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

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 makes 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 resource analysis that exploits contracts expressed in a simplified form of Separation Logic. Throught 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 50x—on a single core. For many applications, the code can then be accelerated further by one or two orders of magnitude by refining the code to exploit 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.

	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.

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.

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. Let us introduce a number of qualitative properties for comparing semi-automatic code optimization tools that rely on some form of user interaction.

- **Generality**: How large is the domain of applicability of the tool? In particular, is it restricted to a domain-specific language?
- **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? Does it moreover provide mechanized proofs?

Next, we review related tools for producing high performance code, before presenting our OptiTrust framework and explaining why it achieves a unique combination of features.

1.2 Related Work

Halide [Ragan-Kelley et al. 2013] is an industrial-strength domain-specific compiler for image processing, used for example to optimize code that runs in products like Adobe 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 applications to machine learning. Although Halide and TVM have demonstrated strength for particular applications, these tools are inherently limited to their domain-specific languages, 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). Besides, it may make harder the understanding of optimization strategies by the programmer, because a same optimization may take relatively different forms between a functional language and an imperative language familiar to the high-performance programmer. In particular, the chaining of optimizations generally leads to the introduction of cascades of *map* operations much less readable than if the corresponding operations appear in sequence in a for-loop.

Exo [Ikarashi et al. 2022] is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. Exo features for-loops, if-statements, arrays and procedures. 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 using 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. Where a tool like Pluto [Bondhugula et al. 2008a] acts as a black-box for optimizing such loop nests, Clay provides a decomposition of polyhedral optimizations (known as a *schedule*) 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 with 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 Overview

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.

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 location. 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 *resource-based type system*. This type system may be thought of as a variant of the Rust type system with augmented expressiveness a scaled down version of Separation Logic [Reynolds 2002]. Separation Logic has been successfully applied on languages ranging from machine code to high-level functional programming languages, to verify programs ranging from operating systems components to general-purpose data structures and algorithms [Charguéraud 2020; O'Hearn 2019]. 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.

Before describing an example optimization script, let us evaluate OptiTrust against the aforementioned criteria.

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-based type system.

For this first release of OptiTrust, we support only core features of the C language: sequences, loops, conditionals, functions, local and global variables, arrays, and structs. For the time being, we do not support break, continue, and non-terminal return statements. There is, however, no inherent limitation: OptiTrust could presumably be extended to support nearly all of the C language (leaving out general goto statements).

Regarding our type system, in the long term we aim for a full-featured Separation Logic similar to RefinedC [Sammler et al. 2021]. In technical terms, our design decisions are geared towards the support of arbitrary *assertions* for capturing all program invariants, user-defined *representation predicates* for describing advanced data structures, and *ghost operations* for logical transformations of the view over memory. At this stage, however, our implementation and case studies mainly demonstrate the manipulation of *shapes* predicates, mainly for describing n-dimensional arrays, without specifying the values stored in these arrays. The pure assertions that are manipulated in our case studies demonstrate mainly the use of arithmetic constraint.

As we show, shape predicates and basic arithmetic suffices to justify a large range of program optimizations. We leave it to future work the demonstration of how one could: (1) exploit invariants on values stored in data structures to justify certain optimizations; and (2) demonstrate how nontrivial invariants can be maintained through code transformations. The point of the present paper is to demonstrate the generality of OptiTrust in terms of being able to apply source-to-source transformations on code written in a general-purpose programming language.

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 catalogue of general-purpose transformations, whose composition enables altering the code with a lot of flexibility.

Let us give an overview of 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, loop splitting, introduction of sliding windows, and introduction and elimination of conditionals. For data layout transformations, we support: interchange of dimensions of an array, and array tiling.

The aforementioned transformations have been motivated by the case studies presented further in this paper. As we complete more case studies, additional transformations will be required. A key strength of OptiTrust is precisely that, unlike monolithical tools, it may be extended with additional transformations without affecting the behavior of existing transformation scripts. We discuss this aspect further in the paragraph on extensionality.

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 **??** presents 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.

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 custom transformations, OptiTrust provides: smart destructors for analyzing and recognizing the input AST, smart constructors for producing fresh subterms for the output AST, as well a term-rewriting facility based on a pattern-matching algorithm. OptiTrust's library provides numerous examples illustrating how to use these features.

For each custom transformation, it is the implementor'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. In the long term, we might be interested in the formal verification of the implementation of OptiTrust. Yet, such a verification endavour consists of a tremendous challenge, beyond the state-of-the-art in compiler verification. Moreover, even if we could tackle the verification of OptiTrust's builtin transformations, it is unlikely that every implementor of a custom transformation would have the expertise and budget to take on formal verification. We therefore designed OptiTrust in such a ways as to mitigate the risks of producing incorrect code.

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. The possibility to review, optimization by optimization, the differences between the reference code and the optimized code may be highly relevant in the context of safety- or security-critical applications.

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.

There is a third potential approach to increasing trust for code optimized using OptiTrust, similar to the one put forward in the aforementioned Alpinist [Sakar et al. 2022], and also used in prior work on the formal validation of programs generated by Halide [Clément and Cohen 2022; van den Haak et al. 2024]. This approach is not demonstrated in the present paper and is left to future work, nevertheless we find it important to mention it because it is at the heart of the original motivation for the OptiTrust project and of its future directions. The idea is that if the input program is fully verified with respect to functional correctness, and if the invariants are maintained throughout all code optimizations, then the correctness of the final optimized program can be validated, totally independently from the sequence of optimizations that have been performed. This validation step could be performed using a third-party tool (e.g., Coq), effectively removing the OptiTrut implementation from the trusted code base.

In summary, in the current version of OptiTrust, even without full functional correctness assertions, the combination of our *minimized trusted code base* approach combined with the possibly of *human-review of transformation reports* provides, we believe, a significant increase in trustworthiness compared with other compilers.

1.4 Contributions

In this paper, we make the following contributions.

- We introduce OptiTrust, an optimization framework that delivers a previously unmatched combination of features in terms of generality, expressiveness, control, feedback, composability, extensibility, and trustworthiness.
- We present criterias for checking the correctness of classical, general-purpose code transformations, with respect to resource usage information expressed in a type system that corresponds to a subset of Separation Logic.
- We explain, for the same classical code transformations, how to update contracts and how to insert or move ghost operations in order to ensure that the output code remains well-typed in our system.
- We introduce a targeting mechanism specialized for targeting locations in abstract syntax trees, allowing the user to concisely and robustly indicate where a transformation should be applied.
- We demonstrate, for the first time, the possibility to produce state-of-the-art high-performance code for 3 classic benchmarks, via a series of source-to-source transformations expressed at the level of C code.

OptiTrust is open-source and available at: https://github.com/charguer/optitrust. The implementation of OptiTrust involves about 25k lines of OCaml code. The regression suite contains 170 unit tests, featuring 880 individual steps. The traces associated with the 3 case studies presented in this paper can be navigated interactively at: TODO.

1.5 Contents of the Paper

We first present the features of OptiTrust by means of example, in Section ??. Then, we present the construction of OptiTrust in three parts. In Section 3, we describe the overall architecture of the implementation, including the reversible encoding of C code into the imperative λ-calculus. In Section 4, we present our target mechanism. In Section 5, we explain our resource-based typing algorithm. In Section 6, we present a set of representative code transformations, illustrating in particular how resource information is exploited to justify correctness, and how function and loop contracts are maintainted through transformations. Finally, we discuss related work in Section ??.

