Université de Strasbourg

Université de Strasbourg

École doctorale Mathématiques, sciences de l'information et de l'ingénieur | ED 269 | Université de Strasbourg

École Doctorale Mathématiques, Sciences de l'Information et de l'Ingénieur

ICube : Laboratoire des sciences de l'ingénieur, de l'informatique et de l'imagerie

THÈSE présentée par

Guillaume Bertholon

soutenue le : **22 septembre 2025**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**
Discipline : Informatique

Interactive compilation via trustworthy source-to-source transformations

Compilation interactive par des transformations source-à-source dignes de confiance

---

Thèse dirigée par :

Arthur Charguéraud	Directeur de recherche, Inria Strasbourg

Rapporteurs :

David Monniaux	Directeur de recherche, CNRS (Université de Grenoble)
Stephan Merz	Directeur de recherche, Inria Nancy

Autres membres du jury :

Chantal Keller	Maitresse de conférences, Université Paris-Saclay
Nicolas Magaud	Professeur des universités, Université de Strasbourg
Pierre-Évariste Dagand	Chargé de recherche, CNRS (Université Paris Cité)

Remerciements

En premier lieu, je tiens très sincèrement à remercier Arthur Charguéraud pour avoir encadré ma thèse. Arthur a été très présent, et particulièrement investi dans son accompagnement lors de ce doctorat, toujours disponible pour répondre à mes nombreuses questions qu'elles portent sur des détails techniques ou sur la culture de notre communauté scientifique. Il a eu à cœur de me partager ses méthodes de travail, de m'expliquer patiemment pourquoi mes premières tentatives de rédactions étaient peu compréhensibles et de faire vivre et rayonner le projet OptiTrust sur lequel nous avons travaillé ensemble avec son optimisme résistant à toute épreuve.

Merci ensuite à tous les membres du jury de cette thèse. C'est un honneur pour moi que vous ayez tous accepté d'en faire partie. Je suis tout particulièrement reconnaissant envers mes deux rapporteurs, Stephen Merz et David Monniaux, qui m'ont fourni dans leurs commentaires un retour essentiel et ont grandement contribué à la qualité de ce manuscrit. Merci à Pierre-Évariste Dagand pour m'avoir aiguillé vers cette thèse en me présentant à Arthur, il y a 4 ans, ainsi que pour ses questions pertinentes et imagées. Merci à Chantal Keller pour avoir accepté de rester membre du jury en visioconférence malgré une jambe cassée. Merci enfin à Nicolas Magaud, grâce à qui j'ai pu découvrir les joies de l'enseignement avec nos élèves de master du département Math-Info en parallèle de ma thèse.

Je remercie tous ceux qui ont travaillé sur le projet OptiTrust, avec qui j'ai pu collaborer. Je commencerai par Thomas Kœhler avec qui j'ai pu développer une partie des transformations de code présentées dans ce manuscrit. Merci à toi d'avoir apporté ton point de vue toujours pertinent sur le projet et d'avoir su nous aider à peser le pour et le contre, avec justesse, sur les choix techniques. Je profite de ces remerciements pour renouveler mes félicitations pour ton poste au CNRS ! Merci également à vous Yanni, Pauline, Élian et Valeran de continuer à faire avancer le projet à vos manières respectives pendant vos stages et bientôt thèses pour certains. Je regrette d'avoir été occupé par la rédaction de ce manuscrit et de ne pas avoir pu collaborer davantage avec vous. J'ai hâte de voir comment vous allez faire évoluer le projet OptiTrust.

Durant cette thèse, j'ai aussi eu la chance de faire partie de l'équipe que l'on nomme au choix ICPS ou bien Camus. Vous avez tous été très accueillants et toujours sympathiques. Merci à vous les membres permanents : Jens, Stéphane, Vincent, Phillipe, Béranger, Alain et Cédric. Vous m'avez chacun apporté votre lot de discussions très enrichissantes. Bien évidemment, merci à vous les doctorants, post-doctorants et ingénieurs de recherche : Clément R., Clément F., Raphaël, Atoli, Erwan, Marek, Ugo, Tom, Haifa, Etienne et Arun. Je n'oublierai pas ces repas et moments partagés entre faits divers improbables et discussions dans la bonne humeur. Merci également aux assistantes d'équipe Inria : Ouiza et Marine. Votre accompagnement administratif combiné à votre investissement et participation à la vie du centre m'ont été très précieux. Merci à toute l'équipe pour avoir créé cette très bonne ambiance que ce soit au labo tout au long de la thèse ou à l'extérieur pendant les soirées raclettes ou la sortie canoë !

Un grand merci ensuite à l'équipe Inria Cambium de m'avoir accueilli à leurs séminaires d'équipe toujours très instructifs, et de m'avoir laissé occuper un bureau chez eux. Tout particulièrement merci à François, Didier, Yannick, Alexandre et Xavier qui m'ont apporté, avec recul, leurs conseils avisés.

Je ne serai pas là sans tous ceux qui m'ont permis de choisir la voie du doctorat. En cela, je suis notamment reconnaissant à l'ensemble des professeurs de l'ENS et du MPRI qui m'ont guidé à travers leurs enseignements vers cette thèse. En particulier, je me dois de remercier Timothy Bourke pour son accompagnement dans mes recherches de stages en tant que tuteur.

Merci ensuite à tous mes amis qui m'ont toujours soutenu dans mon parcours avec bienveillance. Merci pour tous ces projets qu'ils soient musicaux ou basés sur du game design, pour ces parties de jeux de société, ou pour tous ces moments de joie partagés. Vous êtes tous incroyables !

Merci à ma famille qui m'a inculqué l'envie de bien faire dans mes études et la curiosité scientifique qui m'a amené jusque ici. Pour finir, un énorme merci à Masha, mon épouse, pour avoir supporté mes allers-retours Paris-Strasbourg tout au long de cette thèse, pour avoir accepté la rédaction de mon manuscrit juste avant la phase finale de préparation de notre mariage, mais surtout pour son soutien inconditionnel et extrêmement précieux à travers les épreuves de la vie, depuis que nous sommes ensemble.

Contents

1 Introduction

In computer science, programmers are usually facing two limiting factors.

First, programmers are often limited by the computing power of their hardware. This is, for instance, usually the case with applications in the domains of numerical simulations, machine learning and computer graphics. Programmers can push this hardware limit further by optimizing their programs.

Second, and maybe even more crucially, programmers are limited by their ability to produce code that actually computes what they want. In order to trust their programs, programmers need to eliminate the possibility of incorrect outputs (a.k.a. bugs). Unfortunately, testing the program on a few examples is generally not sufficient to avoid those incorrect outputs. Formal verification can go further by proving that the program will always respect a specification, that is a mathematical description of what it should compute. Ensuring that a program computes what the programmer wants becomes more complex as the size of the program grows.

These two limiting factors are interdependent as, most of the time, optimizing a program also makes such program more complex. Therefore, the more effort one puts into limiting the resources needed to execute a program, the more efforts are also required to ensure that such program does not have bugs.

In this PhD, our objective is to reduce the amount of work needed to produce code that is at the same time optimized and exempt from bugs.

1.1 Interactive compilation for program optimization

Retrospective on hardware performance Historically, hardware innovations were the simplest way of making computation faster. In the 1970s, hardware manufacturers were able to quickly increase the number of operations a processor can do per second or per consumed joule. This was possible thanks to the miniaturization of transistors, that lead to an exponential growth of the processor clock frequency year after year, while keeping the power consumption of the whole processor identical. For software developers, this meant that their program could run faster (and consume less energy) with a more recent hardware without any additional work. Since the beginning of the millennium, we cannot hope for a significant growth of the clock frequencies anymore as the hardware size is slowly reaching atomic limits.

Moreover, processor clock frequency is not the only bottleneck when performing computations nowadays. Indeed, when computations are fast, the speed of memory accesses can often be the limiting factor. Typically, in 2025, fetching a value from the RAM[1] can take more than 100 cycles, while a typical processor can compute multiple FMA[2] instructions in one cycle. This is not surprising since, even at the speed of light, one processor cycle is too short for data to travel back and forth between the processor and the RAM. Similarly, energy-wise, getting 32 bits of data from the RAM consumes 640 pJ, while computing an FMA on 32-bits floats only consumes 1.2 pJ [Dal21].

1: Random Access Memory: the main memory region for storing intermediate results of ongoing computations

2: Fused Multiply and Add: this corresponds to the operation $x * y + z$ where x, y and z are floating point values.

[Dal21]: Dally (2021), *The Future of Computing: Domain-Specific Architecture*

Both the end of the exponential growth of the clock frequency and the daunting cost of memory accesses have lead to increasingly complex hardware to keep reducing computation costs [Rag24].

[Rag24]: Ragan-Kelley (2024), *The Future of Fast Code: Giving Hardware What It Wants*

Manufacturers find it incredibly difficult to accelerate one processor core any further, instead they equip CPUs[3] with multiple cores (typically between 4 and 8 in an average computer), each executing different instructions in parallel. Then, on each core, modern architectures also allow performing the same computation on multiple chunks of data with SIMD[4] instructions.

3: Central processing unit: the main processor inside a computer, typically running the operating system and most applications

4: Single Instruction Multiple Data

To partially address the slow memory issue, modern computer architectures include *memory caches*. If all the data cannot be stored close enough to the CPU to benefit from fast accesses, one can instead store the copy of a small subset of that data closer, without taking too much physical space. Such a copy is what we call a memory cache. Hardware manufacturers find the pattern so effective that they place memory caches on memory caches, typically creating up to 3 levels of caches between the RAM and the CPU. A cache near the RAM stores more data, but a cache near the CPU have a faster access time.

Modern CPUs also include features such as branch prediction and out of order execution with an increasingly complex microarchitecture that we will not detail here.

In addition to all of that, nowadays, computers usually embed different kind of processing units called *accelerators* in addition to the CPU. Accelerators can have very different architectures, each of them suited for specific applications, where they perform better than the general-purpose CPU. The most widely used accelerator is the GPU[5]. GPUs are processors tailored for maximizing SIMD efficiency but are inefficient with complex control flow and pointer indirections. More recently, the rise of machine learning gave birth to new kind of accelerators: NPU[6]s are processors specialized for matrix product and convolutions and TPU[7]s mix memory and computation in the same chip to compute tensors. On another level, FPGA[8]s allow programmers to make their own custom accelerator by configuring series of electronic circuit logical gates.

5: Graphical Processing Unit: Historically the SIMD architecture of the GPUs was mostly used for computer graphics to execute programs called shaders on each vertex or pixel, hence the name. Nowadays, GPUs are used for all sorts of massively parallel computations.

6: Neural Processing Unit

7: Tensor Processing Unit

8: Field-Programmable Gate Array: the most widely used technology for reconfigurable electronic circuits

Each of those architectures have their own optimization constraints. Therefore, programming high performance code for a system that combines different kind of accelerators and CPUs can be particularly complex. In this PhD, we only focused on optimizing computations performed on CPUs, which is already a significantly challenging task. That said, we believe that some optimization techniques for CPUs can be reused in presence of accelerators. Even when working with only one kind of processor, the program performing the best depends on the exact targeted hardware configuration. Hence, performance critical code is not portable and programmers need to adapt such code for each specific machine they want to use which incurs a potentially significant maintenance cost.

The challenge of performant code for complex hardware We see that our complex CPUs need carefully chosen instructions to perform best. However, nowadays, very few programmers are directly writing their code in assembly to choose those instructions by hand. Indeed, most programmers rely on programming languages that are further away from the machine, and then execute a compiler that chooses the exact instruction sequence sent to the CPU. On arbitrary imperative code, fully automatic compilers fail to produce the fastest code for a given algorithm. Even starting from a programming language relatively close to the machine such as C, the exploration space is very large, and cost models are very complex [Vac+03].

[Vac+03]: Vachharajani et al. (2003), *Compiler Optimization-Space Exploration*

Those general-purpose compilers such as GCC or Clang must rely on heuristics that greatly improve the average case but produce an output that is nowhere near the best performance achievable and is sometimes not enough [BI19]. Therefore, producing highly performant code is out of reach for a general-purpose and fully automatic compiler.

[BI19]: Barham et al. (2019), *Machine Learning Systems are Stuck in a Rut*

A common practice in the high performance programming community is to sacrifice automation and write a micro-optimized version of their C code by hand in order to guide the compiler in its choices [Ama+20; Eva+22]. This practice generally brings significant speedups that can typically make the same algorithm run 50 times faster [KK22]. However, manually micro-optimized code is much more complex than the corresponding naive implementation of the same algorithm. This complexity is due to the fact that a lot of abstractions cannot be kept in the optimized code, and that simple computations can be hidden beneath complex instructions. Besides, optimized code tends to be longer than a naive version of the same algorithm. This added complexity makes writing by hand an optimized version of an algorithm not satisfying for at least three reasons:

[Ama+20]: Amaral et al. (2020), *Programming languages for data-Intensive HPC applications: A systematic mapping study*
[Eva+22]: Evans et al. (2022), *A survey of software implementations used by application codes in the Exascale Computing Project*
[KK22]: Kelefouras et al. (2022), *Design and Implementation of 2D Convolution on x86/x64 Processors*

- First, writing manually optimized code is a very time-consuming process that requires advanced knowledge on how computers work. It can take experts months to optimize a single application [Bar18].
- Second, the resulting optimized code is significantly harder to maintain than the naive algorithm implementation. This added maintenance complexity is particularly unfortunate since those optimizations are not portable and the code might need to be adapted to new hardware.
- Third, there are a lot of opportunities to introduce bugs in complex code. Moreover, the added complexity makes such bugs harder to track down, especially when concurrency is involved.

[Bar18]: Barsamian (2018), *Pic-Vert: A Particle-in-Cell Implementation for Multi-Core Architectures*

Interactive compilers An alternative to rewriting the code by hand is to exploit an interactive approach, whereby the user to collaborates with an *interactive compiler* to optimize the code. In this case, programmers first write a naive and easy to read version of their algorithm, then write a second kind of source code to guide the compiler (thus replacing some heuristics with more carefully chosen strategies) and produce the high performance code. The main challenge with this approach is to define a model of interaction between the user and the compiler explaining how programmers describe what they want and what kind of feedback they receive.

In the subdomain of image processing, interactive compilers such as Halide [Rag+13] or TVM [Che+18] separate the description of the functional behavior of an algorithm, and the *scheduling* of the computation. That scheduling determines the order of computations and the memory layout. This separation allows testing multiple schedules in order to keep the best performing one without risking changing the semantics of the code in the process. In order to improve the feedback loop, tools such as Roly-poly [Ika+21] can create interactive visualizations to help in the conception of Halide schedules.

[Rag+13]: Ragan-Kelley et al. (2013), *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines*
[Che+18]: Chen et al. (2018), *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*
[Ika+21]: Ikarashi et al. (2021), *Guided Optimization for Image Processing Pipelines*

Some interactive compilers such as Elevate [Hag+20b] and Exo [Ika+22] replace schedules with *source-to-source transformation scripts*. In that case, instead of giving instructions on how to lower functional code into an efficient implementation in one pass, the user progressively transforms an executable code by applying rewriting rules. This point of view leads to the following user interaction loop: at each step, the tool can display the current version of the source code and ask the user which transformation to apply

[Hag+20b]: Hagedorn et al. (2020), *Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies*
[Ika+22]: Ikarashi et al. (2022), *Exocompilation for productive programming of hardware accelerators*

next and where. Generally, source-to-source transformation scripts give more control and feedback to the user compared to monolithic schedules.

Existing interactive compilers (which we describe in more details in section 2.4.1) all have their own domains of applicability and their own strengths and weaknesses. In this PhD, we try to build an interactive compiler that accepts imperative code as input, and gives full control to the user to apply a sequence of source-to-source transformations.

Interactive compilers should not accept code transformations that could create bugs in the software. In other words, we say that interactive compilers should guarantee transformation *correctness*. One focus of this PhD is the experimentation around different ways to ensure such transformation correctness, and the impact of such choices on the trustworthiness of the compiler.

1.2 Correctness for interactive compilers

One way of ensuring the correctness of code transformations, is to guarantee that all transformations preserve the semantics of the code. We say that a transformation is *semantic-preserving* when the possible behaviors of the produced code are included in the possible behaviors of the original code.

