (and code analyses) at the level of C code. ROSE [Quinlan 2000; Quinlan and Liao 2011] and Cetus [Bae et al. 2013; Dave et al. 2009] are two such frameworks that provide facilities for manipulating C ASTs. Source-to-source transformation frameworks have also been employed to produce code targeting GPUs [Amini 2012; Konstantinidis 2013; Lebras 2019]. These frameworks implement generic optimization strategies, in a similar fashion as general-purpose compilers. In contrast, OptiTrust leverages transformation scripts to guide the optimization of a specific program. Moreover, the OptiTrust infrastructure supports resource typing, which provides much more precise information than the classic static code analyses implemented in the frameworks such as ROSE and Cetus.

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

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

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

Proof-transforming compilation. The notion of *Proof Carrying Code* [Necula 1998] refers to the idea that we should be able to produce compiled code that carries invariants establishing the same guarantees that are available on the high-level source code. These invariants may capture safety properties (e.g., no out-of-bound accesses), not necessarily full functional correctness. The related notion of *Proof-Transforming Compilation* refers to the process of taking of formally-verified program, and generating, in addition to the compiled code, a derivation (a.k.a. proof tree) that formally establishes the correctness of the compiled code.

The work by César Kunz [Barthe et al. 2009; Kunz 2009] shows how to realize proof-transforming compilation for standard compiler optimizations, applied at the level of the RTL intermediate language. The work on Alpinist [Sakar et al. 2022] demonstrates the feasibility, for a small number of GPU-oriented optimizations, of transforming GPU code while preserving logical invariants. Our work demonstrates the feasability, for a large number of general-purpose code optimizations, of transforming C code while preserving resource-based invariants. OptiTrust has been designed for supporting the manipulation of arbitrary Separation Logic invariants, and we look forward to experiment with this possibility in future work.

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

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

The effectiveness of Separation Logic has been demonstrated accross a broad range of applications, both for low-level and high-level code [Charguéraud 2020; O'Hearn 2019]. By building OptiTrust on Separation Logic assertions, we are confident that our framework has the potential to be generally applicable.

A ADDITIONAL TRANSFORMATIONS

A.1 Read-Last-Write

Read immediately preceeding write. The combined transformation `Expr.read_preceeding_write` expresses that, immediately after writing the result of a reproducible expression e in p, a read in p yields the same result as re-evaluating e. The decomposition, shown next, involves `Variable.bind_one`, `Ghost.read_after_write`, `Rewrite.eq`, and `Variable.inline` applied to the reproducible expression e. (In practice, we implement this transformation as a basic transformation for efficiency reasons, and taking into account the fact that its correctness criteria is trivial.)

$$\boxed{\begin{array}{l} \mathrm{set}(p, e); \\ E_{instr}[\mathrm{get}(p)]; \end{array}} \longmapsto \boxed{\begin{array}{l} \textbf{let } y = e; \\ \mathrm{set}(p, y); \\ \textbf{let } x = \mathrm{get}(p); \\ E_{instr}[x]; \end{array}} \longmapsto \boxed{\begin{array}{l} \textbf{let } y = e; \\ \mathrm{set}(p, y); \\ \textbf{let } x = \mathrm{get}(p); \\ \textbf{assert}(x = y); \\ E_{instr}[x]; \end{array}} \longmapsto \boxed{\begin{array}{l} \textbf{let } y = e; \\ \mathrm{set}(p, y); \\ E_{instr}[y]; \end{array}} \longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ E_{instr}[e]; \end{array}}$$

where e is reproducible.

Read last write. The combined transformation `Expr.read_last_write` generalizes the previous one to the case where a group of instructions, written T may appear between the write operation of e into x and the read operation into x. The transformation may be decomposed as follows: `Variable.bind_one`, `Instr.swap` (checking that no operations from T modifies the contents of x), `Expr.read_preceeding_write` (checking that e is reproducible), `Instr.swap` (checking that the evaluation of e commutes with all the instructions from T), then `Variable.inline_one`.