2 CASE STUDIES

[WORK IN PROGRESS]

3 THE OPTITRUST FRAMEWORK

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

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. Crucially, our encoding-decoding scheme is designed for round-trip stabilitiy: 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 to be valid C code. We discuss further on the language features that we do not yet handle. Besides, as we detail further on, 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, these annotations may indicate whether a variable is stack- or heap-allocated; whether an access is written `*(x.f)` or `x->f`; etc. These annotations are exploited during the decoding phase. Printing details such as spaces, tabulation, and line printing may not preserved with respect to the C code initially provided by the programmer. However, when 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 a 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] serve as as intermediate verificiation language for C, Java, and Ada programs. Viper [Müller et al. 2017] and Why3 [Filliâtre and Paskevich 2013] 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

This paper focuses on the encoding-decoding of C code. Presumably, we could apply a similar encoding-decoding scheme to other well-typed languages, such as OCaml, Rust, OpenCL/Cuda, Java, etc. Once the encoding-decoding is defined for another language, most of OptiTrust's code transformations, which expressed on OptiTrust's internal λ-calculus, become immediate to available for this language. We leave the investigation of other languages to future work.

3.2 Unsupported C Features and their Handling by OptiTrust

Our translation covers a subset of the C language. In particular, as of writing, our translation does support several features. Function pointers and variadic functions: we believe that there is no specific difficulty, however we have not yet implemented support for them. 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. General **goto** and inline assembly: we have no plan to support them in OptiTrust. Pre/post-increment/decrement operators: their semantics is highly nontrivial. For simplicity, at the moment we simply encode the statements `t++` and `++t` as `t += 1`, and we reject occurrences that of in-place increment operators that appear inside subexpressions. Likewise for in-place decrement operators. We leave it to future work to investigate how to leverage our type system to accept idomatic C patterns such as `t[i++] = v`.

Our type system covers a subset of the C language yet slighly smaller than the subset covered by our translation. Mainly, we do not yet support the control-flow operators **return**, **break**, and **continue**. 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, not only with respect to resource typing, but also with respect to loop transformations.

Based on these two layers of restrictions, we can classify C functions in 3 categories.

1. *Functions that OptiTrust is able to translate and to typecheck*. For these definitions, all OptiTrust code transformation are available, and they are guaranteed to be preserve the code semantics.
2. *Functions that OptiTrust is able to translate, yet unable to typecheck*. 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., read-last-write), or can be applied by the programmer yet without any correctness guaranteed (e.g., loop-swap).
3. *Functions that OptiTrust is unable to translate*. 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.

3.3 OptiTrust's Internal AST

Fig. 1 gives the grammar of OptiTrust's internal λ-calculus. 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. A special variable, named **res** is used to denote the result value of a function. As we will see, **return** t is encoded as "**let res** = t; **return**". Moreover **res** appears in function contrats to specify the return value.[1]

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.

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.

---

[1] 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].

π	$:=$	$\mid\ \mathbf{par}\ \mid\ \cdot$	"parallel" flag on for-loops
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
		$\mid\ (t_1; \ldots; t_n)\ \mid\ \mathbf{let}\ x = t\ \mid\ t_0(t_1, \ldots, t_n)$	sequence, declaration, and function call
		$\mid\ \mathsf{alloc}\ \mid\ \mathsf{get}\ \mid\ \mathsf{set}\ \mid\ \mathsf{free}$	primitive operations on memory cells
		$\mid\ \mathsf{ref}\ \mid\ \mathsf{ref\_uninit}$	allocations of local memory cells
		$\mid\ t_1[t_2]\ \mid\ t_1.f$	projection from array/struct values
		$\mid\ t_1 \boxplus t_2\ \mid\ t_1 \boxdot f$	address computations
		$\mid\ \mathbf{for}^{\pi}(i \in r)\ t_{\text{body}}$	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
		$\mid\ \mathbf{return}\ \mid\ \mathbf{break}\ \mid\ \mathbf{continue}$	control-flow operators (no return value)

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

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, as well as control-flow operators such as return, break, and continue. A C source file is also described as a sequence, which may moreover contain declarations of types, functions, and global variables.

Primitive operations are provided 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 $\mathsf{ref}(t)$ allocates a memory cell initialized with a specific contents t. The operation operation $\mathsf{ref\_uninit}(())$ allocates an uninitialized memory cell—in which read operations have undefined behavior. These two operations are meant to occur as part of a let-binding, e.g. $\mathbf{let}\ x = \mathsf{ref}(t)$. We have considered the possibility of encoding ref and ref_uninit using alloc and free, but ultimately concluded that this approach is not practical.[2]

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 a can be computed by the operation $t \boxplus i$. This operations to the C pointer arithmetic operation `t+i`. The contents of that cell may be retrived by evaluating $\mathsf{get}((t \boxplus i))$. 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 $\mathbf{for}^{\pi}(i \in \mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}}))\ t_{\text{body}}$ 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

---

[2] Using alloc and free to encode the behavior of stack-allocated variables would introduce additional statements in the OptiTrust AST that have no not correspond to any line in the C code. The presence of such extra "hidden" statements makes it very difficult to retain an intuitive behavior for target resolution. An alternative approach would be to display the additional statements to the end user, however we have found that making free operations explicit for every local mutable variable is fairly verbose and harms readability of the rest of the code.

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. 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`.[3]

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.

3.4 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.

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

Observe that unique variable identifiers are also applied to linear resources, even though the name of these resources might not be displayed to the programmer. For a given linear resource, its identifier remains the same only until the point where the resource is consumed. Further on, if the same resource is recovered, it is assigned a fresh identifiers. One exception is for a read-only resources. If a piece of a read-only is carved out, what remains of the resource retains its current

---

[3] The restrictions imposed by OpenMP on the ranges of parallel for-loops essentially constraint them to fit the format $\mathbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$, which we use for simple-for-loops.

$$
\begin{array}{lcl}
\lfloor x \rfloor & = & \left| \begin{array}{ll} \mathrm{get}(x) & \text{if } x \in \Gamma \\ x & \text{otherwise} \end{array} \right. \\
\lfloor \&u \rfloor & = & t \quad \text{where } \lfloor u \rfloor \text{ is guaranteed to be of the form } \mathrm{get}(t) \\
\lfloor {*u} \rfloor & = & \mathrm{get}(\lfloor u \rfloor) \\
\lfloor u_1 = u_2 \rfloor & = & \mathrm{set}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) \\
\lfloor u_1 \mathrel{+}= u_2 \rfloor & = & \mathrm{set\_add}\ (\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) \\
\lfloor u_1\mathbf{[}u_2\mathbf{]} \rfloor & = & \lfloor u_1 \rfloor \mathbf{[}\lfloor u_2 \rfloor\mathbf{]} \\
\lfloor u.f \rfloor & = & \left| \begin{array}{ll} \mathrm{get}(t \boxdot f) & \text{if } \lfloor u \rfloor \text{ is of the form } \mathrm{get}(t) \\ \lfloor u \rfloor.f & \text{otherwise} \end{array} \right. \\
\lfloor T\ x = u; \rfloor & = & \left| \begin{array}{lll} \mathbf{let}_T\ x = \lfloor u \rfloor & & \text{if } x \text{ immutable} \\ \mathbf{let}_{(T^*)}\ x = \mathrm{ref}(\lfloor u \rfloor) & \text{with } x \text{ added to } \Gamma & \text{otherwise} \end{array} \right. \\
\lfloor T\ x; \rfloor & = & \mathbf{let}_{(T^*)}\ x = \mathrm{ref\_uninit}() \qquad \text{with } x \text{ added to } \Gamma \\
\lfloor \mathbf{for}\ (\mathrm{int}\ i{=}u_1; i{<}u_2; i{+}{=}u_3)\ u_4 \rfloor & = & \mathbf{for}(i \in \mathbf{range}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor, \lfloor u_3 \rfloor))\ \lfloor u_4 \rfloor \\
\left\lfloor \begin{array}{l} \mathbf{\#pragma\ openmp\ parallel} \\ \mathbf{for}\ (\mathrm{int}\ i{=}u_1; i{<}u_2; i{+}{=}u_3)\ u_4 \end{array} \right\rfloor & = & \mathbf{for}^{\mathbf{par}}(i \in \mathbf{range}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor, \lfloor u_3 \rfloor))\ \lfloor u_4 \rfloor \\
\lfloor \mathbf{for}\ (u_1; u_2; u_3)\ u_4 \rfloor & = & \{\lfloor u_1 \rfloor; \mathbf{while}\ \lfloor u_2 \rfloor\ \mathbf{do}\ \{\lfloor u_4 \rfloor\}; \lfloor u_3 \rfloor\} \\
\lfloor t \rfloor \text{ for other terms} & = & \text{apply the translation recursively on every subterm}
\end{array}
$$

Fig. 2. Translation from C to OptiTrust's internal λ-calculus.
A global context Γ keeps track of the identifiers of mutable variables.

identifier. Symmetrically, if the carved out piece is merged back, the resulting resource retains the same identifier as the original read-only resource.