Some code transformations are always correct with respect to the semantics of the language. For instance, for an integer variable x the simplification of $x + 1 - 1$ into x is always correct[9]. However, a lot of useful transformations are not valid in the general case, but are still applicable on a particular instance due to properties enforced by the algorithm at hand. For example, a transformation replacing $(x/2) \times 2$ into x where x is an integer variable does not preserve the semantics in general (as for instance if $x = 3$ then $(x/2) \times 2$ computes to 2 instead of 3), but this transformation is semantic-preserving under the hypothesis that x is even before the evaluation of $(x/2) \times 2$. Dually, a transformation that removes an assignment to a variable might be correct because such value is never read afterwards and does not contribute to the program output. In this case, the transformation does not locally preserve the semantics of the deleted instruction but globally preserves the expected behavior of the initial code.

9: Note that, if we assume the standard wrapping behavior of addition and subtraction on fixed size integers, this is true even when the expression $x + 1$ overflows. In languages such as C, signed integers do not wrap but trigger undefined behavior instead. This is not an issue either because replacing an undefined behavior with any fixed behavior is always correct.

These properties that describe either the hypotheses before executing an instruction in the code or the expected behavior of such instructions are called *program invariants*. The two previous examples show that it is necessary to exploit those program invariants to check that the transformations requested by the user are indeed applicable. However, usually, finding relevant invariants and checking that they hold is a complex task.

One classical way to tackle this problem is to rely on static analysis to synthesize the relevant invariants. Static analyses are available for specific use-cases, such as basic linear algebra [JM18], but there is no hope to derive automatically the required invariants for any code and for any kind of transformation, as this task in undecidable in general. Such limitation implies that we need to sometimes rely on the user to help the interactive compiler to find and check those invariants.

[JM18]: Journault et al. (2018), *Inferring functional properties of matrix manipulating programs by abstract interpretation*

That said, our goal is to build an interactive compilation framework that performs user-guided transformations leveraging program invariants to guarantee correctness. We believe that such a tool can allow writing code that matches the performance of manually micro-optimized code, without the extra risks and efforts of the manual approach. Building this framework raises several fundamental questions:

- How can we describe the invariants that are needed to justify the correctness of transformations?
- What kind of annotations need to be provided by the user to deduce those invariants anywhere in the code? How are these annotations algorithmically checked and propagated?
- How can we leverage the invariants of the code in the implementation of transformations to justify their correctness?
- In order to perform chains of transformations, we need to maintain the capability of computing invariants between each step. How can we implement transformations in a way such that they maintain meaningful annotations?
- How can we gain confidence in the fact that the implementation of the transformations does not contain bugs that would otherwise compromise the correctness of the output code?

1.3 Description of invariants

As said in previous section, program invariants are properties that either describe hypotheses or expected behaviors of program instructions, and we need such properties to reason about the correctness of transformations.

Hoare triples The field of program logics formalizes this notion of program invariant with the concepts of *precondition*, *postcondition* and *Hoare triple*. A precondition is a mathematical formula describing the hypotheses that one can assume true before the execution of some code. Dually, the postcondition is a mathematical formula describing hypotheses that must be true after the execution of some code. Usually, for the same code a more precise precondition can lead to a more precise postcondition. In this setup, a Hoare triple denoted $\{H\}\ t\ \{Q\}$ where H is a precondition, t is a program and Q a postcondition, describes the fact that in an input state described by the precondition H, the (sub)program t terminates in an output state described by the postcondition Q[10].

For a function f with one argument, a property of the form $\forall x,\ \{H\}\ f(x)\ \{Q\}$ can be viewed as a contract fulfilled by f. More precisely, if before a call to $f(x)$ the precondition H holds, then after such call the postcondition Q must hold as well. This vision from outside on Hoare triples can be generalized to any instruction, and is very useful in practice to abstract away entire subprograms when reasoning about transformations.

10: For some program logics, Hoare triples do not guarantee termination. In such cases, if H holds, either t does not terminate or the output state is described by Q. We do not consider this alternative definition in the rest of this manuscript.

Separation logic as invariant structure The definition of Hoare triples does not state what kind of properties are expressed as pre- and postconditions. In theory, an invariant can specify anything about the values of variables currently in scope and about the state of the entire program memory. In practice, however, being able to express arbitrary properties about the entire memory is not desirable. Indeed, suppose that an invariant is verified before an instruction that mutates a memory cell. How can one tell if this invariant still holds after the execution of the instruction? If such invariant does not hold, what can still be asserted about the memory after the execution? The answer to these two questions depends on the way the invariant is expressed. Thus, to avoid arbitrarily complicated reasoning steps to show invariant preservation when instructions modify the memory, we want to define a fixed structure for invariants that refer to mutable memory.

[Rey02]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

One such structure is introduced by separation logic [Rey02] with the key notion of *separating conjunction*. The separating conjunction of two propositions, denoted $H_1 \star H_2$, states that both properties H_1 and H_2 hold on disjoint parts of the memory. This simple, yet very powerful, notion enables modular reasoning about the memory, thanks to what is called the *frame rule*.

$$\frac{\{H\}\ t\ \{Q\}}{\{H \star H'\}\ t\ \{Q \star H'\}}$$

Figure 1.1: Frame rule in separation logic. Properties above the line are hypotheses and the property below is a conclusion.

Intuitively, if an instruction modifies a single memory cell, then any property about memory that is separated from this memory cell is necessarily preserved. More formally, the frame rule states that if a Hoare triple $\{H\}\ t\ \{Q\}$ holds, for any property H', the triple $\{H \star H'\}\ t\ \{Q \star H'\}$ also holds[11]. To link this statement with the intuition, one may think of H as the description of the memory cells modified by t before such modification, of Q as their description after the modification, and of H' as the description of the rest of the memory that is not manipulated by t.

11: In the case where H' describes a fragment of memory that is not disjoint with H, it is impossible to satisfy $H \star H'$ with any memory, thus the triple is useless but true anyway.

With separation logic, one can view each invariant as a set of resources available at a given program point. Some resources are independent of the memory state and only depend on the local context of immutable variables. We call those resources *pure*, and we can freely duplicate them because they remain true no matter which instruction is executed. The rest of the resources are called *linear*, and each of them describe a disjoint non-empty part of the memory. With this point of view, each instruction can consume and produce a set of linear resources, and similarly each instruction can depend on and produce a set of pure resources. Such manipulations describe the difference between the invariant before such instruction and the invariant after.

Concurrent separation logic Separation logic also simplifies reasoning about *concurrent* programs, that is, programs with several parallel execution threads acting on the same memory[12]. Reasoning about such concurrent programs can be hard because threads are executing instructions in a non-deterministic order, and the side effects of one thread can impact another thread's computations. In order to avoid such non-deterministic interferences, one may force all instructions executing concurrently to manipulate disjoint linear resources. However, by itself, forcing disjoint resources for each concurrent thread would not be very powerful. Indeed, there are common safe concurrent program patterns that involve some amount of resource sharing. One of such pattern, is reading at the same memory location in multiple different threads, provided no other thread concurrently writes at this location.

12: In this manuscript, we will not consider the case of weak memory models, as sequentially consistent concurrency is already challenging enough.

To allow this kind of resource sharing, a standard extension to separation logic allows cutting a resource in two halves, that give reduced permissions until they are merged back together. In that case, each half is considered disjoint for the separating conjunction, and can be split further. Typically, read-only fractions allow making multiple separated resources describing the same memory location, by temporarily losing the permission to write at those locations. A triple for a program t that mentions only a read-only fraction of a permission in its precondition expresses the fact that t can read but never modify the locations described by the permission. Therefore, such triple asserts that t can be executed in parallel with another subprogram that also only needs to read at those locations.

In concurrent separation logic, the concept of fraction is not limited to read-only resources. To give one more example, a fetch-and-add[13] fraction gives the permission to concurrently use the fetch-and-add atomic instruction at a given location but reading or writing an arbitrary value at that location is impossible until all the fractions are merged back.

13: Modern CPUs feature some atomic operations. Those atomic operations provide guarantees about what happens when executed concurrently at the same memory cell on two threads. Fetch-and-add is one of such atomic operations, that adds a value at a given location, taking concurrent contributions into account. Atomic operations are slower that the corresponding non-atomic operations, therefore they must be used sparsingly.

1.4 Choosing the level of detail for the invariants

Once settled on concurrent separation logic, we still have a lot of flexibility to choose the level of details for the invariants. This choice is important because each level of detail allows proving different kind of program transformations, with different trust models, and at the price of different user annotation efforts.

Full functional correctness invariants The strongest level of details for invariants is called *full functional correctness*. In that case, the final postcondition expresses all the expected mathematical properties about the output. Usually, this implies defining a mathematical function that relates program inputs to expected outputs, asserting that the program actually behaves like this function.

For code transformations, having full functional correctness invariants is very powerful. Indeed, in that case, the final postcondition replaces the code as the ground truth of what must be computed, and the code is simply a description of how such computation is performed. Thus, if a transformation produces any code respecting the same Hoare triple as the original code, it is correct by definition, no matter how far the produced code is from the original code.

An important detail is that, with this level of specification, bugs in the original code can only be located inside the pre- and post-condition of the top-level Hoare triple. Indeed, if the source code contains bugs that do not also appear in the Hoare triple, such Hoare triple cannot be verified. Since the pre- and the post-condition of the Hoare triple are typically shorter than the source code, and are written with a higher level of abstraction, having to trust only those invariants can highly increase the confidence in the software. However, note that it is possible to write and verify an imprecise specification, which can then allow bugs in code produced by transformations even though the original code does not exhibit any bug.

In practice, with full functional correctness, proving that an invariant holds can be an arbitrarily hard task that may involve any complex mathematical theorem and requires finding non-trivial intermediate invariants. Therefore, this level of details for program invariants may require more work than affordable.

Incomplete functional correctness invariants If one cannot afford the costs of full functional correctness proofs, an alternative is to aim instead for *incomplete specifications*. Incomplete functional correctness specifications can use the same kind of invariants as full functional correctness, but those invariants are not expected to capture all the properties that make the algorithm correct. For example, with incomplete specification a postcondition may only state that the output value must be an integer between 0 and 1000 that is even, without specifying which integer, even though a specific return value is expected.

Unlike with full functional correctness invariants, with incomplete specifications, transformations must preserve the semantics of the original code instead of preserving the top-level pre- and post-condition. However, in numerous cases, those incomplete specifications can suffice to justify semantic-preservation of some transformations that are not semantic-preserving in the general case. For instance, knowing that a value is even allows rewriting

$(x/2) \times 2$ into x, and knowing that the value is between 0 and 1000 can guarantee that the value can be stored on 16 bits even though the original code allocated 32 bits for its storage.

Shape-only invariants One particular case of incomplete specification that can be very useful for transformations is the specification of *data shapes*. Such shapes correspond to a description of the memory layout of the data structures.

In practice, the information captured by invariants that only specify data shapes is similar to the information captured by advanced typesystem that track ownership and mutability, such as the typesystem of Rust.

When compilers need to reason about pointers, one particularly important fact is to know whether two pointers may or may not refer to the same memory cells. We call this kind of fact an *aliasing* property. For instance, replacing two consecutive assignments of the form $T[i] = 0; U[j] = 1;$ with $U[j] = 1; T[i] = 0;$ preserves the semantics of the program if the arrays T and U do not alias. If transformations can rely on invariants that specify the memory layout, all of those aliasing properties are available and can therefore be exploited.

Moreover, with fractional permissions, knowing that a subexpression only consumes a read-only fraction of a resource can justify the correctness of a transformation introducing concurrent read-only access to the same resource.

In this PhD, we consider concurrent separation logic invariants with three possible levels of specification: shape-only, incomplete functional correctness, and full functional correctness.

1.5 Mechanically checking and deducing invariants

In the previous section, we saw different kinds of invariants we might want to exploit at each program point in order to perform transformations. However, in order to trust those invariants we need a tool that checks the adequacy of invariants with respect to the behavior of the code. Moreover, writing by hand those invariants between all lines of code would be too repetitive and therefore require too much work. Therefore, before starting interactive code transformations, we also need a mechanized way of deducing invariants between each instruction from only a few user-written annotations.

Hoare proof trees In order to check and deduce those program invariants, we can draw inspiration from formal methods for software verification. In software verification, the typical problem is to check that a given Hoare triple $\{H\}\ t\ \{Q\}$ holds.

$$\frac{\{H\}\ t_1\ \{Q\} \qquad \{Q\}\ t_2\ \{R\}}{\{H\}\ t_1; t_2\ \{R\}}$$

Figure 1.2: Typical rule for the Hoare triple of a sequence of two instructions.

One nice property about Hoare triples is that we can find a small set of rules that can be composed together to prove triples about any program. Those rules form a *Hoare logic* that describes a set of true triples for each construction of the programming language. Typically, the proof of a Hoare triple for a complex program consists of a tree with rules to deduce triples for primitive operations at the leaves, and rules to deduce triples for complex constructions as the nodes (like in figure 1.2).

If we have a Hoare logic proof tree, we get not only the guarantee that a triple holds, but we also can extract the intermediate invariants between each language construction since those intermediate invariants are written in intermediate proof goals. The challenge then becomes to find a way to semi-automatically build such a proof tree.

In practice, the choice of the rules for assembling such a proof tree for a Hoare triple is mostly mechanical as it is syntactically constrained by the program. In that regard, the main choice is the order in which the recursive construction is made in rules such as the one presented in figure 1.2: either a forward approach is taken and then Q is constructed as the *strongest postcondition* deduced from the recursive construction of $\{H\}\ t_1\ \{Q\}$, or a backward approach is taken and then Q is constructed as the *weakest precondition* deduced from the recursive construction of $\{Q\}\ t_2\ \{R\}$. In any case, there are three difficulties:

- Some program constructions require the choice of an intermediate invariant that cannot be guessed automatically in the general case. One such construction is the loop: the Hoare logic rule for a loop requires providing the invariant that holds before and after each iteration. We call such loop invariant a *loop contract*. Another construction that requires the choice of an invariant is the function definition: there the Hoare logic rule checks that a given *function contract* holds for the function body, but such contract cannot be syntactically inferred. Recall that such function contracts corresponds to a Hoare triple that the function body respects and that is used for function calls.
- Sometimes the invariants obtained by the simple application of the Hoare logic rules are not in the right shape or are too precise to conclude. To handle that, Hoare logics usually include *structural rules*, such as the frame rule we saw before in figure 1.1, or the *consequence rule* (figure 1.3) which allows weakening a postcondition or strengthening a precondition[14]. Those structural rules can be applied anywhere and therefore are not syntactically constrained.
- Some Hoare logic rules require hypotheses that are not simply other Hoare triples on subprograms, but arbitrarily complex logical propositions that do not mention any code. We call such propositions *pure proof leaves*. Those pure proof leaves occur, for instance, in the consequence rule (figure 1.3) with the hypotheses $H \Rightarrow H'$ and $Q' \Rightarrow Q$. Such formula $H \Rightarrow H'$ denotes an *entailment*, asserting that the resources mentioned in the invariant H can be reorganized to form the resources of the invariant H' without losing anything. In general, proving a pure proof leaf such an entailment may require an arbitrarily complex mathematical reasoning.

$$\frac{H \Rightarrow H' \qquad Q' \Rightarrow Q \qquad \{H'\}\ t\ \{Q'\}}{\{H\}\ t\ \{Q\}}$$

Figure 1.3: Typical consequence rule for a Hoare logic.

14: Those two rules can be combined together in a single consequence frame rule, which can sometimes help mechanization.

In order to resolve these three issues and build proof trees that assert Hoare triples, software verification tools usually fall in one of two categories: *annotation-guided* or *interactive*.

Annotation-guided software verification Annotation-guided software verification tools leverage user-annotations written in the middle of the code to generate the proof tree. Those annotations specify the Hoare triples that must be checked and the intermediate invariants and structural rules that cannot be automatically guessed. Typically, users of annotation-guided verification tools annotate every function with its intended Hoare triple, and every loop with an invariant that should be true before and after every iteration. Those users, also typically add *ghost instructions* to their code. Such ghost instructions are annotations that take the form of a program

instruction that semantically does nothing; however, they correspond to the manual application of a structural rule, which is sometimes necessary to complete the proof tree. Most annotation-guided verification tools try to limit the number of annotations that are required to verify the Hoare triples. One very common idea is to systematically insert consequence and frame rules around every instruction, limiting ghost instructions for cases where required entailments cannot be guessed.