$$\boxed{\begin{array}{l} \mathrm{set}(p, e); \\ T; \\ E_{instr}[\mathrm{get}(p)]; \end{array}} \longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ T; \\ \textbf{let } x = \mathrm{get}(p); \\ E_{instr}[x]; \end{array}} \longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ \textbf{let } x = \mathrm{get}(p); \\ T; \\ E_{instr}[x]; \end{array}} \longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ \textbf{let } x = e; \\ T; \\ E_{instr}[x]; \end{array}} \longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ T; \\ \textbf{let } x = e; \\ E_{instr}[x]; \end{array}}$$

$$\longmapsto \boxed{\begin{array}{l} \mathrm{set}(p, e); \\ T; \\ E_{instr}[e]; \end{array}}$$

A.2 Function Inlining

Basic function inlining. The basic transformation `Function.inline_pure` inlines the body of a function at a call site where all the arguments are pure values. Recall that return statements from the C language are encoded in OptiTrust as special bindings of the form **let res**. During the inlining process, these bindings are replaced with writes into a fresh mutable variable, say r. Besides, recall that OptiTrust only supports return statements appearing at the end of a control-flow branch (i.e. abrupt termination is not supported yet); patterns of the form `if (c)return;` are encoded using if-then-else construct.

$$\boxed{\begin{array}{l}\textbf{let } f = \textbf{fun}(a_1, \ldots, a_n) \mapsto T;\\ \ldots\\ E_{instr}[f(v_1, .., v_n);\,]\end{array}} \longmapsto \boxed{\begin{array}{l}\textbf{let } f = \textbf{fun}(a_1, \ldots, a_n) \mapsto t;\\ \ldots\\ \textbf{let } r = \textbf{new}(\bot);\\ T';\\ E_{instr}[\mathrm{get}(r);\,]\end{array}}$$

where T' is a copy of T with occurrences a_i replaced with v_i, and with instructions of the form **let res** := e replaced with set(r, e).

Function inlining with simplifications. Our combined transformation `Function.inline` applies the necessary transformation to apply the basic transformation `Function.inline_pure`, then applies it, then performs a number of simplications. First, in case some of the arguments appearing in the call are not pure values, it introduces bindings for them. Then, it applies `Function.inline_pure`, and we attempt two simplifications. First, if the mutable variable r (to which the result of the function is assigned) is set only once, as part of the sequence of instruction, then we attempt to move declaration next to the assignement, and to apply `Variable.init_attach` and `Variable.to_const`. If this process applies, then the result produced by the function is bound by a simple let-binding. Furthermore, if that variable r, now pure, has exactly one occurence, its occurence may be inlined. (An optional argument is available to disable this inlining.) Second, for each argument for which a binding was introduced, if the the corresponding variable has exactly one occurrence, then a call to `Variable.inline` is attempted. In fact, the user may pass optional flags to request an inlining of an argument, even if it means duplicating the corresponding expression. The benefits of doing so is to limit the number of intermediate bindings introduced during the inlining process.

REFERENCES

Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, et al. 2020. Programming languages for data-intensive HPC applications: A systematic mapping study. *Parallel Comput.* 91 (2020), 102584.

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

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

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

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

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

Paul Barham and Michael Isard. 2019. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems.* 177–183.

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

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

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

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008a. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08).* Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008b. 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, Vol. 2694. 55–72. https://doi.org/10.1007/3-540-44898-5_4

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

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. *Journal of Automated Reasoning* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

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

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.

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

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

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

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

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

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

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

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–29.

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

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conf. on Programming Language Design and Implementation.* 703–718.

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)*. IEEE, 1–5.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018a. 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

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. *JFP* 28 (2018). https://doi.org/10.1017/S0956796818000151

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

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

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

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

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

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

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

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

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

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

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

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968 The appendix is linked as supplementary material from the ACM digital library..

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

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

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

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

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74. http://www.cs.cmu.edu/~jcr/seplogic.pdf

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

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

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

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

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003*. IEEE, 204–215.

Lars B. van den Haak, Anton Wijs, Marieke Huisman, and Mark van den Brand. 2024. HaliVer: Deductive Verification and Scheduling Languages Join Forces. arXiv:2401.10778 [cs.LO]

Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. 2018. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. *International Journal of High Performance Computing Applications* 32, 1 (Jan. 2018), 28–44. https://doi.org/10.1177/1094342017718068

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