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. Furthemore, as we will see in the next section, identifiers for linear resources serve as key in the data structures that describe the *usage* information of every subterm.

3.5 Encoding and Decoding of C Code

Fig. 2 defines our translation from C to OptiTrust's internal language. Fig. 3 defines the reciprocal translation. In the figures, we write $\lfloor u \rfloor$ the encoding of a C term u (which could be either a statement or an expression). We write $\lceil t \rceil$ the decoding of an OptiTrust term t.

The encoding process essentially performs two tasks: (1) it eliminates the notion of l-value, instead manipulating addresses explicitly; (2) it eliminates stack-allocated mutable variables, viewing them like heap-allocated data; The decoding process applies exactly the opposite steps.

As mentioned earlier, during the encoding, a number of "style" annotations can be attached to the terms produced, in order to guide the decoding phase and ensure the round-trip property. Importantly, these annotations do not matter with respect to the semantics. It is always safe to drop annotations in the OptiTrust AST. Fig. **??** omits the details about annotations.

We prove the round-trip theorem: if u is a valid C program, and if u does not contain spurious `&*u` or `*&u` patterns, then $\lfloor u \rfloor$ is well-defined and $\lceil \lfloor u \rfloor \rceil = u$. (If the spurious patterns occur, they are simply eliminated by the round-trip.)

4 TARGETS IN OPTITRUST

[WORK IN PROGRESS]

$$
\begin{array}{lcl}
\lceil t \rceil^{\mathsf{L}} & = & \left| \begin{array}{ll} u & \text{if } \lceil t \rceil \text{ is of the form } \&u \\ *\lceil t \rceil & \text{otherwise} \end{array} \right. \\
\lceil x \rceil & = & \left| \begin{array}{ll} \&x & \text{if } x \in \Gamma \\ x & \text{otherwise} \end{array} \right. \\
\lceil \mathrm{get}(t) \rceil & = & \lceil t \rceil^{\mathsf{L}} \\
\lceil \mathrm{set}(t_1,\, t_2) \rceil & = & \lceil t_1 \rceil^{\mathsf{L}} = \lceil t_2 \rceil \\
\lceil \mathrm{set\_add}\,(t_1,\, t_2) \rceil & = & \lceil t_1 \rceil^{\mathsf{L}} \mathrel{+}= \lceil t_2 \rceil \\
\lceil t_1 [\![t_2]\!] \rceil & = & \lceil t_1 \rceil [\![\lceil t_2 \rceil]\!] \\
\lceil t.f \rceil & = & \lceil t \rceil.f \\
\lceil t_1 \boxplus t_2 \rceil & = & \& \lceil t_1 \rceil [\![\lceil t_2 \rceil]\!] \\
\lceil t \boxdot f \rceil & = & \& \lceil t \rceil^{\mathsf{L}}.f \\
\lceil u.f \rceil & = & \left| \begin{array}{ll} \mathrm{get}(t \boxdot f) & \text{if } \lceil u \rceil \text{ is of the form } \mathrm{get}(t) \\ \lceil u \rceil.f & \text{otherwise} \end{array} \right. \\
\lceil \mathbf{let}_{(T^*)}\, x \;=\; \mathrm{ref\_uninit}() \rceil & = & T\,x; \qquad \text{with } x \text{ added to } \Gamma \\
\lceil \mathbf{let}_{(T^*)}\, x \;=\; \mathrm{ref}(t) \rceil & = & T\,x = \lceil t \rceil; \qquad \text{with } x \text{ added to } \Gamma \\
\lceil \mathbf{let}_{T}\, x \;=\; t \rceil & = & T\,x = \lceil t \rceil; \qquad \text{for other let-bindings} \\
\lceil \mathbf{for}(i \in \mathbf{range}(t_1,\, t_2,\, t_3))\; t_4 \rceil & = & \mathbf{for}\ (\mathrm{int}\ i{=}\lceil t_1 \rceil;\, i{<}\lceil t_2 \rceil;\, i{+}{=}\lceil t_3 \rceil)\ \lceil t_4 \rceil \\
\lceil \mathbf{for}^{\mathbf{par}}(i \in \mathbf{range}(t_1,\, t_2,\, t_3))\; t_4 \rceil & = & \left\{ \begin{array}{l} \#\mathbf{pragma\ openmp\ parallel} \\ \mathbf{for}\ (\mathrm{int}\ i{=}\lceil t_1 \rceil;\, i{<}\lceil t_2 \rceil;\, i{+}{=}\lceil t_3 \rceil)\ \lceil t_4 \rceil \end{array} \right. \\
\lceil \{ t_1;\, \mathbf{while}\ t_2\ \mathbf{do}\ \{ t_4 \};\, t_3 \} \rceil & = & \mathbf{for}\ (t_1;\, t_2;\, t_3)\ t_4 \quad \text{if term annotated as for-loop} \\
\lceil t \rceil \text{ for other terms} & = & \text{apply the translation recursively on every subterm}
\end{array}
$$

Fig. 3. Translation from OptiTrust's internal λ-calculus back to C.
A global context Γ keeps track of the identifiers of mutable variables.

5 COMPUTING PROGRAM RESOURCES

As we have illustrated through Section 2, resource typing is key to obtaining information that is precise sufficiently for justifying numerous practical code transformations. This section explains the details of the type checking algorithms, as well as the design choices behind it.

5.1 Overview of the typing strategy

As we have seen through examples, typing with resources requires a number of ghost operations for rearranging the view on memory/resources. In our design, some of these ghost operations are materialized as ghost instructions that appear explicitly in sequences, whereas certain classes of ghost operations are performed on-the-fly. Let us motivate this design choice.

We are seeking for a good tradeoff between:

- robustness of type-checking (after a transformation, code needs to remain well-typed)
- understandability of transformations (the user needs to see ghosts to understand why a transformation fails to apply)
- readability of transformed programs (too many ghosts harm readability)
- effort for writing the initial program (ghosts are very tedious to write)
- efficiency of typechecking (inferring ghosts may be costly)

The most important criteria is robustness. If we have too many implicit ghost operations, then type-checking becomes fragile: a local modification on a well-typed program may turn it into an ill-typed program with no obvious way of fixing it. Besides, inferring implicit ghost operations over and over again can be quite costly, and limit scalability.

Can we go for fully explicit, i.e. have ghost operations all explicit? Then, the code becomes hard to read, and every harder to write. Let us separate the two matters.

For reading code, we could attempt to mitigate the issue by hiding certain ghost operations. Yet, hiding instructions makes it harder to get a target system to work well. Instead, we carved a carefully-chosen set of ghost operations that represent a significant fraction of ghost operations, and that we know can be robustly recovered after code transformations. We go implicit for these ghost operations, and explicit for all others.

There remains the challenge of making it realistic to write the unoptimized code. Our initial attempts at case studies revealed that even with the policy above, it appears too tedious for the user to write numerous ghost operations and loop contracts, even if seeing those information is useful for subsequently transforming the code.

We therefore opted for an approach where we perform an advanced elaboration phase for the original input code, to infer a number of ghost operations. After this initial elaboration phase, the typechecking algorithm remains simple. (We can afford to spend more time on typechecking the first AST than on typechecking all the pieces of AST that are subsequently produced during the execution of the transformation script.)

In what follows, we first describe the typechecking algorithms, and only afterwards describe the elaboration algorithm. We also present the loop contract minimization algorithm, which is useful both for the elaboration algorithm and is used as postprocessing for most loop-based transformations.