Famous examples of such annotation-guided tools are Why3 [FP13] for verification of code written in an imperative lambda calculus or FramaC with its WP plugin [Bau+20] for verification of C code. Neither FramaC nor Why3 is based on separation logic, and both have a management of aliasing that can be hard to use with complex data structures. Separation logic is available in more recent tools such as VeriFast [Phi+14], or Viper [MSS16].

[FP13]: Filliâtre et al. (2013), *Why3—Where Programs Meet Provers*

[Bau+20]: Baudin et al. (2020), *WP plug-in manual*

[Phi+14]: Philippaerts et al. (2014), *Software Verification with VeriFast: Industrial Case Studies*

[MSS16]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

Usually, those annotation-guided verification tools discharge the pure proof leaves to an SMT solver[15]. When the automatic proof inference fails, the user of annotation-guided tools, is invited to add more ghost instructions to help the verification process, such as useful intermediate assertions. SMT solvers usually significantly reduce the amount of work needed to verify a program. However, when an SMT solver fails, it is often hard to understand why, as it may either be because the computation became too expensive, or because the property is simply false.

15: Satisfiability Modulo Theories solver: fully automatic program that tries to check if a mathematical property is provably true. Internally, SMT solvers are based on the resolution of the SAT problem which is famously NP-hard, therefore on some inputs, SMT solvers will not provide any answer in reasonable time. That said, the proportion of program verification properties that can be automatically solved by modern SMT solvers in a reasonable time is impressive.

Other annotation-based tools such as RefinedC [Sam+21] remove the need for ghost instructions at the price of more complex annotations on functions, loops, and data type definitions. RefinedC does not rely on SMT solvers, and instead uses its own heuristics for goal directed automatic proof search.

[Sam+21]: Sammler et al. (2021), *RefinedC: automating the foundational verification of C code with refined ownership types*

Interactive software verification On the other hand, interactive software verification tools are more predictable, as they display much more information to the user with an interaction loop. Interactive software verification tools are usually embedded in a proof assistant such as Rocq[16] or Isabelle, and inherit their model of interactivity. The main idea is that a proof assistant always shows what remains to be proven, and asks the user for a proof step. For software verification, this often means that the verification tool displays at each step the Hoare triple that remain to be proven. Then, each user interaction corresponds to the application of a proof step which can either be a structural rule that updates the pre- and post-conditions, or a syntactic rule that decomposes the Hoare triple to prove into Hoare triples on smaller subprograms. In this setup, the proof leaves are mathematical goals that must be proven in the underlying proof assistant.

16: Rocq is the recent new name for the tool formerly known as Coq.

With proof assistants, the series of steps written by the user constitutes a proof script that can be replayed to check the proof on another machine. In the case of Rocq, the rules for the applicability of proof steps do not need to be trusted, as the tool also produce a proof term that can be independently checked by its *kernel*.

Examples of interactive software verification tools with separation logic assertions are CFML [Cha10] and Iris [Jun+18b], both embedded in the Rocq proof assistant. Usually, those interactive software verification tools are used on complex software where invariants are hard to find, or, in the case of Iris, in presence of non-trivial concurrency patterns.

[Cha10]: Charguéraud (2010), *Program Verification Through Characteristic Formulae*

[Jun+18b]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

Hybrid approaches Interactive and annotation-based approaches are not necessarily mutually exclusive categories for software verification tools. For example, Why3 allows locally using the Rocq proof assistant instead of an

SMT solver for proving a proof leaf. Reciprocally, Isabelle [BP13] and Rocq [Arm+11] allow calling SMT solvers on some proof goals. About the Hoare proof tree construction itself, our preliminary work [BC23] showed that one can use an interactive software verification workflow inside Rocq to add all the required annotations to verify a given program, and reciprocally transform a sufficiently annotated program into a proof using CFML.

[BP13]: Blanchette et al. (2013), *Hammering Away*

[Arm+11]: Armand et al. (2011), *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*

[BC23]: Bertholon et al. (2023), *An AST for Representing Programs with Invariants and Proofs*

Foundational software verification tools Additionally, a few software verification tools are *foundational*. A tool is foundational if its implementation is fully proven correct with respect to the semantics of the manipulated programming language. The trusted code base of a program verified by a foundational tool only consists of the formalization of the semantics of the language, the mechanized proof system (e.g. the Rocq kernel) and the program specification itself. Examples of foundational tools include Iris and RefinedC.

In this PhD, we use the annotation-based approach to deduce and check invariants between all instructions. However, unlike most annotation-based tools, we do not try to explore how to best integrate SMT solvers or other automated procedures to discharge proof leaves. Instead, we focus on providing expressive annotations with ghost instructions that may specify proof terms to fulfill proof leaves. Like in Rocq, those proof terms can be typechecked to verify their validity. This design with manually written proof leaves in expressive annotations is chosen because annotations are not only written by the user on the initial code but can also be inserted by our trustworthy transformations whose correctness checks should not depend on the unpredictable execution time of a solver. Moreover, our preliminary work showed that it is feasible to add support for interactive workflows in our interactive compiler, but a full integration of such workflow with the rest of the project is left for future work. Similarly, trying to build a foundational tool is left for future work.

1.6 Correctness of transformation implementations

Once program invariants are found, compilers (interactive or not) can leverage those invariants to check the correctness of their code transformations. However, like any program, the implementation of those transformations themselves may contain bugs.

In general, a bug inside a compiler leads to one of the two undesirable outcomes: either the compiler fails to produce any output, or the compiler produces *miscompiled* executables that do not behave like their source code specifies. It is unfortunate when a bug makes the compiler fail to produce an output, but such bugs only limit the applicability of the compiler, and therefore do not alter the trustworthiness of the compiler on successful outputs. On the other hand, most of the time, miscompilation leads to a bug in the compiled executable even though its original source code is correct.

Testing and fuzzing To limit the risks of miscompilation, general purpose compilers mostly rely on test-driven approaches. However, by construction, tests cannot catch all the bugs, but only the most anticipated ones. Another technique used by general purpose compilers to limit miscompilation bugs is *fuzzing* which corresponds to automated testing on random inputs. One

example of a fuzzing software generating random C code inputs is Csmith [Yan+11]. Fuzzing is not perfect either because a significant amount of compiler bugs only manifest on statistically improbable programs with respect to the chosen random program generator, and therefore have almost no chance being detected. In any case, both regular testing and fuzzing can only prove the presence of compiler bugs, never the absence of such bugs, and miscompilation is still relatively common inside general purpose optimizing compilers[17].

[Yan+11]: Yang et al. (2011), *Finding and Understanding Bugs in C Compilers*

17: Indeed, as one can read in the Csmith paper [Yan+11], intoducing new fuzzing techniques usually finds new miscompilation bugs.

Formal verification of a compiler In order to completely avoid miscompilation issues, one may opt for fully verified compilers using formal methods. In that case the formally verified compiler comes with a theorem that states that if the compiler returns without error an output program, then such output program have fewer possible behaviors than the source program. Generally such a proof can be quite tedious and is formalized in a proof assistant. The most famous examples of verified compilers are probably CompCert [Ler09] and CakeML [Kum+14]. In order to prove such a theorem, let us present three standard techniques: *refinement*, *translation validation*, and *equational reasoning*.

[Ler09]: Leroy (2009), *Formal verification of a realistic compiler*

[Kum+14]: Kumar et al. (2014), *CakeML: A Verified Implementation of ML*

A proof based on refinement is constructed around a set of logical rules with a judgment of the form $\{H\}\ t_s \supseteq t_t\ \{Q\}$ that asserts that under the precondition H, the source program t_s has more possible behaviors than the target program t_t and that those behaviors are all described by the postcondition Q. This kind of refinement has been introduced first in the context of Hoare logic [Ben04], and more recently in the context of separation logic [Gäh+22]. A formal compiler proof then examines the source code of the compiler to show that, in all cases, it is possible to build a proof tree using refinement logical rules to assert the required behavior inclusion.

[Ben04]: Benton (2004), *Simple relational correctness proofs for static analyses and program transformations*

[Gäh+22]: Gäher et al. (2022), *Simuliris: a separation logic framework for verifying concurrent program optimizations*

Translation validation is an orthogonal method that sometimes simplifies compiler formal verification by decoupling two phases: first the compiler produces the target source code, and then a validator is executed with both the source and the target code and such validator is responsible to check that behaviors of the target code are included in behaviors of the source code. If the validator fails, then the compilation process is aborted, and an error is reported to the user. With translation validation, the proof of correctness then only refers to the source code of the validator, which is usually way shorter, and simpler to reason about to establish the required behavior inclusion. Translation validation is one of the key ingredients of CompCert [TL08].

[TL08]: Tristan et al. (2008), *Formal verification of translation validators: a case study on instruction scheduling optimizations*

Finally, equational reasoning is a strategy for proving the correctness of source-to-source transformations over functional programs. The idea, is that two functional programs can be proven equal if they always return the same value with the same input. In order to prove a transformation using equational reasoning, we can express theorems that conclude equalities between the source and the target programs. Then, by composing such theorems, it is possible to prove correct the full compiler. Equational reasoning is the key ingredient for proving the correctness of the ATL interactive compiler [Liu+24].

[Liu+24]: Liu et al. (2024), *A Verified Compiler for a Functional Tensor Language*

Validating the output program with respect to a specification Verified compilers prove that their output code admit the same semantics as their input code. However, relating the semantics of the output code to the semantics of the input code is not always necessary to guarantee correctness of such output code. Indeed, if the source program is equipped with a full

functional correctness specification, the user can accept any target program that satisfies the same specification. In such case, a semantic-preserving compiler can be replaced by a *specification-preserving* compiler that ensures that the target code it produces respects the same specification as the source code. Actually, in this setup, the source code is only a naive implementation hint for the compiler, that could instead rely on program synthesis techniques (e.g. [SGF10]) to transform the input specification into a naive verified implementation. Those specification-preserving compilers can usually apply more aggressive optimizations: indeed, those compilers may produce target code with different semantics than the source code as long as this target code can be independently verified with respect to the specification.

[SGF10]: Srivastava et al. (2010), *From program verification to program synthesis*

The challenge to build a verified specification-preserving compiler is to design a procedure that actually can check that the output code respects the same specification as the source code (instead of admitting the same semantics). Following the validation approach, one may try to build a validator that takes a specification and a program and checks that such program admits such specification. As said in section 1.5, checking that a program admits a given specification usually requires non-trivial proof elements that can be given through code annotations. This idea leads to *proof-carrying code* which contains enough annotations to allow a validator to certify that such code respects some properties (for instance, that it satisfies a given specification). Then, a *certifying compiler* can ensure specification preservation by carrying proof elements from the source proof-carrying code down to the target proof-carrying code, and validating the proof carried by the target code.

Related work on proof-carrying code The original line of work on proof-carrying code [Nec98] did not aim at full functional correctness properties, but rather focused on simpler invariants capturing safety properties, such as the absence of out-of-bound accesses.

[Nec98]: Necula (1998), *Compiling with proofs*

Subsequent work introduced compilers that take as input a formally-verified program and produce as output compiled code accompanied by a formal proof that the compiled code satisfies the same functional correctness specification as the input program. In particular, the PhD work of César Kunz [Kun09; Bar+09] shows how to realize such *proof-transforming compilation* for standard compiler optimizations, applied at the level of an RTL[18] intermediate language.

[Kun09]: Kunz (2009), *Proof preservation and program compilation*
[Bar+09]: Barthe et al. (2009), *Certificate Translation for Optimizing Compilers*

18: Register Transfer Language: Intermediate representation of a program inside a compiler which describes how the data moves and is processed between CPU registers. RTL is very close to an assembly language.

Later projects introduced variants of proof-carrying code using separation logic. This is the case of Alpinist [Sak+22] a pragma-based compiler[19] that targets GPU array-based computations, and preserves proof annotations (that can be later checked in Viper) through code transformations.

[Sak+22]: Sakar et al. (2022), *Alpinist: An Annotation-Aware GPU Program Optimizer*

19: A pragma-based compiler is an interactive compiler in which users can request a specific transformation by adding an annotation on the transformed instruction. This model of interaction tends to be hard to use whenever there is more than one transformation to apply on a single instruction.

In this PhD, we build an interactive compilation framework that supports the proof-carrying code approach: each transformation updates a formal proof that the code satisfies its specification, and a separation logic proof typechecker can validate the well-formedness of such proof at any step. The proof obtained at the final step is enough to guarantee the absence of miscompilation only if the specification fully characterizes the wanted behavior of the program. If the user gives incomplete specification, such proof obtained at the end only guarantees what is written in this incomplete specification. In particular, for shape-only annotations, the final proof only guarantees memory safety properties. Said differently, with incomplete specifications a specification-preserving compiler is not enough and should be replaced by a semantic-preserving compiler. In this incomplete specification case, we currently use the same trust model as general-purpose compilers, relying

on tests to check that transformations are not causing miscompilation on common cases. We think that most users wanting strong formal guarantees are more interested in full specifications, and therefore the absence of formal compiler verification in the incomplete specification case is mostly a theoretical issue.

1.7 Contributions

During my PhD, I worked on an interactive compiler named OptiTrust. OptiTrust is a tool which applies user-guided trustworthy source-to-source transformations. The OptiTrust project is developed within the ICPS team of the ICube laboratory under the direction of my supervisor Arthur Charguéraud. Prior to the start of my PhD, a prototype for OptiTrust was developed by Arthur Charguéraud, Begatim Bytyqi, and Damien Rouhling. This early prototype was not concerned by transformation correctness: back then, OptiTrust blindly applied any user request, even if it might create a bug in the optimized program. My main contribution described in the rest of this manuscript is the implementation of mechanisms that ensure that OptiTrust transformations are correct.

Contents of this manuscript Chapter 2 illustrates the OptiTrust workflow on three case studies: a blur function from image processing, a kernel for particle simulation, and a matrix-multiplication algorithm. In those three cases, we reproduce the same optimization as the state-of-the-art implementation, with our trustworthy workflow. The implementation inside OptiTrust of all the trustworthy source-to-source transformations required to realize these case studies is a joint work with my supervisor and Thomas Kœhler (a post-doc researcher in the team back then).

Chapter 3 describes my preliminary work on the OptiTrust frontend. This chapter contains the formalization of a translation from a language close to C that should be familiar to high-performance code experts (*OptiC*) to an internal imperative λ-calculus that makes implementation and reasoning on transformations easier (*Optiλ*).

Chapter 4 describes a typing algorithm with separation logic resources that uses annotations on loops and functions, as well as a few ghost instructions, to check that specifications hold and find invariants at any point in the program. This system supports the three aforementioned specification levels of detail: either the user simply specifies information about the shape of the data, or they can specify functional correction properties for their algorithm, in that second case they can either choose full or incomplete specifications. This system (and its extension presented in the following chapter) is a personal contribution to the OptiTrust project, that plays a major role in checking the correctness of transformations in OptiTrust.

Chapter 5 describes a key extension of this typing algorithm that is crucial to justify some program transformations: the production of resource usage summaries. These summaries are computed on each node of the program's syntax tree to deduce which resources are essential to the execution of an instruction. The chapter details the format of these resource usage summaries and how our typechecker computes them.

Chapter 6 describes some code transformations inside OptiTrust. Those transformations support two different modes of correctness checks: *semantic preservation* that can be used with incomplete specifications and *specification preservation* that can only be used in presence of full functional specifications.

In semantic preservation mode, transformations use the combination of invariants at all program points and usage summaries (both computed by the aforementioned typechecker) to check their own applicability conditions before executing. On this point, my personal contribution mostly consists in finding patterns of invariant and usage exploitation that can be used across transformations, and helping Thomas Kœhler designing the correctness criterions. In specification preservation mode, the transformations do not need to check their own applicability, their implementation is no longer part of the trusted code base, and the typing algorithm with annotations alone can guarantee that the expected semantics are preserved. In both cases, as mentioned previously, to be able to make chains of transformations, another essential point of my work consisted in adjusting the transformations to maintain relevant annotations after code modification. On this point, I worked in particular on developing strategies for transforming annotations (notably ghost instructions) that are a priori troublesome, so as to increase the scope of transformations on executable code.

In the end, the differences between OptiTrust with shape-only annotations and with functional correctness specifications are rather small, because we anticipated the need for different levels of detail inside specifications. Those small differences are described inside the relevant chapters.