5.2 Top-down typechecking and bottom-up summaries

Our typechecking algorithm is a top-down algorithm. This approach has the following benefits:

- Efficiency: typechecking is performed in a single pass over the AST.
- Simplicity: the typing rules are standard and simple
- 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.

Once typechecking is completed, we know for every statement what resources it consumes and produces. To verify the validity of transformations, and to compute the results of transformations, it helps a lot to have efficient access to a different presentation of the same information. Typically, we need to know for each resource how it used by the statements that depend on it. The "usage" can be read-write, read-only, consumed, etc.

All these informations are attached to the AST nodes.

For technical reasons explained later, we sometimes need to store in the pure context what we call *existential fractions*. These are abstract fractions that can be chosen later during the typing, as long as some constraints are respected. When present, these existential fractions are stored along with their constraints in a separate field E' in a context $\langle E \mid E' \mid F \rangle$.

5.3 Resource sets

In our system, a typing context consists of typed variables (also called *pure resources*) and *linear resources*. Our typing algorithm computes the set of resources at every program point.

Pure resources. The pure part of a typing context contains bindings of the form “$x : \tau$”. The variable x may be either a program variable, in which case τ corresponds to its C type ; or a ghost variable, in which case τ can be any *mathematical type*. A mathematical type can be thought of as Coq types (or types of another higher order logic). In particular, it includes types such as $\mathbb{Z}$, finite

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. 4. Common heap predicates

or infinite sets, but also C types (viewed as a deep embedding). Mathematical types also include propositions: for example $p : n > 0$ describes a proof p establishing $n > 0$. In summary, the pure part of a typing context is an interleaving of a traditional program typing context and of a Coq context.

Linear resources. The linear part of a typing context contains bindings of the form "$y : H$". The resource name y is used in particular for the usage maps to refer to this resource. The heap predicate H describes ownership of part of the memory. Fig. 4 summarizes the most common heap predicates, which have already been discussed in Section **??**, in particular, $p \rightsquigarrow \mathsf{Matrix1}_\tau(n)$ is syntactic sugar for $\bigstar_{i \in 0..n}\, p[i] \rightsquigarrow \mathsf{Cell}_\tau$. Likewise, $p \rightsquigarrow \mathsf{Matrix2}_\tau(n, m)$ denotes $\bigstar_{i \in 0..n} \bigstar_{j \in 0..m}\, p[i][j] \rightsquigarrow \mathsf{Cell}_\tau$.

Read-only fractions. Following standard separation logic, we represent read-only permissions using *fractional resources*. Intuitively, possessing a non-null fraction of a linear resource gives read-only access. Possessing the full fraction (i.e. one) of a resource gives read-write access. The pair of the resources $\alpha H \star \beta H$ entails $(\alpha + \beta)H$, and reciprocally. In practice, when we have αH at hand, we can carve out a subfraction βH, leaving as remainder $(\alpha - \beta)H$. We carve out subfractions in such a way each time we need to provide a read-only permission. This strategy ensures that we always keep at hand a fraction of the read-only permission. At some point, we need to merge back βH and $(\alpha - \beta)H$ into the original αH. Because the carve-out operation can be performed in cascade, and that merge-back operations can be performed in any order, we need a general simplification operation. We call this operation CloseFracs. Formally, CloseFracs repeats the following rewrite rule:

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

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

Permissions on uninitialized cells. A standard separation logic for C code ensures that there is no reads of an uninitialized memory cell, because it would be undefined behavior. To achieve this a read is allowed with permission $p \rightsquigarrow v$ but with a side condition that $v \neq \bot$, where $\bot$ is a special token denoting uninitialized content. Rather than introducing $p \rightsquigarrow \bot$ in our logic, we introduce permissions of the form $\mathsf{Uninit}(H)$ to describe not only individual uninitialized cells but also uninitialized arrays and matrices. Concretely, $\mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell})$ corresponds to $p \rightsquigarrow \mathsf{Cell}$ with the additional constraint that reading the memory at location p is forbidden. 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. If $\mathsf{Uninit}(H)$ is well-defined, it can be obtained by weakening from H.

Notations for resource sets. 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 of type H_i. Moreover, certain bindings $x_i : \tau_i$ may be *alias definitions* of the form $x_i : \tau_i := v_i$, which corresponds to a local definition, and may also be interpreted as a singleton type. In practice, we simply write $x_i := v_i$ because τ_i can be inferred. 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.

This organization separating pure facts (either conventional bindings or alias bindings) and linear facts is directly inspired by practical tools based on separation logic (e.g. Iris, CFML). 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.)

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.

Operators on resource sets. 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 $\Gamma_1 \star \Gamma_2$ 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 $\mathsf{int} \rightarrow \mathsf{bool}$ 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:

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

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

Substitutions, specialization and renaming in resource sets. 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 $\mathsf{Specialize}\{\sigma\}(\Gamma)$ to eliminate certain bindings from Γ, substituting the corresponding occurrences with specified values. This operation assumes $\mathsf{dom}(\sigma)$ to be included in set of keys of $\Gamma.\mathsf{pure}$. Concretely, $\mathsf{Specialize}\{x := v\}(\langle E_1,\ x : \tau,\ E_2 \mid F\rangle)$ evaluates to $\langle E_1,\ \mathsf{Subst}\{x := v\}(E_2) \mid \mathsf{Subst}\{x := v\}(F)\rangle$. More generally,

$$\begin{aligned} \mathsf{Specialize}\{\sigma\}(\langle x : \tau,\ E \mid F\rangle) \ &:= \begin{cases} \mathsf{Specialize}\{\sigma'\}(\mathsf{Subst}\{x := v\}(\langle E \mid F\rangle)) & \text{when } \sigma = \{x := v\} \uplus \sigma' \\ [x : \tau] \star \mathsf{Specialize}\{\sigma\}(\langle E \mid F\rangle) & \text{when } x \notin \mathsf{dom}(\sigma) \end{cases} \\ \mathsf{Specialize}\{\emptyset\}(\langle E \mid F\rangle) \ &:= \ \langle E \mid F\rangle \end{aligned}$$

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, $\mathsf{Rename}\{x := x'\}(\langle E_1,\ x :$

τ, $E_2 \mid F\rangle$) evaluates to $\langle E_1, x' : \tau, \mathrm{Subst}\{x := x'\}(E_2) \mid \mathrm{Subst}\{x := x'\}(F)\rangle$. Rename can also be used to rename the linear resources: for example $\mathrm{Rename}\{y := y'\}(\langle E \mid F_1, y : H, F_2\rangle)$ evaluates to $\langle E \mid F_1, y' : H, F_2\rangle$.

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.

5.4 Contracts

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

Function contracts. A function definitions annotated with a contract γ takes the form $\mathbf{fun}(a_1, ..., a_n)_\gamma \mapsto t$. Here γ consists of two resource sets, one for the pre-condition, one for the post-condition. Formally, we write it $\{\mathrm{pre} = \Gamma_{pre}\ ;\ \mathrm{post} = \Gamma_{post}\}$. 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 $\gamma.\mathrm{pre}$, (subfractions of) the resources in F and Γ. The loop body needs to produce the resources $\gamma.\mathrm{post}$, it needs to give back the resources from F that it recieved, and produce the resources $\mathrm{Subst}\{i := i + 1\}(\Gamma)$. 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 $\chi.\mathrm{shrd.inv} = \emptyset$). We write $\mathrm{parallelizable}(\chi)$ in this case.

5.5 Typechecking of terms

Triples. Our typing judgement takes the form $\{\Gamma\}\ t\ \{\Gamma'\}$, capturing the fact that, in context Γ the term t is well typed and produces a context Γ'. If t has a return value, then, by convention, it 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 $\{\Gamma\}\ t\ \{\Gamma'\}$, the postcondition Γ' repeats all the pure entries of the precondition Γ. The pure entries that appear in Γ' but not in Γ may correspond: (1) to the entry 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 entries of Γ' may be arbitrarily modified compared with those in Γ, reflecting on the side-effects performed by t.