Publications My PhD research lead to the following publications:

- Guillaume Bertholon and Arthur Charguéraud. "An AST for Representing Programs with Invariants and Proofs". In: *34èmes Journées Francophones des Langages Applicatifs (JFLA 2023)*. 2023

  This 15 page long workshop paper presents our preliminary work for finding an appropriate representation for programs annotated with invariants and the proof that such invariants hold. This work presents an encoding in Rocq for Hoare proof trees as programs annotated with specifications and ghost instructions, and then proposes an interactive program verification workflow to add relevant annotations to an existing program, and extract a foundational proof out of a fully annotated program. This PhD manuscript does not reproduce the content of this publication since most of its ideas are currently not integrated in OptiTrust.

- Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. "Interactive source-to-source optimizations validated using static resource analysis". In: *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 2024

  This 7 page long workshop paper gives an overview of OptiTrust with shape-only annotations, illustrating the tool on the matrix multiplication case study. This case study is reproduced in this manuscript in section 2.3.1.

- Guillaume Bertholon and Arthur Charguéraud. "Bidirectional Translation between a C-like Language and an Imperative λ-calculus". In: *36èmes Journées Francophones des Langages Applicatifs (JFLA 2025)*. 2025

  This 16 page long workshop paper describes our preliminary work to formally describe OptiC, Optiλ, and the translation between those two languages. Most of the chapter 3 is directly taken from this paper.

Moreover, we submitted the following publication that is currently undergoing a major revision before publication in the journal TOPLAS:

- Guillaume Bertholon, Arthur Charguéraud, Kœhler, Begatim Bytyqi, and Damien Rouhling. "OptiTrust: Producing Trustworthy High-Performance Code via Source-to-Source Transformations". Draft paper. 2024

  This draft of a 60-page-long journal paper describes OptiTrust in the particular case of shape-only annotations. Chapters 2, 4, 5 and 6 are improved versions of sections from this draft. Most importantly, these sections are extended with arbitrary functional correctness properties, and specification preserving transformations.

Besides those publications, OptiTrust is an open source software available on GitHub[20]. In the project codebase, I made the following contributions:

20: https://github.com/charguer/optitrust

- I personally wrote all the code of the separation logic typechecker and the utilities to parse and manipulate annotations.
- I adapted the code of several transformations to use typechecking information to check their correctness and adapt annotations for the next step.
- I developed the annotation only transformations we describe in section 6.6.
- I integrated a system of composable smart deconstructors similar to what can be found in OCaml's ppxlib[21] to make the code of transformations more readable.

21: https://ocaml-ppx.github.io/ppxlib/ppxlib/matching-code.html#ast_pattern_intro

- I clarified the design of Optiλ and fixed the broken translation between OptiC and Optiλ.
- I heavily contributed to the interactive transformation trace visualization infrastructure.
- I added a basic form of continuous integration to ensure that new commits always pass the test suite.

Current limitations At the end of my thesis, OptiTrust still has important limitations:

- OptiTrust manipulates programs written in OptiC, a programming language which does not yet support all features of an imperative language such as C. Most importantly, OptiC does not feature any language construction that could introduce non-termination such as general **while** loop or recursive functions. As our case studies show, this subset nevertheless suffices to express numerous practical, high-performance programs, in an idiomatic programming style both for the unoptimized and for the optimized code. Moreover, for simplicity, in our transformations, we currently use unbounded integers to ignore complications related to arithmetic overflows, and we use symbolic reals to avoid floating point rounding.
- We have so far restricted ourselves to a small subset of separation logic. Our resource-based typesystem is able to describe the ownership of arrays, matrices, or individual cells, however there is currently no support for user-defined heap predicates. Nevertheless, as our case studies show, we are able to demonstrate the usefulness of our approach in this limited setup.

- We have already implemented dozens of transformations, among the most standard ones. We believe that these transformations suffice to assess the interest of the OptiTrust approach to code optimization. However, for production usage, dozens of additional transformations remain to implement.
- We strived to limit the trusted code base of OptiTrust. In particular, with full functional correctness invariants, the implementation of the transformations is not in the trusted code base, and, with incomplete specifications, complex transformations do not need to be trusted because they are built on top of a smaller subset of trusted basic transformation. However, we are still far from having a foundational tool. Indeed, we have not yet proven correct our separation logic typechecking rules and our typechecker implementation with respect to the semantics of our programming language, we currently use an untrusted backend to produce machine code after the interactive steps, and we have not proven correct our basic transformations that support incomplete specifications. Completing such mechanized proof will presumably require several years of additional work. Note that state-of-the-art compilers such as Halide have been described in publications that did not include correctness proofs.

Chapter 7 gives perspectives on how to remove those limitations in the future. The effectiveness of separation logic has been successfully demonstrated across a broad range of applications, both for low-level and high-level code [OHe19; Cha20b]. By building OptiTrust on separation logic, we are confident that our framework has the potential to be generally applicable.

[OHe19]: O'Hearn (2019), *Separation logic*
[Cha20b]: Charguéraud (2020), *Separation logic for sequential programs (functional pearl)*

In summary, the version of OptiTrust we present in this PhD can readily be exploited to optimize certain classes of programs, but we acknowledge that future work remains necessary to achieve full generality. Note that we have taken great care in our design and implementation to anticipate for the extensions to a richer programming language and to a richer separation logic.

2 OptiTrust in practice

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

A central aspect of OptiTrust is that it guarantees that the code transformations requested by the programmer preserve the semantics or the full specifications of the program. To that end, OptiTrust leverages our separation logic typesystem.

For typechecking separation logic resources, functions and loops need to be equipped with *contracts* describing their resource usage. These contracts must be provided as annotations (in the form of no-op instructions) in the OptiC source code. OptiTrust is able to automatically infer simple loop contracts, thus not all loops need to be annotated manually. Crucially, every OptiTrust transformation takes care of updating contracts in order to reflect changes in the code. In other words, a well-typed program must remain well-typed after a successful transformation. This property is essential to ensure that subsequent transformations in the optimization chain can be validated by exploiting information from our typechecker.

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

An OptiTrust user manipulates programs written in *OptiC*, an imperative C-like language augmented with typing annotations—function and loop *contracts*, as well as *ghost code*. However, OptiTrust does not directly manipulate an AST for OptiC. Instead, it operates on an intermediate representation that essentially consists of an imperative λ-calculus. We call this internal language *Optiλ*. Concretely, the OptiC code, expressed in C syntax, is first parsed using Clang. Then, the OptiC code is translated into Optiλ. In particular, our translation eliminates mutable variables and operations involving left-values. Importantly, our translation is bidirectional, allowing to print back human-readable OptiC code after transformations are applied on the internal AST. Considering a simple internal language such as Optiλ considerably helps to tame the complexity of the design and implementation of typing rules, code transformations, and correctness criterions associated with transformations.

This chapter presents these features of OptiTrust through three case studies. In section 2.1, we reproduce manually written code from OpenCV—a very popular, optimized computer vision library. In section 2.2, we consider a physics simulation program featuring a kernel typical of particle simulations; we demonstrate how to apply, using OptiTrust, several optimizations that are ubiquitous in this kind of application. In section 2.3, we reproduce an optimized implementation of matrix multiplication, similar to the one produced by TVM, the state-of-the-art specialized compiler for machine learning applications. On this example, we demonstrate the use of two different levels of details for resource annotations. First, like both other examples, we show how we can derive optimized code by relying on incomplete specifications only specifying the shape of data. Then, we show that OptiTrust is able to preserve a proof of full functional correctness through optimizations, if such functional correctness is established on the initial code. Then, in section 2.4, we evaluate OptiTrust against the desirable properties for interactive code optimization frameworks.

2.1 The OpenCV row-based blur case study

In image processing, a *blur* is typically used to remove noise and smoothen images. A two-dimensional blur can be decomposed as a combination of *column-based blur*, *row-based blur*, and (optionally) the application of a normalization pass. Our case study focuses on a *row-based blur* function, as implemented in the state-of-the-art OpenCV library [BK+00].

[BK+00]: Bradski et al. (2000), *OpenCV*

Unoptimized code If performance was not a concern at all, the row-based blur function would be implemented as shown in figure 2.1. The output is a single-row image, stored in an array named `D`, made of `n` pixels. The input is a single-row image, stored in an array named `S`, made of `n+w-1` pixels, where the parameter `w` corresponds to the width of the blur. The input pixels in `S` are encoded on `cn` integers of type **`uint8_t`**, whereas the output pixels in `D` are encoded on `cn` integers of type **`uint16_t`**. The output pixel `D[i]` is computed as the sum of the values of the input pixels in the range from `S[i]` to `S[i+w-1]`. This sum is computed independently for each of the `cn` color channels. The code accommodates any value of `cn`, but practical values include `cn=1` for grayscale, `cn=3` for RGB, `cn=4` for RGBA.

Optimized code The handwritten OpenCV library includes an implementation of row-sum blur structured like the code shown in figure 2.2. The original OpenCV code may be viewed online[1]. The code from figure 2.2 corresponds to the code that we produce using OptiTrust.

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

```
void rowSum(const int n, const int cn, const int w, const uint8_t S[n+w-1][cn], uint16_t D[n][cn]){
  for (int i = 0; i < n; i++) { // for each target pixel in the row described by D
    for (int c = 0; c < cn; c++) { // for each channel (e.g., red, green, and blue)
      uint16_t s = 0;
      for (int k = i; k < i+w; k++) // for each source pixel nearby to the right
        s += (uint16_t) S[k][c];
      D[i][c] = s;
} } }
```

Figure 2.1: Unoptimized C code for the OpenCV case study, using multidimensional arrays.

```
void rowSum(int32_t n, int32_t cn, int32_t w, uint8_t* S,
      uint16_t* D) {
  if (w == 3) {
    for (int32_t ic = 0; ic < cn * n; ic++) {
      D[ic] = (uint16_t) S[ic]
            + (uint16_t) S[cn + ic]
            + (uint16_t) S[2 * cn + ic];
    }
  } else if (w == 5) {
    for (int32_t ic = 0; ic < cn * n; ic++) {
      D[ic] = (uint16_t) S[ic]
            + (uint16_t) S[cn + ic]
            + (uint16_t) S[2 * cn + ic]
            + (uint16_t) S[3 * cn + ic]
            + (uint16_t) S[4 * cn + ic];
    }
  } else if (cn == 1) {
    uint16_t s = (uint16_t) 0;
    for (int32_t i = 0; i < w; i++) {
      s += (uint16_t) S[i];
    }
    D[0] = s;
    for (int32_t i = 0; i < n - 1; i++) {
      s += (uint16_t) S[i + w] - (uint16_t) S[i];
      D[i + 1] = s;
    }
  } else if (cn == 3) {
    uint16_t s0 = (uint16_t) 0;
    uint16_t s1 = (uint16_t) 0;
    uint16_t s2 = (uint16_t) 0;
    for (int32_t i = 0; i < 3 * w; i += 3) {
      s0 += (uint16_t) S[i];
      s1 += (uint16_t) S[i + 1];
      s2 += (uint16_t) S[i + 2];
    }
    D[0] = s0;
    D[1] = s1;
    D[2] = s2;
    for (int32_t i = 0; i < 3 * n - 3; i += 3) {
      s0 += (uint16_t) S[3 * w + i] - (uint16_t) S[i];
      s1 += (uint16_t) S[3 * w + i + 1] - (uint16_t) S[i + 1];
      s2 += (uint16_t) S[3 * w + i + 2] - (uint16_t) S[i + 2];
      D[i + 3] = s0;
      D[i + 4] = s1;
      D[i + 5] = s2;
    }
  } else if (cn == 4) {
    // [...] similar to cn == 3, with one more variable
  } else {
    for (int32_t c = 0; c < cn; c++) {
      uint16_t s = (uint16_t) 0;
      for (int32_t i = 0; i < cn * w; i += cn) {
        s += (uint16_t) S[c + i];
      }
      D[c] = s;
      for (int32_t i = c; i < cn * n - cn + c; i += cn) {
        s += (uint16_t) S[cn * w + i] - (uint16_t) S[i];
        D[cn + i] = s;
  } } } }
```

Figure 2.2: Our optimized C code for the OpenCV case study, showing the body of the `rowSum` function. This code exploits essentially the same optimizations as the original OpenCV code.

This optimized implementation is *multi-versioned* code, with dedicated execution paths for handling specific values of the parameters. The branches `w == 3` and `w == 5` correspond to values of the width that are commonly used by library users. For these small constant values of `w`, the inner loop on `k` from figure 2.2 is unfolded. Otherwise, the loop on `k` is not unfolded and a standard algorithmic optimization called *sliding window* is applied. Note that Halide, the state-of-the-art specialized compiler for image processing, does not support the introduction of sliding windows—and the developers of Halide do not plan to lift this limitation.[2]

The branch of the code that uses the sliding window optimization is then further specialized with branches for commonly used parameters: `cn == 1`, `cn == 3`, and `cn == 4`. For these small constant values of `cn`, the outer loop on `c` is unfolded, then the multiple occurrences of the loop on `i` that result from this unfolding are fused into a single loop. The final **`else`** branch in the code from figure 2.2 corresponds to the generic implementation. Moreover, in the last three branches, the loops are reindexed to augment the counter `i` by steps of `cn`, thereby saving multiplication operations.

2: Halide does not support sliding windows for reasons explained on: https://github.com/halide/Halide/issues/180. Hence, the programmer either needs to manually refine the code to introduce the sliding window before scheduling; or needs to exploit other transformation tools specialized in sliding window optimizations [Cha+15; Kan+24].

Multidimensional versus flat arrays The code from figure 2.1 is presented using C syntax for multidimensional arrays, for the sake of improved readability. However, the optimized code from figure 2.2 and our contract-annotated code from figure 2.3 instead use a flat array representation. The flat representation is frequently used in high-performance code: it allows performing simplifications in array accesses, moreover it allows for compatibility with C++ parsers[3]. We leave to future work the parsing of multidimensional arrays.

3: In C++, unlike in C, arrays must have a size known at compilation time, therefore the code from figure 2.1 is rejected by a Clang parser in C++ mode. For technical reasons such as the need for anonymous functions in OptiC, OptiTrust currently relies on a C++ parser.

Annotated unoptimized code Before we can start optimizing the code from figure 2.1 using OptiTrust, we need to annotate the code with *function contracts*, *loop contracts*, as well as *ghost instructions*. A contract consists of a description of the assumptions and guarantees associated with a function or

```
void rowSum(int n, int cn, int w, int* S, int* D) {
  __requires("w >= 0, n >= 1, cn >= 0");
  __reads("S ⇝ Matrix2(n+w-1, cn)");
  __writes("D ⇝ Matrix2(n, cn)");
  for (int i = 0; i < n; i++) { // for each pixel
    __xwrites("for c in 0..cn → &D[MINDEX2(n, cn, i, c)] ⇝ Cell");
    for (int c = 0; c < cn; c++) { // for each channel
      __xwrites("&D[MINDEX2(n, cn, i, c)] ⇝ Cell");
      __ghost(assume, "is_subrange(i..(i+w), 0..(n+w-1))");
      int s = 0;
      for (int k = i; k < i+w; k++) {
        __ghost(in_range_extend, "k, i..(i+kn), 0..(n+kn-1)");
        __ghost_begin(focus, matrix2_ro_focus, "S, k, c");
        s += S[MINDEX2(n+w-1, cn, k, c)];
        __ghost_end(focus);
      }
      D[MINDEX2(n, cn, i, c)] = s;
    }
  }
}
```

Figure 2.3: Unoptimized OptiC code for the OpenCV case study, using flat arrays and resource annotations.

a loop, as well as a description of the side effects that may be performed. A ghost instruction behaves, semantically, as a no-op. Its purpose is to guide the typechecker of OptiTrust, typically by altering the way the memory state is described in the separation logic invariants. These invariants may be exploited for guiding code transformations, and for checking their correctness.

Ghost instructions may also be used to keep track of nontrivial arithmetic reasoning involved in the typechecking process. Typically, we need to derive arithmetic inequalities, to justify that a certain range falls within the bounds of an array. In this example, we derive from the lemma `in_range_extend` the fact that the range from i to $i + w$ is included in the range from 0 to $n + w - 1$. In future work, we hope to integrate an SMT solver to discharge this kind of goal.

Besides, to ease the manipulation and typechecking of multidimensional arrays, all accesses are assumed to be written using a family of functions called `MINDEX`. For example, `D[MINDEX2(n,cn,i,c)]` denotes access in the flat array `D`, of dimensions `n` × `cn`, at the coordinates `(i,c)`. We leave it to future work to support input programs written without explicit dimensions on array accesses.