Typing rules. In the rest of 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 $\{\Gamma\}\ t\ \{\Gamma'\}$, Γ and t are inputs whereas Γ' is an output.

When the typing algorithm checks that a postcondition is met, it verifies that an *entailment* holds. We denote $\Gamma \Rrightarrow \Gamma'$ such entailment. Formally, $\Gamma \Rrightarrow \Gamma'$ holds if there exist a map σ with an entry $x := v$ 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 $\mathsf{Specialize}\{\sigma\}(\Gamma')$ that can be unified together.

When there is a contract instantiation, we need to check that we can instantiate all the pure variables required by the contract, and provide all the linear resources consumed. For that we use a context subtraction operation $\Gamma \ominus \Gamma'$ that fails if resources in Γ' cannot be found in Γ and return **Some** (σ, F) otherwise. In the latter case, σ is a map from pure variables of Γ' to instantitation 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, when **Some** $(\sigma, F) = \Gamma \ominus \Gamma'$, F is one of the strongest linear resource sets such that $\mathrm{dom}(\sigma) = \Gamma.\mathsf{pure}$ and that $\Gamma \Rrightarrow \mathsf{Specialize}\{\sigma\}(\Gamma') \star F$.

The entailment algorithm that decides if premises of the form $\Gamma \Rrightarrow \Gamma'$ holds an how $\Gamma \ominus \Gamma'$ is computed will be discussed in next 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 $get(x)$.

$$
\begin{array}{rcll}
v & ::= & x & \text{Variable} \\
 & | & \mathbb{N} & \text{Integer literal} \\
 & | & \mathbb{F} & \text{Float literal} \\
 & | & v \boxplus v & \text{Pure operation}
\end{array}
$$

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 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. The rule for typing a sequence 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 last instruction of the sequence has a return value, we consider it is the return value of the whole sequence. The temporary x_n in the rule is only here for symmetry with other instructions, in practice we omit both renamings mentioning x_n.

Sequence is the only constructor that delimits a new scope of program variable. We take a conservative approach for pure typing context scopes: when a sequence is exited, each immutable program varaible 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

Val

$$\frac{}{\{\Gamma\}\ v\ \{\Gamma \star [\mathbf{res} := v]\}}$$

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

$$\frac{\begin{array}{c} x_i \text{ fresh} \qquad \forall i \in [1, n].\ \ \{\Gamma_{i-1}\}\ t_i\ \{\Gamma_i'\}\ \wedge\ \Gamma_i = \mathrm{Rename}\{\mathbf{res} := x_i\}(\Gamma_i') \\ \Gamma_r = \begin{cases} \mathrm{Rename}\{x_i := \mathbf{res}\}(\Gamma_n) & \text{if } t_i \text{ is of the form ``}\mathbf{let\ res} = t_i'\text{''} \\ \Gamma_n & \text{otherwise} \end{cases} \\ \mathbf{Some}\ (\emptyset, \Gamma_f) = \Gamma_r \ominus \mathrm{StackAllocCells}(t_1, ..., t_n) \end{array}}{\{\Gamma_0\}\ \big(t_1; ...; t_n\big)\ \{\Gamma_f\}}$$

Fun

$$\frac{\begin{array}{c} \{[\Gamma_0.\mathrm{pure}] \star [a_1 : \tau_1, ..., a_n : \tau_n] \star \gamma.\mathrm{pre}\}\ t\ \{\Gamma_1\} \qquad \Gamma_1 \Rightarrow \gamma.\mathrm{post} \\ P = \{[a_1 : \tau_1, ..., a_n : \tau_n] \star \gamma.\mathrm{pre}\}\ \mathbf{res}(a_1, ..., a_n)\ \{\gamma.\mathrm{post}\} \end{array}}{\{\Gamma_0\}\ \big(\mathbf{fun}(a_1 : \tau_1, ..., a_n : \tau_n)_\gamma \mapsto t\big)\ \{\Gamma_0 \star [P]\}}$$

Subexpr

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

App

$$\frac{\begin{array}{c} \mathrm{dom}(\rho) = \mathrm{dom}(\gamma.\mathrm{post}) \qquad \mathrm{im}(\rho) \cap \mathrm{dom}(\Gamma_0) = \emptyset \\ \Gamma_0 \ni \{\gamma.\mathrm{pre}\}\ x_0(a_1, ..., a_n)\ \{\gamma.\mathrm{post}\} \qquad \mathbf{Some}\ (\sigma', \Gamma_f) = \Gamma_0 \ominus \mathrm{Specialize}\{a_i := x_i, \sigma\}(\gamma.\mathrm{pre}) \\ \Gamma_p = \mathrm{CloseFracs}(\Gamma_f \star \mathrm{Rename}\{\rho\}(\mathrm{Subst}\{(a_i := x_i), \sigma, \sigma'\}(\gamma.\mathrm{post}))) \end{array}}{\{\Gamma_0\}\ x_0(x_1, ..., x_n)_{\sigma,\rho}\ \{\Gamma_p\}}$$

For

$$\frac{\begin{array}{c} \mathbf{Some}\ (\sigma, \Gamma_f) = \Gamma_0 \ominus [\chi.\mathrm{vars}] \star (\bigstar_{i \in r}\ \chi.\mathrm{excl.pre}) \star \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{first}\}(\chi.\mathrm{shrd.inv}) \\ \Gamma_1 = [i : \mathrm{int}, i \in r] \star [\chi.\mathrm{vars}] \star \chi.\mathrm{excl.pre} \star \frac{1}{r.\mathit{len}} \chi.\mathrm{shrd.reads} \star \chi.\mathrm{shrd.inv} \\ \{\Gamma_1\}\ t_b\ \{\Gamma_2\} \qquad \Gamma_2 \Rightarrow \chi.\mathrm{excl.post} \star \frac{1}{r.\mathit{len}} \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{next}(i)\}(\chi.\mathrm{shrd.inv}) \\ \Gamma_3 = \mathrm{CloseFracs}(\Gamma_f \star \mathrm{Subst}\{\sigma\}(\bigstar_{i \in r}\ \chi.\mathrm{excl.post} \star \chi.\mathrm{shrd.reads} \star \mathrm{Subst}\{i := r.\mathit{last}\}(\chi.\mathrm{shrd.inv}))) \\ \pi = \mathrm{parallel} \longrightarrow \mathrm{parallelizable}(\chi) \end{array}}{\{\Gamma_0\}\ \mathbf{for}^{\,\pi}(i \in r)_\chi\ t_b\ \{\Gamma_3\}}$$

If

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

Fig. 5. Rules of our typesystem

never forgetting any variable tends to blow up pure context size, and we should apply some context filtering in future work.

For the stack allocated variables that go out of scope we need to consume their cells at the end of the sequence. The operator StackAllocCells$(t_1, \ldots, t_n)$ returns the resource set of cells that were

$$\begin{array}{rcl}
\{[v : \tau]\} & \mathsf{ref}(v) & \{[\mathbf{res} : \mathsf{ptr}] \star \mathbf{res} \rightsquigarrow \mathsf{Cell}_\tau\} \\
\{\} & \mathsf{ref\_uninit}() & \{[\mathbf{res} : \mathsf{ptr}] \star \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_\tau)\} \\
\{\} & \mathsf{alloc}() & \{[\mathbf{res} : \mathsf{ptr}] \star \mathsf{Uninit}(\mathbf{res} \rightsquigarrow \mathsf{Cell}_\tau)\} \\
\{[p : \mathsf{ptr}] \star p \rightsquigarrow \mathsf{Cell}_\tau\} & \mathsf{get}(p) & \{[\mathbf{res} : \tau] \star p \rightsquigarrow \mathsf{Cell}_\tau\} \\
\{[p : \mathsf{ptr}, v : \tau] \star \mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell}_\tau)\} & \mathsf{set}(p, v) & \{p \rightsquigarrow \mathsf{Cell}_\tau\} \\
\{[p : \mathsf{ptr}] \star \mathsf{Uninit}(p \rightsquigarrow \mathsf{Cell}_\tau)\} & \mathsf{free}(p) & \{\}
\end{array}$$

Fig. 6. Contracts of built-in functions

allocated on the stack at top-level. Formally,

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

Function abstraction. When typing a function abstraction, the typing algorithm 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 $\gamma.\mathrm{pre}$. After typing the body of the function, the type-checker verifies that the output context entails the postcondition $\gamma.\mathrm{post}$. The function abstraction itself is a pure operation that simply add a binding for **res** as a function of spec γ. The syntax $\{\gamma.\mathrm{pre}\}\ \mathbf{res}(a_1, ..., a_n)\ \{\gamma.\mathrm{post}\}$ 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. 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.