Figure 2.3 shows the same code as in figure 2.1 in OptiC, the input language of OptiTrust. Compared to the corresponding unoptimized C code, the OptiC code is augmented with contracts, relevant ghost instructions, and `MINDEX` accesses.

The clause **`__requires`** contains assumptions about the input parameters. The clause **`__reads`** asserts that the input array `S` can be accessed in read-only mode. The clause **`__writes`** asserts that the output array `D` is completely overwritten by the function and that its content at the beginning of the function is never read. We could also replace this **`__writes`** clause with the more generic clause **`__modifies`** (like we will see in the next case study) to simply assert that data can be modified in place, without specifying that its initial value is ignored[4]. The clause **`__xwrites`** describes a *loop contract*: it indicates not only that the `i`-th iteration writes in certain cells, but also that the other iterations do not access these cells. In other words, the `i`-th

4: Such a less precise annotation conveys less information and thus can make some transformations fail if their correctness depend on the fact that the values described by the clause are never read. However, in some cases such as this one, we can use a procedure called *contract minimization* detailed later in chapter 5 to automatically stenghten a function contract with a **`__modifies`** clause into a contract with a **`__writes`** clause.

```
Reduce.intro [cVarDef "s"];
Specialize.variable_multi ~mark_then:fst ~mark_else:"anyw"
  ["w", int 3; "w", int 5] [cFunBody "rowSum"; cFor "i"];
Reduce.elim ~inline:true [nbMulti; cMark "w"; cCall "reduce_spe1"];
Loop.collapse [nbMulti; cMark "w"; cFor "i"];
Loop.swap [nbMulti; cMark "anyw"; cFor "i"];
Reduce.slide ~mark_alloc:"acc" [nbMulti; cMark "anyw"; cArrayWrite "D"];
Reduce.elim [nbMulti; cMark "acc"; cCall "reduce_spe1"];
Variable.elim_reuse [nbMulti; cMark "acc"];
Reduce.elim ~inline:true [nbMulti; cMark "anyw"; cFor "i"; cCall "reduce_spe1"];
Loop.shift_range StartAtZero [nbMulti; cMark "anyw"; cFor "i"];
Loop.scale_range ~factor:(trm_find_var "cn" []) [nbMulti; cMark "anyw"; cFor "i"];
Specialize.variable_multi ~mark_then:fst ~mark_else:"anycn" ~simpl:custom_specialize_simpl
  ["cn", int 1; "cn", int 3; "cn", int 4] [cMark "anyw"; cFor "c"];
Loop.unroll [nbMulti; cMark "cn"; cFor "c"];
Target.foreach [cMark "cn"] (fun c →
      Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor "i" ~body:[cArrayWrite "D"]];
      Instr.gather_targets [c; cStrict; cArrayWrite "D"];
      Loop.fusion_targets ~into:FuseIntoLast [nbMulti; c; cFor ~stop:[cVar "w"] "i"];
      Instr.gather_targets [c; cFor "i"; cArrayWrite "D"]; );
Loop.shift_range (ShiftBy (trm_find_var "c" [cMark "anycn"]))
  [cMark "anycn"; cFor ~body:[cArrayWrite "D"] "i"];
Cleanup.std ~int_size:[(typ_u8, "S"), (typ_u16,  "D\\|s.*")] ();
```

Figure 2.4: Optimization script for the OpenCV case study.

iteration has exclusive access to that cell. The "x" prefix in **__xwrites** stands for "exclusive". Next case studies also use the x variants of other clauses such as **__xmodifies**.

In particular, the outer loop on i is annotated with a clause involving an iteration construct: **__xwrites**("for c in 0..cn → &D[MINDEX2(n, cn, i, c)] ⇝ Cell"). This clause indicates that the i-th iteration of that outer loop requires exclusive access to all the cells in the i-th row of the destination array D. Further in the manuscript, this same resource may also be written using the corresponding math notation, as: $\bigstar_{c \in 0..\mathsf{cn}} \mathsf{D} \boxplus \mathsf{mIndex2}(\mathsf{n}, \mathsf{cn}, \mathsf{i}, c) \rightsquigarrow \mathsf{Cell}$, where the star symbol is called *iterated separating conjunction* in separation logic. The iteration construct is also used to define the Matrix2 predicate, which describes a 2D range of individual cells. Concretely, the resource D ⇝ Matrix2(n,cn) is equivalent to $\bigstar_{i \in 0..\mathsf{n}} \bigstar_{c \in 0..\mathsf{cn}} \mathsf{D} \boxplus \mathsf{mIndex2}(\mathsf{n}, \mathsf{cn}, i, c) \rightsquigarrow \mathsf{Cell}$, which covers all the n × cn cells of the matrix D.

The lines introduced by **__ghost_begin**, **__ghost_end**, or sometimes just **__ghost** correspond to *ghost instructions*: no-ops whose purpose is to change the view on the resources, or prove mathematical properties. The need for ghost instructions is standard in separation logic frameworks. The specialized keywords **__ghost_begin** and **__ghost_end** materialize a pair of ghost instructions that are the reciprocal of one another. For example, the ghost focus operation allows recovering a single memory cell from the array S, isolating S ⊞ mIndex2(n+w-1, cn, k, c) ⇝ Cell from $\bigstar_{j \in 0..\mathsf{n+w-1}} \bigstar_{c \in 0..\mathsf{cn}} \mathsf{S} \boxplus \mathsf{mIndex2}(\mathsf{n+w-1}, \mathsf{cn}, j, c) \rightsquigarrow \mathsf{Cell}$. Technically, the focus involves read-only fractions and a "magic wand" describing the remaining cells. The matching **__ghost_end** pseudo-instruction applies the symmetrical operation, recovering the original resource. In the future, we could try to rely on heuristics for automatically inferring certain ghost operations, and reduce the number of such ghost operations that need to be explicitly provided by the programmer.

Optimization script syntax Figure 2.4 shows our script for generating the optimized code of figure 2.2 starting from the annotated unoptimized code of figure 2.3. In OptiTrust, optimizations are dictated by means of a script written in the OCaml programming language. For the reader not familiar with OCaml, `f x y` denotes the call of `f` on the arguments `x` and `y`; the symbol `~` is used to provide optional (or named) arguments; `[x; y; z]` denotes a list; `(x, y, z)` denotes a tuple; `x ^ y` denotes a string concatenation; and `let f x = t in` introduces a local function.

A transformation script consists of a series of calls to functions from the OptiTrust library. Each call may depend on a number of arguments controlling the transformations. By convention, the last argument of a transformation always denotes a *target*. Before explaining the working of targets, we first present the transformations involved in our script from figure 2.2. `Reduce.intro` introduces a map-reduce operation for computing the sum over a segment. `Reduce.elim` eliminates a map-reduce into an explicit summation. `Reduce.slide` performs a sliding window optimization on a map-reduce computation. `Specialize.variable_multi` introduces a cascade of if-statements for testing specific variable values. `Loop.collapse` takes two nested loops and replaces them with a single loop that iterates over the product space. `Loop.swap` takes two nested loops and swaps them. `Variable.elim_reuse` takes two variables with equal values and eliminates the second variable. `Loop.shift_range` and `Loop.scale_range` allow altering the iteration range of a loop. `Loop.unroll` unrolls a loop with a statically known number of iterations. `Loop.fusion_targets` fuses targeted loops into a single one. `Instr.gather_targets` reorders instructions in a sequence to make the targeted instructions consecutive. `Cleanup.std` eliminates all dependencies on the OptiTrust header file, performs arithmetic simplifications and sets the size for integer (and floating point) types in order to produce conventional C syntax as final output[5].

5: As of June 2025, the choice of integer size is not yet implemented in `Cleanup.std`. Instead, we currently start with sized integers in the unoptimized code, and we ignore this size during the optimization process. We believe the presentation of this manuscript is better, since it makes explicit that the sizing operation is performed only at the end, and that integer overflows are not checked by our transformations at the moment.

Targets A target provides a way to concisely and robustly refer to one or several code locations, at which to apply a transformation. The construct `Target.foreach`, visible in figure 2.4, can also be used to explicitly iterate over several code locations. A target consists of a list of constraints (prefixed by "c") that is satisfied by code paths that go through nodes satisfying each constraint, in the given order. For example, the constraint `cFunBody "rowSum"` requires visiting the body of a function definition with the name `"rowSum"`. The constraint `cFor "c"` requires visiting a **for** loop over an index with the name `"c"`. The constraint `cMark "cn"` requires visiting an AST node that carries the mark `"cn"`. Such marks are introduced by transformations, on demand of the programmer.

Constraints may also take targets as arguments: `cFor "i" ~body:[cArrayWrite "D"]` requires visiting a **for** loop over an index with the name `"i"`, whose body also writes in the array `D`. Besides, targets may include special modifiers. The modifier `nbMulti` indicates that the programmer expects to find not one but multiple AST nodes that match this target. The modifier `tBefore`, which appears in the other two case studies, allows targeting the interstice before an instruction.

Interactive visualization Each step of an evaluation script may be executed interactively: with the cursor on a line, the OptiTrust user can press a shortcut key in their code editor to visualize the *diff* associated with the transformation on that line. Figure 2.5 shows the diff associated with the `Loop.scale_range` transformation that appears near the middle of the script from figure 2.4. This transformation reindexes a loop. In the present

```
 for (int c = 0; c < cn; c++) {
  uint16_t s = (uint16_t)0;
  for (int i = 0; i < w; i++) {
   s = s + (uint16_t)S[MINDEX2(n + w - 1, cn, i, c)];
  }
  D[MINDEX2(n, cn, 0, c)] = s;
  for (int i = 0; i < n - 1; i++) {
   s = s + (uint16_t)S[MINDEX2(n + w - 1, cn, i + w, c)] -
     (uint16_t)S[MINDEX2(n + w - 1, cn, i, c)];
   D[MINDEX2(n, cn, i + 1, c)] = s;
  }
 }
```

```
 for (int c = 0; c < cn; c++) {
  uint16_t s = (uint16_t)0;
  for (int i = 0; i < cn * w; i += cn) {
   s = s + (uint16_t)S[MINDEX2(n + w - 1, cn, exact_div(i, cn), c)];
  }
  D[MINDEX2(n, cn, 0, c)] = s;
  for (int i = 0; i < cn * (n - 1); i += cn) {
   s = s + (uint16_t)S[MINDEX2(n + w - 1, cn, exact_div(i, cn) + w, c)] -
     (uint16_t)S[MINDEX2(n + w - 1, cn, exact_div(i, cn), c)];
   D[MINDEX2(n, cn, exact_div(i, cn) + 1, c)] = s;
  }
 }
```

Figure 2.5: Diff for the `Loop.scale_range` transformation that appears near the middle of the script from figure 2.4. OptiTrust can also produce a more verbose diff that includes contracts, and ghost instructions.

example, it modifies the indexing from `for (int i = 0; i < w; i++)` to `for (int i = 0; i < cn*w; i+=cn)`, and replaces every occurrence of the index `i` with the expression `exact_div(i,cn)`. In particular, the array access `S[MINDEX2(n + w - 1, cn, i, c)]`, becomes `S[MINDEX2(n + w - 1, cn, exact_div(i,cn), c)]`. The final cleanup step of our script unfolds the definition of `MINDEX2` to obtain `S[cn * exact_div(i, cn) + c]`, then applies an arithmetic simplification to obtain the index `S[c + i]`. The latter expression appears in the final code presented in figure 2.2. Additionally, OptiTrust can produce a complete *execution trace* in the form of an interactive tree. This tree reports the diff not only for every transformation visible in the script, but also for all the internal transformations that are leveraged in the process.[6]

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

Validity checks The transformation script from figure 2.4 consists of *combined* transformations, whose evaluation triggers the application of a chain of *basic* transformations. As said earlier, basic transformations are those that directly modify the AST, whereas combined transformations are defined as the composition of basic transformations. For every basic transformation being applied, OptiTrust checks that this transformation preserves the semantics of the program, by leveraging resource typing information. Because the checks performed by OptiTrust depend on resource typing, every intermediate program must typecheck. In particular, if a transformation modifies the code, it may need to also modify the annotations, such as the loop contracts and the ghost instructions. Correctness criterions and preservation of typing are discussed in details in chapter 6.

2.2 The particle simulation case study

Particle-In-Cell (PIC) is a technique commonly used to simulate plasma, where charged particles are in motion, by approximating the charge distribution using a grid. Our case study is inspired by the work from [Bar+18], who present a PIC implementation featuring state-of-the-art optimizations. In the present case study, we consider a simplified PIC simulation, focusing on the computations associated with one particular cell of the grid. Our goal is to illustrate how OptiTrust can be used to derive a certain number of transformations ubiquitous in particle simulation as well as other physics simulation code.

[Bar+18]: Barsamian et al. (2018), *Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors*

Unoptimized code Figure 2.6 shows the unoptimized simulation kernel that we consider. A number of particles, all with the same mass and charge, move inside a cubic cell. For simplicity, we assume in this case study that the particles do not leave the cube. The initial position and speed of every particle is given. Positions are described with values in the range $[0, 1]$, for

```
void simulate(vect* fieldAtCorners, int nbSteps, real deltaT, real pCharge, real pMass, int nbPart,
     particle* part) {
  __reads("fieldAtCorners ~~> Matrix1(nbCorners)");
  __modifies("part ~~> Matrix1(nbPart)");
  for (int idStep = 0; idStep < nbSteps; idStep++) {
    for (int idPart = 0; idPart < nbPart; idPart++) {
      // Each particle is updated at each time step
      __xmodifies("&part[MINDEX1(nbPart, idPart)] ~~> Cell");
      __ghost_begin(part, particle_open, "&part[MINDEX1(nbPart, idPart)]");
      // Interpolate the field based on the position relative to the corners of the cell
      real* const coeffs = MALLOC1(real, nbCorners);
      compute_corner_interpolation_coeffs(part[MINDEX1(nbPart, idPart)].pos, coeffs);
      const vect fieldAtPos = matrix_vect_mul(coeffs, fieldAtCorners);
      free(coeffs);
      // Compute the acceleration:  F = m*a  and  F = q*E  gives a = q/m*E
      const vect accel = vect_mul(pCharge / pMass, fieldAtPos);
      // Compute the new speed and position for the particle
      const vect speed2 = vect_add(part[MINDEX1(nbPart, idPart)].speed, vect_mul(deltaT, accel));
      const vect pos2 = vect_add(part[MINDEX1(nbPart, idPart)].pos, vect_mul(deltaT, speed2));
      // Update the particle
      part[MINDEX1(nbPart, idPart)].speed = speed2;
      part[MINDEX1(nbPart, idPart)].pos = pos2;
      __ghost_end(part);
} } }
```

Figure 2.6: Unoptimized code for the particle simulation case study, with resource annotations.

each axis. We assume that the particles do not affect each other, and that an external electric field affects the acceleration of the particles[7]. The electric field is described by 8 vectors, one per corner of the cell. The electric field that applies at a given position inside the cubic cell is obtained by linearly interpolating the vectors associated with the corners—a standard technique in particle-in-cell (PIC) simulations.

7: Note that this is a simplification compared to [Bar+18], as they also optimize code for the "charge deposit".

The simulation proceeds as follows. At each time step, all the particles are updated. For a given particle, its speed is first updated, based on the value of the acceleration at the position of this particle. Then, the position of the particle is updated, based on its speed. Observe how, in figure 2.6, these updates are described at a high-level of abstraction, using vector operations, as well as a matrix-vector product for computing the interpolation. The auxiliary function `compute_corner_interpolation_coeffs` computes the interpolation coefficients associated with the position of the particle.

Optimized code Figure 2.7 shows our optimized code for the function `simulate`. Two preliminary transformations are applied. First, auxiliary functions are inlined. In particular, the first 14 lines of the loop on `idPart` visible in the optimized code (involving the variables `rX`, `rY`, `rZ`, as well as `cX`, `cY`, `cZ`) correspond to the code inlined from `compute_corner_interpolation_coeffs`, whose implementation was not shown in figure 2.6. Second, the allocation of the array `coeffs`, used to store the interpolation coefficients, is moved outside the loop[8]. Then, two key optimizations are applied.

8: In the full-featured Particle-in-Cell code [Bar+18], the array `coeffs` is entirely eliminated by further optimizations, which generate large-size arithmetic expressions that may then be processed by vector instructions.

First, the vector and matrix operations are replaced with operations over individual fields (named `pos.x`, `pos.y`, `pos.z`, `speed.x`, `speed.y`, and `speed.z`). Moreover, local vector variables are replaced with families of variables (e.g., `fieldAtPos_x`, `fieldAtPos_y`, and `fieldAtPos_z`).

```
void simulate(vect* fieldAtCorners, int32_t nbSteps, double deltaT, double pCharge, double pMass,
    int32_t nbPart, particle* part) {
  const double fieldFactor = deltaT * deltaT * pCharge / pMass;
  vect* const lFieldAtCorners = (vect*) malloc(nbCorners * sizeof(vect));
  for (int32_t i = 0; i < nbCorners; i++) {
    lFieldAtCorners[i].x = fieldAtCorners[i].x * fieldFactor;
    lFieldAtCorners[i].y = fieldAtCorners[i].y * fieldFactor;
    lFieldAtCorners[i].z = fieldAtCorners[i].z * fieldFactor;
  }
  for (int32_t i = 0; i < nbPart; i++) {
    part[i].speed.x *= deltaT;
    part[i].speed.y *= deltaT;
    part[i].speed.z *= deltaT;
  }
  double* const coeffs = (double*) malloc(nbCorners * sizeof(double));
  for (int32_t idStep = 0; idStep < nbSteps; idStep++) {
    for (int32_t idPart = 0; idPart < nbPart; idPart++) {
      const double rX = part[idPart].pos.x;
      const double rY = part[idPart].pos.y;
      const double rZ = part[idPart].pos.z;
      const double cX = 1. - rX;
      const double cY = 1. - rY;
      const double cZ = 1. - rZ;
      coeffs[0] = cX * cY * cZ;
      coeffs[1] = cX * cY * rZ;
      coeffs[2] = cX * rY * cZ;
      coeffs[3] = cX * rY * rZ;
      coeffs[4] = rX * cY * cZ;
      coeffs[5] = rX * cY * rZ;
      coeffs[6] = rX * rY * cZ;
      coeffs[7] = rX * rY * rZ;
      double fieldAtPos_x = 0.;
      double fieldAtPos_y = 0.;
      double fieldAtPos_z = 0.;
      for (int32_t k = 0; k < nbCorners; k++) {
        fieldAtPos_x += coeffs[k] * lFieldAtCorners[k].x;
        fieldAtPos_y += coeffs[k] * lFieldAtCorners[k].y;
        fieldAtPos_z += coeffs[k] * lFieldAtCorners[k].z;
      }
      const double speed2_x = part[idPart].speed.x + fieldAtPos_x;
      const double speed2_y = part[idPart].speed.y + fieldAtPos_y;
      const double speed2_z = part[idPart].speed.z + fieldAtPos_z;
      part[idPart].pos.x += speed2_x;
      part[idPart].pos.y += speed2_y;
      part[idPart].pos.z += speed2_z;
      part[idPart].speed.x = speed2_x;
      part[idPart].speed.y = speed2_y;
      part[idPart].speed.z = speed2_z;
    }
  }
  free(coeffs);
  for (int32_t i = 0; i < nbPart; i++) {
    part[i].speed.x /= deltaT;
    part[i].speed.y /= deltaT;
    part[i].speed.z /= deltaT;
  }
  free(lFieldAtCorners);
}
```

Figure 2.7: Optimized code for the particle simulation case study.

Second, a *scaling* transformation is applied on the data in order to simplify the arithmetic computations that need to be performed at every time step.

To understand how this scaling optimization works, consider a particle. For simplicity, let us focus on its behavior on the x-coordinate. At a given time step, its speed, written v, and its position, written x, are updated according to the formulas: $a{=}qE/m$ and $v \mathrel{+}= a\Delta_t$ and $x \mathrel{+}= v\Delta_t$. Here, E denotes the electric field interpolated at the location of this particle. The constants q, m, and Δ_t corresponds to the program variables `pCharge`, `pMass`, and `deltaT`, respectively. The idea is to store not the values of E and v, but instead the values E' and v' defined as: $E' = qE\Delta_t^2/m$ and $v' = \Delta_t v$. The interest is that the speed and position updates at a given time step are now described using much simpler formulas that avoid the need for computing multiplications: $v' \mathrel{+}= E'$ and $x \mathrel{+}= v'$. To implement this scaling transformation, the components of the field `speed` of the array `part` are multiplied, in-place, by a factor Δ_t before starting the simulation; symmetrically, at the end of the simulation, the values are divided by Δ_t. For the electric field array, which is read-only, the scaling factor is applied on an auxiliary array named `lFieldAtCorners`, obtained by multiplying the values of `fieldAtCorners` by $q\Delta_t^2/m$. (An in-place update would be disallowed because the array `fieldAtCorners` is described using a read-only permission.) By linearity of the interpolation computations, this scaling propagates to the values computed for the electric field at the particle location (`fieldAtPos_x`, `fieldAtPos_y`, and `fieldAtPos_z`). Note that the unoptimized code for this example uses the special OptiC type **`real`** that represents idealized infinitely precise computations instead of regular floating-point numbers and their rounding between each operation. On this type we allow ourselves to perform any kind of arithmetic rewriting that is mathematically true for real numbers. At the last transformation step, we decide to approximate those computations by using the type **`double`** corresponding to IEEE 754 binary64[9]. This corresponds to a standard workflow performed by some code performance experts, where the order of floating point operation was not carefully chosen on the unoptimized code, and numerical stability is checked a posteriori. Formal reasoning about precision in the optimized code is an orthogonal challenge, which we leave to future work.

9: Like in the previous example with integer sizes, this **`real`** type with infinite precision and its lowering to **`double`** is not yet implemented.

Optimization script Figure 2.8 shows our optimization script. Let us describe the key steps. The transformation `Function.inline_multi` inlines auxiliary functions, in particular vector operations. `Record.split_fields` turns record assignment operations into per-field assignment operations. `Variable.insert` inserts a definition for the multiplicative factor $q\Delta_t^2/m$, which is applied to the electric field. `Accesses.scale` (as well as `scale_var` and `scale_mut`) apply the relevant multiplicative factors on the values stored in the various data structures at hand. Crucially, the correctness of the scaling transformation relies on the knowledge that the same arrays are not accessed by means of other (aliased) pointers. The verification of this property relies on the separation logic information computed during typechecking. `Loop.fusion_targets` fuses the several loops that applied per-field scaling. `Variable.unfold` reveals the definition of a variable at certain of its occurrences. `Variable.inline` eliminates a variable definition, replacing all occurrences with the definition. `Loop.hoist_alloc` pulls the allocation of the `coeffs` array outside the loop. `Cleanup.std` applies final simplifications, as previously explained.

Benefits of using OptiTrust Applied mathematicians commonly write optimized code such as that of figure 2.7 by hand. Revealing the x, y and z

```
let ctx = cFunBody "simulate_single_cell" in
let find_var n = trm_find_var n [ctx] in
let vect = typ_find_var "vect" [ctx] in
let particle = typ_find_var "particle" [ctx] in
let dims = ["x"; "y"; "z"] in
Matrix.local_name_tile ~var:"fieldAtCorners"
  ~elem_ty:vect ~uninit_post:true ~mark_load:"loadField"
  ~local_var:"lFieldAtCorners" [ctx; cFor "idStep"];
Function.inline_multi [ctx; cCalls ["cornerInterpolationCoeff"; "matrix_vect_mul"; "vect_add"; "vect_mul"]];
Variable.inline_and_rename [ctx; cVarDef "fieldAtPos"];
Record.split_fields ~typs:[particle; vect] [tSpanSeq [ctx]];
Record.to_variables [ctx; cVarDefs ["fieldAtPos"; "pos2"; "speed2"; "accel"]];
let deltaT = find_var "deltaT" in
Variable.insert ~name:"fieldFactor" ~value:(trm_mul (trm_mul deltaT deltaT) (trm_exact_div (find_var "pCharge")
  (find_var "pMass"))) [ctx; tBefore; cVarDef "lFieldAtCorners"];
let scaleFieldAtPos d =
   Accesses.scale_var ~factor:(find_var "fieldFactor") [nbMulti; ctx; cVarDef ("fieldAtPos_" ^ d)] in
List.iter scaleFieldAtPos dims;
let scaleSpeed2 d = Accesses.scale_immut ~factor:deltaT [nbMulti; ctx; cVarDef ("speed2_" ^ d)] in
List.iter scaleSpeed2 dims;
let scaleFieldAtCorners d =
  let address_pattern = Trm.(struct_access (array_access (find_var "lFieldAtCorners") (pattern_var "i")) d) in
  Accesses.scale ~factor:(find_var "fieldFactor") ~address_pattern ~uninit_post:true
    [ctx; tSpan [tBefore; cMark "loadField"] [tAfter; cFor "idStep"]] in
List.iter scaleFieldAtCorners dims;
let scaleParticles d =
  let address_pattern =
    Trm.(struct_access (struct_access (array_access (find_var "part") (pattern_var "i")) "speed") d) in
  Accesses.scale ~factor:deltaT ~address_pattern ~mark_preprocess:"partsPrep" ~mark_postprocess:"partsPostp"
    [ctx; tSpanAround [cFor "idStep"]]; in
List.iter scaleParticles dims;
List.iter Loop.fusion_targets [[cMark "partsPrep"]; [cMark "partsPostp"]];
Variable.unfold ~at:[cFor "idStep"] [cVarDef "fieldFactor"];
Variable.inline [ctx; cVarDefs (Tools.cart_prod (^) ["accel_"; "pos2_"] dims)];
Arith.(simpls_rec [expand; gather_rec]) [ctx];
Loop.hoist_alloc ~indep:["idStep"; "idPart"] ~dest:[tBefore; cFor "idStep"] [cVarDef "coeffs"];
Cleanup.std ();
```

Figure 2.8: Optimization script for the particle simulation case study.

coordinates triples the size of the code, and applying a scaling transformation by hand is a highly error-prone task. The aim of OptiTrust is to provide them with an alternative route, more productive and more trustworthy. As we have already explained, for each of the transformations being applied, OptiTrust exploits the separation logic invariants to check criterions that guarantee that the transformations preserve the semantics of the code.

2.3 The matrix-multiply case study

TVM [Che+18] is the state-of-the-art, industrial-strength, interactive compiler for machine learning. The TVM tutorial presents an optimization script[10] (a.k.a. *schedule*) for optimizing a matrix multiplication function, specialized for square matrices of size 1024. This script has been carefully tuned to produce code optimized for specific Intel CPUs. On a 4-core Intel i7-8665U CPU with AVX2 support, the TVM experts thereby achieve a speedup of 150× over a totally naive, sequential implementation of matrix multiplication.[11] The aim of this third case study is to demonstrate the ability of OptiTrust to produce code that matches the performance delivered by TVM. More precisely, we show that we are able to generate code that features the exact same optimization patterns as in the TVM case study, with a reasonably short transformation script.

Moreover, we show on this example that OptiTrust is able to work with and

[Che+18]: Chen et al. (2018), *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*

10: https://github.com/apache/tvm/blob/v0.19.0/gallery/how_to/optimize_operators/opt_gemm.py

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

```
void mm(float* C, float* A, float* B, int m, int n, int p) { // naive matrix-multiply
  __reads("A ⇝ Matrix2(m, p), B ⇝ Matrix2(p, n)");
  __writes("C ⇝ Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __xwrites("for j in 0..n → &C[MINDEX2(m, n, i, j)] ⇝ Cell");
    for (int j = 0; j < n; j++) {
      __xwrites("&C[MINDEX2(m, n, i, j)] ⇝ Cell");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __ghost_begin(focusA, matrix2_ro_focus, "A, i, k");
        __ghost_begin(focusB, matrix2_ro_focus, "B, k, j");
        sum += A[MINDEX2(m, p, i, k)] * B[MINDEX2(p, n, k, j)];
        __ghost_end(focusA);
        __ghost_end(focusB);
      }
      C[MINDEX2(m, n, i, j)] = sum;
    }
  }
}
void mm1024(float* C, float* A, float* B) { // specialization to 1024x1024 matrices
  __reads("A ⇝ Matrix2(1024, 1024), B ⇝ Matrix2(1024, 1024)");
  __writes("C ⇝ Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}
```

Figure 2.9: Unoptimized OptiC code for the matrix-multiply case study, using shape-only resource annotations.

preserve annotations that can have various levels of details, giving different kind of guarantees on the output.

2.3.1 With shape-only annotations

First, like the two previous examples, we start with annotations that specify only the shape of data behind pointers.

Unoptimized code Figure 2.9 shows the unoptimized and annotated matrix-multiply code that we take as input. Note that contrary to the previous case studies, we directly use the type `float` for matrix cells, which means that the optimized code will preserve the rounding behavior of the initial code. Also note that, here again, some annotations could be inferred automatically with additional tooling.

Optimized code TVM output code is expressed not as C code, but directly in the intermediate representation of LLVM. We manually inspected the TVM schedule, intermediate representation, and LLVM IR output to infer what C code we should generate. The code we produce using OptiTrust is shown in figure 2.10. Compared with the naive code from figure 2.9, the optimized code from figure 2.10 integrates 7 key optimizations:

1. The body of the generic matrix multiplication function `mm` is specialized to the size `1024`.

2. An auxiliary matrix named `pB` is allocated to store the coefficients of the matrix `B` in a different order (creating a partial transpose of `B`). The introduction of this auxiliary matrix induces a cost for the initial copy, but then greatly improves the memory access patterns.

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

Figure 2.10: Our optimized code for the matrix-multiply case study. This code features the same optimization patterns as the reference output of TVM.

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

Figure 2.11: Optimization script for the matrix-multiply case study.

3. The matrices are processed by blocks of size 32: each loop over a range of size 1024 is replaced with 2 loops each of range 32. Blocking improves locality in matrix-multiply.

4. Results are not accumulated into a scalar accumulator, but instead into a stack-allocated array named `sum` of size 32×32 that contains all scalar accumulators for a block.

5. Around the inner vectorized loops, the locally relevant row of `sum` is promoted to a smaller array `s` that can be mapped onto a few 256-bit vector registers. On every `i` iteration, two `memcpy` operations are used for synchronizing `s` with `sum`.

6. The various loops are reordered in a particular manner, both to improve cache locality and to enable parallelization. The outermost loops are executed in parallel by several cores. The instructions of the inner loop are parallelized by means of SIMD operations.

7. The 4 loops tagged as `#pragma omp simd` in figure 2.10 are very similar. However, if we attempt to factorize them into a loop with 4 iterations, then Intel's compiler (ICX) produces slower code. Unfolding the loops as shown makes relying on unrolling heuristics unnecessary.

Again, this particular set of optimizations directly comes from the TVM case study. We demonstrate how to reproduce the same optimizations using OptiTrust.

Optimization script Figure 2.11 shows our optimization script, which consists of only 10 lines. Internally, though, the high-level transformations mentioned in the script trigger the application of 55 basic transformations and 61 transformations that only change annotations. An illustrative example is the call to `Loop.reorder_at` on Line 4 of figure 2.11. This combined transformation takes as argument a specific instruction (referred to as "an instruction of the form +=") as well as a description of the desired order for the loops that surround this instruction (the list `["bi"; "bj"; "bk"; "i"; "k"; "j"]`). The `reorder` transformation iteratively "brings down" the loops that need to be swapped closer to the instruction, starting from the innermost loops, and processing the loops until the outermost one. The call to `reorder_at` in our script triggers a total of 4 *loop swaps*, 6 *loop fissions*, and 2 *loop hoist* operations. In particular, the effect of these 2 hoist operations is to turn local variable named `sum` in figure 2.9 into the 2D-array named `sum` in figure 2.10.

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

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

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

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

Figure 2.12: TVM case study for matrix-multiply. On the left, input code in TVM's domain specific language. On the right, TVM optimization script (a.k.a. *schedule*). Both use Python syntax.

Comparison against TVM The TVM matrix-multiply case study appears in figure 2.12. We only comment on specific aspects and refer to TVM's tutorial for further details. In TVM, input programs are written in a domain-specific language embedded in Python. Ideally, the matrix-multiply program shown on the left-hand side of figure 2.12 would replace the definitions of `pB` and `C` with a simpler definition of `C`:

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

Yet, TVM is unable to express the introduction of the transposed matrix of `B`, named `pB`, as a code transformation. The programmer therefore needs to introduce this auxiliary structure manually in the input code. Likewise, the blocking strategy needs to be hardwired in the source code on the left-hand side of figure 2.12. In contrast, our input program for matrix multiplication shown in figure 2.9 builds upon familiar C-like syntax and, most importantly, includes no optimization. Starting from totally unoptimized reference code improves readability, trustworthiness, and maintainability. Besides, although our input code for matrix-multiply is currently expressed using explicit loops, in the future we could alternatively express it using higher-order combinators as well.