SUBEXPR evaluates arguments subexpressions in parallel ensuring there is no interference between them. In this rule $E[t_0, ..., t_n]$ 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’

$$\frac{\begin{array}{c}\Gamma_0 \Rightarrow \left(\bigstar_{i\in[0,n]}\, \hat{\Gamma}_i\right) \star \hat{\Gamma}_r \\ \forall i \in [0,n].\ \{\hat{\Gamma}_i\}\ t_i^{\Delta_i}\ \{\hat{\Gamma}''_i\} \qquad \hat{\Gamma}'_i = \langle \hat{\Gamma}''_i.\mathrm{pure} \cap \mathit{ensured}(\hat{\Delta}_i) \mid \hat{\Gamma}''_i.\mathrm{linear}\rangle \\ \Gamma_c = \mathsf{CloseFracs}(\hat{\Gamma}_r \star \bigstar_{i\in[0,n]}\, \mathsf{Rename}\{\mathbf{res} := x_i\}(\hat{\Gamma}'_i)) \qquad \{\Gamma_c\}\ E[x_0, ..., x_n]\ \{\Gamma_p\}\end{array}}{\{\Gamma_0\}\ E[t_0, ..., t_n]\ \{\Gamma_p\}}$$

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. In the Subexpr rule, the first subexpression gets typed in the full context Γ_0, however while typing it, we learn that only $\hat{\Gamma}_0$ is actually needed, and that Γ_1 was untouched. Therefore we can use this Γ_1 as the typing context for the next subexpression without risking to create an interference. It follows that, by using the $\hat{\Gamma}_i$ iteratively found in Subexpr, the rule Subexpr' is applicable whenever Subexpr is. Note that 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}.

We introduce the operation $\mathsf{Minimize}(\Gamma, \Gamma', \Delta)$ the operation on a precondition Γ, a postcondition Γ' and a usage map Δ. Typically these three arguments come from a valid typing judgement $\{\Gamma\}\ t^{\Delta}\ \{\Gamma'\}$. Minimize returns a triple $(\hat{\Gamma}, \hat{\Gamma}', \Gamma_f)$ such that $\{\hat{\Gamma}\}\ t^{\Delta}\ \{\hat{\Gamma}'\}$, and intuitively Γ_f is a maximal frame removed from both Γ and Γ'.

- $\hat{\Gamma}$ is a *minimized precondition* that contains resources of Γ or weakened versions of resources of Γ. The selection of resources is guided by Δ. In particular, when Δ tells that a given resource $y : H$ is used in read only mode, we weaken it to an arbitrary subfraction αH. Similarly, when Δ tells that a given resource $y : H$ is used as an uninitialized varaible we weaken it as $\mathsf{Uninit}(H)$. Formally,

  $$\begin{aligned}\hat{\Gamma} = {} & [\Gamma.\mathsf{pure} \cap \mathit{required}(\Delta)] \star (\Gamma.\mathsf{linear} \cap \mathsf{full}(\Delta)) \\ & (\star\ \mathsf{IntoRO}(\Gamma.\mathsf{linear} \cap \mathsf{RO}(\Delta))) \star (\mathsf{IntoUninit}(\Gamma.\mathsf{linear} \cap \mathsf{Uninit}(\Delta)))\end{aligned}$$

  where

  $$\begin{array}{rcll} \mathsf{IntoRO}(y : \alpha H, F) & = & [\beta : \mathsf{frac}] \star (y : \beta H) \star \mathsf{IntoRO}(F) & \\ \mathsf{IntoRO}(y : H, F) & = & [\alpha : \mathsf{frac}] \star (y : \alpha H) \star \mathsf{IntoRO}(F) & \text{where } H \text{ is not of the form } \alpha H \\ \mathsf{IntoRO}(\emptyset) & = & \emptyset & \\ \mathsf{IntoUninit}(y : \mathsf{Uninit}(H), F) & = & y : \mathsf{Uninit}(H) \star \mathsf{IntoUninit}(F) & \\ \mathsf{IntoUninit}(y : H, F) & = & y : \mathsf{Uninit}(H) \star \mathsf{IntoUninit}(F) & \text{where } H \text{ is not of the form } \mathsf{Uninit}(H) \\ \mathsf{IntoUninit}(\emptyset) & = & \emptyset & \end{array}$$

  Note that IntoRO and IntoUninit need only be defined on resources that have been recorded with a usage RO or Uninit respectively.
- Γ_f is the *maximal frame*. It is a context such that $\Gamma \Rightarrow \hat{\Gamma} \star \Gamma_f$. We put all the pure varaibles from Γ in Γ_f.
- $\hat{\Gamma}'$ is the new postcondition after removing the frame Γ_f in Γ'.

The rule App for the function application searches in the context Γ_0 a specification for the called function. Then, it instantiates the precondition of the function by finding a pure variable substitution σ', and consuming pure resources in Γ_0 thus creating the frame context Γ_f. Then it add 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. Each binding $x := v$, where x is a pure variable of the function precondition forces to instantiate it with v. It must be used whenever the value v cannot be found by unification.

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 typing algorithm is concerned, these ghost calls can be seen as regular function calls without return value. They are typed using the same rule as any other function without program arguments.

6 JUSTIFYING TRANSFORMATION CORRECTNESS

We now explain how program resources are leveraged to express sufficient correctness conditions for a collection of basic transformations. We do not further discuss the correctness of combined transformations or entire transformation scripts in this section, because it is checked by transitivity of basic transformation correctness.

Our correctness conditions are checked by exploiting the resource usage information computed by our type system. For every transformation, we also explain how they produce well-typed programs by synthesizing contracts and ghost operations, such that multiple transformations can be chained.

When chaining multiple transformations, even on well-typed programs, it can happen that the generated contracts and ghosts become complicated. Instead of complexifiying our basic transformations, including their correctness conditions and annotation synthesis, we rely on combined transformations to simplify or move annotations around. For example, combined transformations may decide to minimize contracts, to push certain ghosts upwards, downwards, or to attach ghosts to certain instructions.

Similarly, we prefer deriving complex transformations by composition to keep basic transformations as simple as possible: for example by recursively fusing N loops two-by-two instead of defining a basic transformation directly fusing N loops.

We now discuss the most interesting basic transformations operating on instructions, loops and variables; before briefly mentioning other supported transformations.

6.1 Instruction Transformations

Moving Instructions. The `Instr.move` transformation allows moving a group of instructions to a given destination. Two instructions can be safely swapped when they exclusively share read-only resources:

$$\begin{array}{c}\overline{T_1;\Delta_1}\\ \underline{T_2;\Delta_2}\end{array} \quad\longmapsto\quad \begin{array}{c}\overline{T_2;}\\ \underline{T_1;}\end{array} \qquad \begin{array}{l}\text{correct when:}\\ \begin{cases}\text{notRO}(\Delta_1)\cap\Delta_2=\emptyset\\ \text{notRO}(\Delta_2)\cap\Delta_1=\emptyset\end{cases}\end{array}$$

The Concept of Span. Notice that in the previous criteria, T_1 and T_2 are possibly empty *spans* of instructions. It is natural to query resource contexts around a span, and to query the resource usage of a span by collapsing the usages of its instructions:

$$\underline{\overline{\Gamma_1\ T;\Delta\ \Gamma_2}} \quad\equiv\quad \underline{\overline{\Gamma_1\ t^1;\Delta^1\ldots t^n;\Delta^n\ \Gamma_2}} \qquad \text{where } \Delta\equiv\Delta^1;\ldots;\Delta^n$$

Swapping multiple instructions by collapsing their usages is equivalent to iteratively swapping pairs of instructions. However, it is algorithmically cheaper, requiring $n+m+2$ operations of linear cost on Δs (;, $\cap$), instead of $n\times m\times 2$ operations. From now on, we use spans when describing transformations.