The right-hand side of figure 2.12 shows TVM's optimization script. Our optimization script shown in figure 2.11 is not much more verbose than that of TVM. TVM does not produce C code but directly outputs LLVM IR code. However, we have carefully checked that the code produced using OptiTrust features the same optimizations and delivers the same performance as the LLVM IR code produced using TVM (as shown in table 2.1).

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

Table 2.1: Benchmark of the performance of a single 1024×1024 matrix multiplication over 200 runs on a 4-core Intel i7-8665U CPU with AVX2 support for different implementations. First column shows the median across 200 runs, second column shows the 90th percentile. Both OptiTrust and TVM reach the same performance that corresponds to a 150× speedup compared to the naive algorithm. (This benchmark's data was collected by Thomas Kœhler.)

Finally, let us comment on interactivity. Guided by all the contents from figure 2.12, TVM applies a monolithic compilation pass to produce optimized code. TVM does not provide interactive, easily-readable feedback for the transformations performed. In contrast, OptiTrust applies a series of local, source-to-source transformations, directly manipulating OptiC programs. Moreover, it provides human-readable diffs for every step and every substep involved in the optimization process.

2.3.2 With full functional correctness annotations

Let us use the same example for illustrating one extra feature of OptiTrust: its ability to preserve a proof of functional correctness through code transformations. Usually, obtaining code that is at the same time competitively

```
__DECL(reduce_sum, "int * (int → float) → float");
__AXIOM(reduce_sum_empty, "forall (f: int → float) → 0.f =. reduce_sum(0, f)");
__AXIOM(reduce_sum_add_right, "forall (n: int) (f: int → float) (_: n >= 0) → reduce_sum(n, f) +.
    f(n) =. reduce_sum(n + 1, f)");
__DEF(matmul, "fun (A B: int * int → float) (p: int) → fun (i j: int) → reduce_sum(p, fun k → A
    (i, k) *. B(k, j))");
```

Figure 2.13: Axiomatization of iterated sum and definition of abstract matrix multiplication

optimized and formally verified is a task that requires too much effort to be performed. This case study demonstrates that we can use OptiTrust to reduce the amount of effort needed thanks to transformations that preserve such a proof a functional correctness.

Axiomatization of matrix multiplication Before showing how we preserve functional correctness properties across transformations, let us comment on how the user provides an initial proof of functional correctness about the unoptimized version of their source code. In OptiTrust, this initial proof takes the form of annotations that are very similar to the shape-only annotations we saw before, but are more precise. Those annotations are then checked by our separation logic typechecker, and serve as the base for justifying transformation correctness.

To add this extra precision, we provide separation logic predicates extended with *models*. The most basic predicate $p \mapsto v$ states that p is a pointer to a value modeled by v. For matrices, we introduce an extended version of the predicate `Matrix2` with one additional argument. A resource `a` $\rightsquigarrow$ `Matrix2(m,n,A)` states that a is a pointer to an $m \times n$ matrix which contains at index (i, j) the value modeled by $A(i, j)$.

In practice, to express the functional specification, we start with a definition of what a matrix multiplication is. Figure 2.13 shows how we write this definition as OptiC annotations. In this example, we decided to axiomatize iterated sums with the `reduce_sum` operator and two rewriting rules `reduce_sum_empty` and `reduce_sum_add_right`, and define the matrix multiplication `matmul` as an application of this operator at each cell.

One important design decision to notice is the fact that matrices are modeled by functions taking two indices and returning one number, effectively creating infinite abstract matrices. These infinite matrices are easier to work with in specifications since we do not need to ensure that indices are in range when using the model. Observe how this design influence the arguments of `matmul` that takes the model of two matrices `A` and `B` and the size of the common dimension `p` along which the multiplication is performed, and return the infinite matrix that corresponds to the multiplication.

Unoptimized annotated code With these definitions, we can annotate the code of matrix multiplication like in figure 2.14. Compared to the simpler version without models in figure 2.9, the extra work for full functional correctness invariants consists in specifying values in function and loop contracts and inserting a few more ghost rewriting operations.

The matrix multiplication function itself is annotated with a **`__requires`** clause that declares two unspecified matrix models `A` and `B`, a **`__reads`** clauses that assert that `a` and `b` point to matrices modeled by respectively `A` and `B`, and a **`__writes`** clause that assert that `c` is a matrix that will be completely overwritten with a matrix modelled by the result of the matrix multiplication $\texttt{A} \cdot \texttt{B}$. Then, the loops over `i` and `j` are annotated with a

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

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

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

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

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

        __ghost(in_range_bounds, "k", "k_gt_0 <- lower_bound");
        __ghost(rewrite_float_linear, "inside := fun v → &sum ↦ v,
          by := reduce_sum_add_right(k, fun k → A(i, k) *. B(k, j), k_gt_0)");
      }

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

Figure 2.14: Unoptimized OptiC code for matrix multiplication, with full functional correctness annotations.

`__xwrites` annotation asserting that each iteration writes the correct values in the output matrix row or cell. Finally, the loop over `k` is annotated with a `__spreserves` annotation (the prefix `s` here stands for "sequentially") asserting that between each loop iteration, the variable `sum` contains $\sum_{k_0=0}^{k} A(i, k_0) \cdot B(k_0, j)$.

Compared to the initial code with shape-only annotations described in figure 2.9, we need to add some ghost operations that rewrite models in presence of functional correctness specifications. The ghost instruction `rewrite_float_linear` rewrites a floating point expression inside a resource by using an equality. In this case study, those equalities directly stem from the axioms declared for iterated sums (recall figure 2.13). In order to use the `reduce_sum_add_right` axiom, we need to provide a proof that $k \geq 0$. This fact stems from the fact that k is a loop index, and therefore $k \in 0..p$. The ghost operation `in_range_bounds` allows extracting inequalities from a range inclusion. Moreover, the ghost instructions that focus matrix cells which were already present with shape-only specifications, are still needed in the full functional correctness setup.

Optimization script To optimize such a program we reuse the optimization script seen in figure 2.11. In OptiTrust, most transformations work transparently with annotations featuring any level of detail, and preserve that level of detail. Indeed, in this example, full functional correctness annotations are preserved between each transformation step and can be used for

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

Figure 2.15: Excerpt of our optimized code for matrix multiplication with full functional correctness annotations. This excerpt contains annotated generated loops to pre-transpose the matrix `b` in `pB` before computation.

typechecking.

An important point to notice is that at any point during the optimization script, our typechecker ensures that annotations are enough to deduce that the function specification holds. Since in full functional correctness the function specification is complete, knowing that such function specification hold is enough to ensure that our transformed code still computes the same matrix multiplication. In particular, the implementation of the transformations themselves need not be trusted, since the typechecker alone can guarantee that the code respects its complete specification. In other words, OptiTrust full functional correctness annotations form a proof of functional correctness, and transformations are preserving this proof.

Resulting optimized code After the transformations the produced code is a fully annotated program which typechecks. Two excerpts are reproduced in figures 2.15 and 2.16. You may notice that our transformation script is able to find invariants for all the generated loops with relevant models, and it also adds all the ghost operations that correspond to the required proof steps to establish those invariants. For instance, the lines starting with `__ASSERT` statically check arithmetic facts needed later, the ghost operations `tile_index_in_range` establish that an index expression introduced by a tiling operation is in a given range, which is required for checking array access bounds, and the ghost operation `rewrite_int_linear` performs an arithmetic rewriting needed for syntactically matching the loop invariant.

Notice that the annotations in the optimized code are based on the initial ghost operations placed by the user. For instance, the ghost pair

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

Figure 2.16: Excerpt of our optimized code for matrix multiplication with full functional correctness annotations. This excerpt corresponds to the last of the four inner-loops accumulating in the local variable `s`.

`focusA3` and the ghost operation `rewrite_float_linear` in figure 2.16 respectively comes from the ghost pair `focusA` and the second occurrence of `rewrite_float_linear` in figure 2.14 with specialized indices.

Overall, we believe that this proof preserving workflow reduces significantly the amount of work needed to formally verify a matrix multiplication implementation that matches the performance of TVM. Indeed, the initial proof of correctness represented by full functional correctness annotations, is performed on the very simple unoptimized code. Then, all proof elements such as loop invariants or rewriting steps needed because the optimized code is more complex are automatically inserted by the transformation themselves.

2.4 Comparison of OptiTrust with other interactive compilers

Now that we have given a tour of the features of the OptiTrust framework, let us introduce a number of qualitative properties, before reviewing related tools for interactive compilation and finally explaining why OptiTrust achieves a unique combination of features.

- **Generality**: How large is the domain of applicability of the tool?
- **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?
- **Modularity** of analyses: for transformations whose correctness depends on a code analysis, can the tool deal with specifications that summarize the effects of each function, or are all functions inlined during the analyses?
- **Trustworthiness**: Does the tool ensure that user-requested transformations preserve the semantics of the code? Can it moreover provide mechanized proofs?

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

2.4.1 Evaluation of other interactive compilers

This section summarizes the properties of existing approaches, highlighting their diversity.

Halide [Rag+13] is an industrial-strength domain-specific compiler for image processing, used e.g. to optimize code running in Photoshop and YouTube. Halide popularized the idea of separating an *algorithm* describing what to compute from a *schedule* describing how to optimize the computation. This separation makes it easy to try different schedules. TVM [Che+18] is a tool directly inspired by Halide, but tuned for machine learning applications; it is used by most of the major CPU/GPU manufacturers. Other tools inspired by Halide include Fireiron [Hag+20a], used at Nvidia, as well as PartIR [Ala+24], used at Google. All these tools do not support higher-order composition of transformations, and are not easily extensible [Rag23; BI19]. Moreover, understanding their output is difficult as the applied transformations are not detailed to the user, even though interactive scheduling systems have been proposed to mitigate this difficulty [Ika+21].

Elevate [Hag+20b] 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 [Rag23]. Elevate strategies are applied on programs expressed in a functional array language named Rise, followed by compilation to imperative code. The use of a functional array language greatly simplifies rewriting, however it restricts applicability and makes controlling imperative aspects difficult (e.g. memory reuse).

Exo [Ika+22] is an imperative DSL embedded in Python, geared towards the development of high-performance libraries for specialized hardware. The strength of Exo lies in externalizing target-specific code generation to user-level code instead of compilation backends. Exo programs can be optimized by applying a series of source-to-source transformations. These transformations are described in a Python script, with a cursor mechanism for targeting code points [Ika+25]. The user can add custom transformations,

[Rag+13]: Ragan-Kelley et al. (2013), *Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines*

[Che+18]: Chen et al. (2018), *TVM: An Automated End-to-End Optimizing Compiler for Deep Learning*

[Hag+20a]: Hagedorn et al. (2020), *Fireiron: a data-movement-aware scheduling language for GPUs*

[Ala+24]: Alabed et al. (2024), *PartIR: Composing SPMD Partitioning Strategies for Machine Learning*

[Rag23]: Ragan-Kelley (2023), *Technical Perspective: Reconsidering the Design of User-Schedulable Languages*

[BI19]: Barham et al. (2019), *Machine Learning Systems are Stuck in a Rut*

[Ika+21]: Ikarashi et al. (2021), *Guided Optimization for Image Processing Pipelines*

[Hag+20b]: Hagedorn et al. (2020), *Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies*

[Rag23]: Ragan-Kelley (2023), *Technical Perspective: Reconsidering the Design of User-Schedulable Languages*

[Ika+22]: Ikarashi et al. (2022), *Exocompilation for productive programming of hardware accelerators*

[Ika+25]: Ikarashi et al. (2025), *Exo 2: Growing a Scheduling Language*

possibly defined by (higher-order) composition. A major limitation of Exo is that it is restricted to static control programs[12], with quasi-affine array indexing[13]. Another important limitation of Exo is that the transformations are performed on code in which all functions are inlined. This approach, which lacks modularity, may harm scalability to larger or more complex programs. Because Exo code almost syntactically translates to C code, its code generation stage makes few decisions beyond user control, and we assume that understanding the output of Exo is relatively easy. Similarly to Elevate rewrite rules, extending Exo transformations is easier since the definition and correctness of each transformation is independent from other ones.

12: Static control means that the control flow cannot depend on data stored in arrays

13: Quasi-affine indexing means that the index of array accesses must be a linear combination of loop variables and constants with division and modulo by constants allowed

Clay [Bag+16] is a framework to assist in the optimization of loop nests that can be described in the *polyhedral model* [Fea92]. The polyhedral model only covers a specific class of loop transformations, with restriction over the code contained in the loop bodies, however it has proved extremely powerful for optimizing code falling in that fragment. Clay provides a decomposition of polyhedral optimizations as a sequence of basic transformations with integer arguments. The corresponding transformation script can then be customized by the programmer. Clint [ZHB18] adds visual manipulation of polyhedral schedules through interactive 2D diagrams. LoopOpt [Che+21] 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.

[Bag+16]: Bagnères et al. (2016), *Opening Polyhedral Compiler's Black Box*

[Fea92]: Feautrier (1992), *Some efficient solutions to the affine scheduling problem: one dimensional time*

[ZHB18]: Zinenko et al. (2018), *Visual Program Manipulation in the Polyhedral Model*

[Che+21]: Chelini et al. (2021), *LoopOpt: Declarative Transformations Made Easy*

ATL [Liu+22] is a purely functional array language for expressing Halide-style programs. Its particularity is to be embedded into the Rocq proof assistant. ATL programs can be transformed through the application of rewrite rules expressed as Rocq theorems. With this approach, transformations are inherently accompanied by machine-checked proofs of correctness. The set of rules includes expressive transformations, some beyond the scope of Halide, and can be extended by the user. Once optimized, ATL programs are then compiled into imperative C code. Like Rise, generality and control are restricted by the functional array language nature of ATL.

[Liu+22]: Liu et al. (2022), *Verified Tensor-Program Optimization via High-Level Scheduling Rewrites*

Alpinist [Sak+22] is a *pragma*-based tool for optimizing GPU-level, array-based code. It is able to apply basic transformations such as loop tiling, loop unrolling, data prefetching, matrix linearization, and kernel fusion. The key characteristic of Alpinist is that it operates over code formally verified using the VerCors framework [Blo+17]. 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.

[Sak+22]: Sakar et al. (2022), *Alpinist: An Annotation-Aware GPU Program Optimizer*

[Blo+17]: Blom et al. (2017), *The VerCors Tool Set: Verification of Parallel and Concurrent Software*

Clava [BC20] 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 [Sil+19] for optimizing specific programs. LARA allows expressing user-guided transformations by combining declarative queries over the abstract syntax tree and imperative invocations of transformations, with the option to embed JavaScript code. The application paper on the Pegasus tool [Pin+20] illustrates this approach on loop tiling and interchange operations.

[BC20]: Bispo et al. (2020), *Clava: C/C++ source-to-source compilation using LARA*

[Sil+19]: Silvano et al. (2019), *The ANTAREX domain specific language for high performance computing*

[Pin+20]: Pinto et al. (2020), *Pegasus: Performance Engineering for Software Applications Targeting HPC Systems*

2.4.2 The unique features of OptiTrust

When considering the aforementioned criterions and tools, OptiTrust achieves a unique combination of features. To our knowledge, Exo is

currently one of the best tool regarding expressiveness, control, feedback, extensibility, and composability for the static control programs it can handle. We think that the current version of OptiTrust is comparable to Exo on those criterions. However, OptiTrust can handle more complex control flow, but currently does not handle GPU architectures supported by Exo. At the same time, we think that OptiTrust reaches the level of Alpinist for modularity, and trustworthiness, by using the same idea of transforming annotations along with the code. To our knowledge, Alpinist is one of the best tools for modularity, and only ATL performs better on trustworthiness by providing foundational proofs of transformation correctness.

Generality As pointed out in section 1.7, this release of OptiTrust has a number of limitations: it supports a relatively limited imperative language, it does not yet support defining custom separation logic representation predicates, and there remains many useful transformations to implement. Thus, OptiTrust in its current form does not yet demonstrate full generality. Yet, every aspect of OptiTrust has been designed towards that goal.

Expressiveness OptiTrust supports a number of basic transformations that, taken individually, might appear relatively straightforward. However, by chaining such transformations in the desired manner, the OptiTrust user is able to achieve state-of-the-art high-performance code, similar to what an expert might have written by hand.