Inserting Instructions. The `Instr.insert` transformation.

Deleting Instructions. The `Instr.delete` transformation.

6.2 Loop Transformations

Tiling Loops. The `Loop.tile` transformation.

$$\begin{array}{l}\hline \textbf{for}_{\gamma}\ i\ \textbf{in}\ r\ \{\\ \quad T;\\ \}\\ \hline\end{array} \longmapsto \begin{array}{l}\hline G_1;\\ \textbf{for}_{\gamma_o}\ i_o\ \textbf{in}\ r_o\ \{\\ \quad \textbf{for}_{\gamma_i}\ i_i\ \textbf{in}\ r_i\ \{\\ \qquad \mathrm{Subst}\{i := \mathrm{new\_i}(i_o, i_i)\}(T);\\ \quad \}\\ \}\\ G_2;\\ \hline\end{array}$$

always correct:

$$G_1 : \mathop{\bigstar}\limits_{i\in r} \gamma.\mathrm{excl.pre} \Rightarrow \mathop{\bigstar}\limits_{i_o\in r_o}\mathop{\bigstar}\limits_{i_i\in r_i} \gamma.\mathrm{excl.pre}$$

$$G_2 : \mathop{\bigstar}\limits_{i_o\in r_o}\mathop{\bigstar}\limits_{i_i\in r_i} \gamma.\mathrm{excl.post} \Rightarrow \mathop{\bigstar}\limits_{i\in r} \gamma.\mathrm{excl.post}$$

$$\gamma_i \equiv \mathrm{Subst}\{i := \mathrm{new\_i}(i_o, i_i)\}(\gamma)$$

$$\gamma_o \equiv \begin{cases}\mathrm{vars} \equiv \gamma.\mathrm{vars}\\ \mathrm{shrd} \equiv \mathrm{Subst}\{i := \mathrm{new\_i}(i_o, r_i.\mathrm{start})\}(\gamma.\mathrm{shrd})\\ \mathrm{excl} \equiv \bigstar_{i_i\in r_i} \gamma_i.\mathrm{excl}\end{cases}$$

Example Ranges. Correct when tile size divides (or min etc) ghost calls: ghost tile divides + ghost tile undivides

$$\begin{aligned} r &= 0..(n\times m)\\ r_o &= 0..n\\ r_i &= 0..m\\ \mathrm{new\_i}(i_o, i_i) &= i_o * m + i_i\end{aligned}$$

Interchanging Loops. The `Loop.swap` transformation.
correct when both loops are parallelisable correct when only outer loop is parallelisable

$$\begin{array}{l}\hline \textbf{for}_{\gamma_i}\ i\ \textbf{in}\ r_i\ \{\\ \quad \textbf{for}_{\gamma_j}\ j\ \textbf{in}\ r_j\ \{\\ \qquad T;\\ \quad \}\\ \}\\ \hline\end{array} \longmapsto \begin{array}{l}\hline G_1;\\ \textbf{for}_{\gamma'_j}\ j\ \textbf{in}\ r_j\ \{\\ \quad \textbf{for}_{\gamma'_i}\ i\ \textbf{in}\ r_i\ \{\\ \qquad T;\\ \quad \}\\ \}\\ G_2;\\ \hline\end{array} \quad \begin{array}{l}\text{corect when:}\\ \qquad \gamma_i.\mathrm{shrd.modifies.linear} = \emptyset\\ \text{with:}\end{array}$$

ghost swap group

Fissioning Loops. The `Loop.fission` transformation.
cleanup efracs and local vars

$$\begin{array}{l}\hline \textbf{for}_{\ \gamma}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_1;\ \Delta_1\\ \quad \Gamma\\ \quad T_2;\ \Delta_2\\ \}\\ \hline\end{array} \longmapsto \begin{array}{l}\hline \textbf{for}_{\ \gamma_1}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_1;\\ \}\\ \textbf{for}_{\ \gamma_2}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_2;\\ \}\\ \hline\end{array}$$

correct when:

$$\begin{cases} i \text{ not free in } \gamma.\text{shrd} \\ \text{notRO}(\Delta_1) \cap \Delta_2 \cap \gamma.\text{shrd.modifies} = \emptyset \\ \text{notRO}(\Delta_2) \cap \Delta_1 \cap \gamma.\text{shrd.modifies} = \emptyset \end{cases}$$

with:

$$R \equiv \text{cleanup}(\Gamma - \gamma.\text{shrd})$$

$$\gamma_1 \equiv \begin{cases} \text{vars} \equiv \gamma.\text{vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_1 \\ \text{excl.pre} \equiv \gamma.\text{excl.pre} \\ \text{excl.post} \equiv R \end{cases}$$

$$\gamma_2 \equiv \begin{cases} \text{vars} \equiv \gamma.\text{vars} \\ \text{shrd} \equiv \gamma.\text{shrd} \cap \Delta_2 \\ \text{excl.pre} \equiv R \\ \text{excl.post} \equiv \gamma.\text{excl.post} \end{cases}$$

Fusing Loops. The `Loop.fusion` transformation.

$$\begin{array}{l}\hline \textbf{for}_{\ \gamma_1}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_1;\\ \}\ \Delta_1\\ \textbf{for}_{\ \gamma_2}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_2;\\ \}\ \Delta_2\\ \hline\end{array} \longmapsto \begin{array}{l}\hline \textbf{for}_{\ \gamma}\ i\ \textbf{in}\ r_i\ \{\\ \quad T_1;\\ \quad T_2;\\ \}\\ \hline\end{array}$$

correct when:

$$\begin{cases} i \text{ fresh in } \gamma_1.\text{shrd and } \gamma_2.\text{shrd} \\ \text{notRO}(\Delta_1) \cap \Delta_2 \cap \gamma_1.\text{shrd} = \emptyset \\ \text{notRO}(\Delta_2) \cap \Delta_1 \cap \gamma_2.\text{shrd} = \emptyset \end{cases}$$

with:

$$R, Q_1, P_2 \equiv \gamma_1.\text{excl.post} -^{!} \gamma_2.\text{excl.pre}$$

$$\gamma \equiv \begin{cases} \text{vars} \equiv \gamma_1.\text{vars} \cup \gamma_2.\text{vars} \\ \text{shrd} \equiv \gamma_1.\text{shrd} \star \gamma_2.\text{shrd} \\ \text{excl.pre} \equiv \gamma_1.\text{excl.pre} \star P_2 \\ \text{excl.post} \equiv \gamma_2.\text{excl.post} \star Q_1 \end{cases}$$

Hoisting an Allocation. The `Loop.hoist` transformation allows to hoist an allocation outside of a loop.

$$\begin{array}{l} \hline \textbf{for}_{\gamma}\ i\ \textbf{in}\ r_i\ \{ \\ \quad \textbf{let}\ x = \mathsf{MALLOC}(ds,\ \tau); \\ \quad T; \\ \quad \mathsf{MFREE}(ds,\ x); \\ \} \\ \hline \end{array} \quad\longmapsto\quad \begin{array}{l} \hline \textbf{let}\ x_i = \mathsf{MALLOC}(r_i :: ds,\ \tau); \\ \textbf{for}_{\gamma'}\ i\ \textbf{in}\ r_i\ \{ \\ \quad \mathsf{Subst}\{x := t_x\}(T); \\ \} \\ \mathsf{MFREE}(r_i :: ds,\ x); \\ \hline \end{array}$$

correct when i is not free in ds.

$$t_x \equiv x_i[\mathsf{MINDEX}(r_i :: ds,\ i :: 0^{ds})]$$

$$\gamma' \equiv \begin{cases} \mathsf{vars} \equiv \gamma.\mathsf{vars} \\ \mathsf{shrd} \equiv \gamma.\mathsf{shrd} \\ \mathsf{excl} \equiv \gamma.\mathsf{excl} \star \mathsf{Uninit}(t_x \rightsquigarrow \mathsf{Cell}) \end{cases}$$

Hoisting an Instruction. The `Loop.move_out` transformation allows to hoist an instruction outside of a loop.

correct when:

- i fresh in T_1 (same code for every iteration)
- T_1 does not self interfere ($T_1; T_1 \leftrightarrow T_1$), i.e.

$$\begin{cases} \Gamma_2[\mathsf{produced}(\Delta_1)] - \Gamma_1[\mathsf{Uninit}(\Delta_1)] = \emptyset \\ \mathsf{full}(\Delta_1) = \emptyset \end{cases}$$

- T_2 does not alter the effects of T_1 ($T_1; T_2; T_1 \leftrightarrow T_1; T_2$), i.e:

$$\Delta_1 \cap \mathsf{notRO}(\Delta_2) = \emptyset$$

- if r_i was the empty range, doing T_1 once has no observable effect:

$$G_3 : \Gamma_2[\mathsf{produced}(\Delta_1)] \Rrightarrow \mathsf{Uninit}(\Gamma_2[\mathsf{produced}(\Delta_1)])$$

The code should typecheck with G_3, however G_3 is erased from the final result.

with:

$$\begin{array}{l} \hline \textbf{for}_{\gamma}\ i\ \textbf{in}\ r_i\ \{ \\ \quad \Gamma_1\ T_1; \Delta_1 \\ \quad \Gamma_2\ T_2; \Delta_2 \\ \} \\ \hline \end{array} \quad\longmapsto\quad \begin{array}{l} \hline T_1; \\ \textbf{for}_{\gamma'}\ i\ \textbf{in}\ r_i\ \{ \\ \quad T_2; \\ \} \\ G_3; \\ \hline \end{array} \qquad \gamma' \equiv \begin{cases} \mathsf{vars} \equiv \gamma.\mathsf{vars} \\ \mathsf{shrd.modifies} \equiv \\ \quad \Gamma_2 - \gamma.\mathsf{excl.pre} - \gamma.\mathsf{shrd.reads} \\ \mathsf{shrd.reads} \equiv \gamma.\mathsf{shrd.reads} \\ \mathsf{excl} \equiv \gamma.\mathsf{excl} \end{cases}$$

Also works if t_1 and t_2 are blocks of instructions.

Shifting a loop range. The `Loop.shift` transformation.

$$\begin{array}{l} \hline \textbf{for}_{\gamma}\ i\ \textbf{in}\ r\ \{ \\ \quad T; \\ \} \\ \hline \end{array} \quad \longmapsto \quad \begin{array}{l} \hline G_1 \\ \textbf{for}_{\gamma'}\ i'\ \textbf{in}\ r'\ \{ \\ \quad \text{Subst}\{i := f(i')\}(T); \\ \} \\ G_2 \\ \hline \end{array}$$

correct when $f(i')$ and r' are convertible to formulas.
with:

$$G_1 \equiv \bigstar_{i \in r} \gamma.\text{excl.pre} \Rightarrow \bigstar_{i' \in r'} \text{Subst}\{i := f(i')\}(\gamma.\text{excl.pre})$$

$$G_2 \equiv \bigstar_{i' \in r'} \text{Subst}\{i := f(i')\}(\gamma.\text{excl.post}) \Rightarrow \bigstar_{i \in r} \gamma.\text{excl.post}$$

$$\gamma' \equiv \text{Subst}\{i := f(i')\}(\gamma)$$

Sliding a loop. The `Loop.slide` transformation.
similar to Loop.tile + Sequence/Instr.insert/delete/Loop.move_out
correct when:

- loop is parallelizable: $\gamma.\text{shrd.modifies} = \emptyset$
- T does not self interfere ($T; T \leftrightarrow T$)

$$\begin{array}{l} \hline \textbf{for}_{\gamma}\ i\ \textbf{in}\ r\ \{ \\ \quad T; \\ \} \\ \hline \end{array} \quad \longmapsto \quad \begin{array}{l} \hline G_1; \\ \textbf{for}_{\gamma_o}\ i_o\ \textbf{in}\ r_o\ \{ \\ \quad \textbf{for}_{\gamma_i}\ i_i\ \textbf{in}\ r_i\ \{ \\ \quad\quad \text{Subst}\{i := \text{new\_i}(i_o, i_i)\}(T); \\ \quad \} \\ \} \\ G_2; \\ \hline \end{array}$$

$$G_1 : \bigstar_{i \in r} \gamma.\text{shrd.pre} \Rightarrow \bigstar_{i_o \in r_o} \bigstar_{i_i \in r_i} \gamma.\text{shrd.pre}$$

$$G_2 : \bigstar_{i_o \in r_o} \bigstar_{i_i \in r_i} \gamma.\text{shrd.post} \Rightarrow \bigstar_{i \in r} \gamma.\text{shrd.post}$$

$$\gamma_i \equiv \text{Subst}\{i := \text{new\_i}(i_o, i_i)\}(\gamma)$$

$$\gamma_o \equiv \begin{cases} \text{vars} \equiv \gamma.\text{vars} \\ \text{shrd.reads} \equiv \gamma.\text{shrd.reads} \\ \text{FIXME: asymmetric prefix:} \\ \text{shrd.modifies} \equiv \bigstar_{i \in r} \gamma.\text{shrd} \\ \text{excl} \equiv \emptyset \end{cases}$$

Loop.unroll. Unrolling a loop over a constant range is always safe:

$$\textbf{for}\ i\ \textbf{in}\ r\ \{T;\} \leftrightarrow \{T[i := r.\text{start}]\}; \ldots; \{T[i := r.\text{last}]\};$$

Loop.delete Loop.extend_range

6.3 Variable Transformations

The `Variable.local_name` transformation.

$$\underline{\overline{\Gamma_1\ T; \Gamma_2}} \quad \longmapsto \quad \begin{array}{l} \hline \textbf{let } x' = \textbf{new}(\bot); \\ T_1; \\ G_1; \\ \text{Subst}\{x := x'\}(T); \\ G_2; \\ T_2; \\ \hline \end{array}$$

correct when the following typechecks (erased from the final result):

$$G_1 \equiv M(x \rightsquigarrow \mathsf{Cell}) \Rightarrow H \star (H --^{*} M(x \rightsquigarrow \mathsf{Cell}))$$
$$G_2 \equiv H \star (H --^{*} M(x \rightsquigarrow \mathsf{Cell})) \Rightarrow M(x \rightsquigarrow \mathsf{Cell})$$

with:

$$M(x \rightsquigarrow \mathsf{Cell}) \in \Gamma_1$$
$$\begin{cases} T_1 \equiv \mathsf{set}(x', \mathsf{get}(x)); & \text{if } \Gamma_1 \Rightarrow \mathsf{RO}(x \rightsquigarrow \mathsf{Cell}) \\ T_1 \equiv \emptyset & \text{otherwise} \end{cases}$$
$$\begin{cases} T_2 \equiv \mathsf{set}(x, \mathsf{get}(x')); & \text{if } \Gamma_2 \Rightarrow \mathsf{RO}(x \rightsquigarrow \mathsf{Cell}) \\ T_2 \equiv \emptyset & \text{otherwise} \end{cases}$$

It is also possible to remove T_1 / T_2 in other cases, but not necessary for correctness, this ensure typechecking.

7 RELATED WORK

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

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 feedback. Second, general-purpose compilers relies on fully-automated procedures, often guided by heuristics, to determine what transformations to apply. In contrast, OptiTrust transformations are fully controlled by the programmer, either directly via basic transformations, or indirectly via combined transformations. Third, general-purpose compilers rely on static analysis applied to plain C code to determine whether certain transformations are applicable, and as a result may lack information to trigger a transformation. In 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.

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