Let us summarize the transformations currently supported in OptiTrust. For instruction-level transformations, we support: function inlining, constant propagation, common subexpression elimination, instruction reordering, switching between stack and heap allocation, and basic arithmetic simplifications. For control-flow transformations, we support: loop interchange, loop tiling, loop fission, loop fusion, loop-invariant code motion, loop unrolling, loop deletion and loop splitting. For data layout transformations, we support: interchange of dimensions of an array, and array tiling.

Expressiveness also depends on the generality of the correctness criterions associated with every transformation. In practice, with shape-only or incomplete functional correctness annotations, there could be situations where the user might want to legitimately apply a basic transformation, yet OptiTrust's implementation is unable to recognize this application as correct. One option is for the user to treat this particular step as "user-trusted", and to rely on human review of the diff associated with that step. For some transformations, OptiTrust features correctness criterions that may require complex reasoning steps and generate proof obligations. In that case, those proof obligations must either be proved correct, or be admitted by the user. If OptiTrust is unable to automatically verify a transformation, a third option for the user is to enrich the annotations with full functional correctness invariants. With such full functional correctness invariants, replacing any piece of code with another piece of code that provably satisfies the same specification is correct by definition.

Control Transformation scripts in OptiTrust empower the user with very fine-grained control over how the code should be transformed. A challenge when providing such level of control is to keep transformation scripts concise. To that end, OptiTrust provides high-level *combined* transformations, effectively recipes for combining the *basic* transformations provided by OptiTrust. The matrix multiplication case study presented the example of `Loop.reorder`, 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.

Moreover, OptiTrust is a source-to-source transformation framework. Therefore, there is no monolithic code generation phase that can make arbitrary choices for the user at the end of the transformation script.

Feedback For each step in the transformation script, OptiTrust delivers feedback in the form of human-readable OptiC syntax. 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. Full traces can also be very useful for third-party reviewing of an optimization process. Besides, a key feature of OptiTrust is its fast feedback loop. The production of fast, human-readable feedback in a system with significant control is reminiscent of interactive proof assistants, and of the aforementioned ATL tool [Liu+22].

[Liu+22]: Liu et al. (2022), *Verified Tensor-Program Optimization via High-Level Scheduling Rewrites*

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

Modularity The separation logic contract provided by the programmer for a function `f` constitutes a complete summary of the side effects that this function may perform. Hence, when a transformation operates on a piece of code that contains a call to `f`, the analysis involved in checking the correctness of that transformation needs not traverse the implementation of `f`. In that sense, all our analyses, including the typechecking process, are modular. This modularity has numerous benefits. First, it implies that one may change the implementation of `f` without invalidating the optimization script associated with another function `g`, provided that the optimization

of `g` was not relying on an inlining of the function `f`. Second, it means that analyses can much more easily scale up to larger and more complex programs, without computation costs blowing up. Third, it makes it easier to devise clearer, more concise error messages. Indeed, in a modular system, errors depend solely on local information.

In compiler design in general, there exists a tension between modularity and optimizations, because certain key optimizations need to be applied across abstraction barriers. OptiTrust handles this tension by leaving it up to the user to decide where functions should be inlined—thereby deciding on a per-need basis where modularity should be given up to the benefits of performance.

Also note that annotating a function with (full or incomplete) functional correctness properties can reduce the need for inlining functions by revealing interesting properties at the calling site.

Trustworthiness OptiTrust leverages separation logic resource information to check the correctness of transformations before applying them. In practice, all the built-in OptiTrust transformations return an error if they cannot check the correctness of their own application.

However, like all compilers, interactive optimization tools like OptiTrust are highly subject to implementation bugs. To analyze in more detail the level of trustworthiness of OptiTrust we need to distinguish two cases: the optimization of functions with shape-only or incomplete functional correctness specifications on one hand, and the optimization of functions with full functional correctness annotations on the other hand.

With shape-only or incomplete functional correctness annotations, OptiTrust mitigates the risks of transformation implementation bugs producing incorrect code in two ways. First, the diff of every step can be thoroughly scrutinized, and unit tests are checking the most common cases of transformations. Second, as explained in the introduction of this chapter, we have organized the OptiTrust code base in such a way as to isolate the implementation of the *basic* transformations, which consist of transformations that directly modify the AST. Only basic transformations need to be trusted. We have been careful to systematically minimize the complexity of the interface and of the implementation of our basic transformations. All other transformations—the *combined* transformations—are *not* part of the trusted computing base (TCB).

With full functional correctness annotations, none of the transformations are part of the TCB. Indeed, program annotations can be viewed as a proof of correctness of the algorithm with respect to its annotated specification, which is valid whenever the program typechecks. Since transformations preserve annotations, OptiTrust can be viewed as a transformation framework preserving a proof of correctness in parallel with the code modifications. By definition, full functional correctness contracts fully characterize the expected properties of the manipulated algorithm, therefore checking that such contract holds is enough to guarantee correctness. Therefore, with full functional correctness annotations, a chain of transformations for a function is correct if the optimized code typechecks with the same external contract as the unoptimized code. In that case, OptiTrust's TCB consists only of our typechecker which implements the logical rules described in chapter 4. In the future, we hope to reduce this TCB even further by allowing to export annotated programs as foundational proofs embedded in a proof assistant (such as Rocq) equipped with a separation logic framework (such as Iris).

This completes our high-level presentation of the OptiTrust framework. The rest of this manuscript presents the implementation of OptiTrust: its internal AST, its typechecking algorithm, and its transformations.

Syntax and semantics in OptiTrust

3

As illustrated in the previous chapter, in OptiTrust, input programs are written in OptiC (the targeted subset of C, augmented with annotations), and the same OptiC language is used to report the diff associated with every transformation in terms of a concise syntax familiar to the programmer.

One difficulty for performing transformations on C-like languages, is that they extensively use the concept of *left-value*. Expressions that are in *left-value position*, like the subterm of an address-of operator (&) or on the left of an assignment (e.g. = or +=) do not have the same semantics as if they were in *right-value position*. Indeed, in left-value position, the expression evaluates to its address and not to its value. This difference of semantics of expressions in left-value position creates a burden for transformations that need to handle differently expressions depending on whether they appear as left- or right-values. Moreover, C-like languages feature constructions that are redundant semantic-wise (such as `if` statement versus ternary operator) but those can help a user navigate through the code.

We see here that there is a tension between the readability of user-facing code and the desired simplicity of compiler intermediate representations. Traditional fully automatic compilers avoid these issues by lowering the code into simpler internal representations that are better suited for transformations. This chapter shows that, even if we need to output readable OptiC code at each step, we can still manipulate a simpler intermediate representation, and translate back into readable OptiC whenever needed. With this strategy, OptiC is reserved for user interactions, and a second language, called Optiλ, is used internally by all code transformations. The language Optiλ is an imperative λ-calculus with a special annotation feature that enables a better round-trip translation to OptiC.

Another potential benefit of separating the internal language is that we could more easily add support for other user-facing languages. Indeed, with a different user-facing language, the implementation of the transformations, which operate on Optiλ, would remain the same.

This chapter presents both languages OptiC and Optiλ, and the bidirectional translation that exists between the two.

In section 3.1, we give an overview of how our bidirectional translation works on some examples. Such a translation is relatively standard: C compilers generally include a phase that eliminates mutable variables and *l*-values. The specificity of our translation is that it attaches annotations on certain subterms to allow computing the reciprocal translation. In section 3.2, we present a formal definition of the internal Optiλ language, along with its key design elements, and the underlying call-by-value semantics. Optiλ constructs appear throughout the rest of the manuscript, from the statement of the typing rules to the description of the transformations. Symmetrically, in section 3.3, we describe the user-facing OptiC language and its associated semantics. In section 3.4, we give the rules for the translation from OptiC to Optiλ. In section 3.5, we give the rules for the reverse translation from Optiλ to OptiC, and describe the round-trip property for OptiC program.

3.1 Overview of the internal encoding process

This section gives an overview on the design choices for the internal language Optiλ and for the bidirectional translation required to convert between Optiλ and OptiC.

Optiλ is an imperative λ-calculus, which crucially does not have a notion of left-values. To ensure that, in Optiλ, all directly accessible variables are immutable. Then, mutability is encoded by explicit accesses through pointers. This explicit pointer manipulation must be introduced and removed by the bidirectional translation. Let us present how this bidirectional translation works on a few examples.

Translation with pure variables only We start with a simple function `norm2`, that does not include any mutable variable. Immutable variables can be encoded as a simple **let** binding when their address is never taken, as shown in the following example. For the purpose of typechecking and of computing reverse translations, the **let** bindings introduced by the translation carry the type of the bound variable. Such types appear as subscript in the example Optiλ code below. Remember that the translation is bidirectional, so given only the imperative λ-calculus term on the right, our tool is capable of reproducing the exact same OptiC program on the left.

```
int norm2(int x, int y) {
  const int xsq = x * x;
  const int ysq = y * y;
  const int res = xsq + ysq;
  return res;
}
```

```
let norm2 = fun(x : int, y : int) ↦ {
  let_int xsq = mul(x, x);
  let_int ysq = mul(y, y);
  let_int res = add(xsq, ysq);
  res
};
```

On the example above, the `return` instruction that appears at the end of the body of the function is translated into a terminal value at the end of a chain of **let** bindings. As we explain later, in the syntax of our internal imperative λ-calculus, we exploit *n*-ary sequences instead of cascades of **let in** constructs. Doing so makes it easier for programmers to target spans of contiguous instructions, and simplifies the implementation of numerous transformations.

We say that a variable is *pure* if its definition is translated into a plain **let** binding. Technically, a variable `x` can only be *pure* if there is no assignment operation on `x` and if the address of the variable `x` is never computed via the operator `&x`. Equivalently, a variable `x` can be *pure* if and only if `x` could have been declared with the modifiers `const register`, in the terminology of the C standard.

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

Translation with impure variables Let us now present an example involving impure variables. The function `norm2Acc`, shown below, computes the same value as `norm2`, yet using a mutable accumulator named `acc`.

```
int norm2Acc(int x, int y) {
  int acc;
  acc = x * x;
  acc += y * y;
  return acc;

}
```

```
let norm2Acc = fun(x : int, y : int) ↦ {
  let_ptr(int) acc = stackAlloc_UninitCell_int ();
  set(acc, mul(x, x));
  inplaceAdd(acc, mul(y, y));
  let_int res = get(acc);
  res
};
```

In that case, our translation replaces the mutable variable `acc` with the address to the stack space it occupies. The definition of the variable `acc` is replaced by an explicit allocation of a cell on the stack materialized as a call to $\text{stackAlloc}_{\text{UninitCell}}$. Then, all functions that modify a mutable variable such as set or inplaceAdd take the address of the mutable variable as argument. When the value of a mutable variable is read, such as in the return statement, our translation inserts an explicit get operation. For reasons we explain later, our internal language syntactically only allows using a variable as the result of a sequence, and therefore the translation of the return adds a new pure binding on a variable named `res`.

Figure 3.1 presents additional examples illustrating our translations. The lines involving x and z summarize the treatment of pure and impure variables. The lines involving a illustrate a heap allocated variable. A read operation `*a` is encoded as the function call get(a), and an assignment `*a = v` is encoded as set(a, v). Thus, heap-allocated variables and impure stack-variables are treated essentially the same way in our internal λ-calculus—with the main difference that stack-allocated variables are implicitly deallocated. The name of the primitive operation, whether stackAlloc or heapAlloc, is used to guide

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

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

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

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

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

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

Figure 3.1: Example translations from C code into the OptiTrust's internal AST. We suppose that a function `void f(int)` is defined. We also suppose that variables marked as `const` are never modified and that their address is never taken.

```
int* const t = malloc(9 * sizeof(int));
t[1] = t[1] + 2;
f(t[1])
free(t);

int* u = malloc(9 * sizeof(int));

free(u);
u = malloc(8 * sizeof(int));
u[1] = u[1] + 2;

int v[9];
t[1] = v[1] + 2;
int* const a = &v[1];
f(*a);

int* const m = malloc(MSIZE1(9) * sizeof(int));
m[MINDEX1(9,1)] = m[MINDEX1(9,1)] + 2;

free(t);
```

$$
\begin{aligned}
&\longleftrightarrow \mathbf{let}_{\mathsf{ptr(int)}}\ t = \mathsf{heapAlloc}_{\mathsf{UninitArray}_{\mathsf{int}}(9)}();\\
&\longleftrightarrow \mathsf{set}(t \boxplus 1, \mathsf{get}(t \boxplus 1) + 2);\\
&\longleftrightarrow f(\mathsf{get}(t \boxplus 1));\\
&\longleftrightarrow \mathsf{free}(t);\\
&\longleftrightarrow \mathbf{let}_{\mathsf{ptr(ptr(int))}}\ u =\\
&\qquad \mathsf{ref}_{\mathsf{ptr(int)}}(\mathsf{heapAlloc}_{\mathsf{UninitArray}_{\mathsf{int}}(9)}());\\
&\longleftrightarrow \mathsf{free}(\mathsf{get}(u));\\
&\longleftrightarrow \mathsf{set}(u, \mathsf{heapAlloc}_{\mathsf{UninitArray}_{\mathsf{int}}(8)}());\\
&\longleftrightarrow \mathsf{set}(\mathsf{get}(u) \boxplus 1, \mathsf{get}(\mathsf{get}(u) \boxplus 1) + 2);\\
&\longleftrightarrow \mathbf{let}_{\mathsf{ptr(int)}}\ v = \mathsf{stackAlloc}_{\mathsf{UninitArray}_{\mathsf{int}}(9)}();\\
&\longleftrightarrow \mathsf{set}(t \boxplus 1, \mathsf{get}(v \boxplus 1) + 2);\\
&\longleftrightarrow \mathbf{let}_{\mathsf{ptr(int)}}\ a = v \boxplus 1;\\
&\longleftrightarrow f(\mathsf{get}(a));\\
&\longleftrightarrow \mathbf{let}_{\mathsf{ptr(int)}}\ m = \mathsf{heapAlloc}_{\mathsf{UninitMatrix1}_{\mathsf{int}}(9)}();\\
&\longleftrightarrow \mathsf{set}(m \boxplus \mathsf{mIndex1}(9,1),\\
&\qquad \mathsf{get}(m \boxplus \mathsf{mIndex1}(9,1)) + 2);\\
&\longleftrightarrow \mathsf{free}(t);
\end{aligned}
$$

Figure 3.2: Additional example translations involving arrays. If a corresponds to the address of an array, then the memory address of i-th cell of the array a can be computed by the operation $a \boxplus i$. We suppose that a function `void f(int)` is defined. We also suppose that variables marked as `const` are never modified and that their address is never taken.

the reverse translation.

The lines of figure 3.1 involving `y` illustrate the encoding of operators. The lines involving `p` show how we handle the address-of operator. Finally, the lines involving `q` show how our translation handles a mutable variable that stores pointers.

Translation in presence of arrays According to the C standard, after a local declaration `int t[2]`, the variable `t` is effectively an immutable pointer to the first cell of the array. This fact guides our translation, and makes us treat `t` as a pure pointer. Then, in Optiλ all array cells are accessed through pointer arithmetic. Figure 3.2 shows example of the bidirectional translation in presence of arrays. In this figure, the lines involving `t` show how we translate heap allocated arrays. The lines involving `u` show manipulations on a mutable pointer to an array. The lines involving `v` show that only the allocation function changes between stack allocated and heap allocated array. Lines with `a` show how we handle address of array cells. Finally, lines with `m` show that matrices are simply encoded as arrays with a slightly different allocation and a primitive function (here `mIndex1`) to compute index offset. As we saw in case studies, in OptiTrust, we do not support native C multidimensional arrays and use `MINDEX` accesses instead as it is a common practice to flatten arrays in manually micro-optimized code.

Translation in presence of structures Figure 3.3 shows examples of translation with structures and field accesses. In this figure, the first lines with `a` and `b` show how we encode manipulation of constant struct values. Lines with c show our handling of mutable struct values. Notice that we prefer to first compute a precise location for the field before reading the data as this prevents reading fields that we are not interested in. Lines with u

```
typedef struct { int x; int y; } point;      ⟷ type point = {x : int, y : int};
const point a = { 0, 1 };                    ⟷ let_point a = {x = 0, y = 1};
f(a.x)                                       ⟷ f(a.x);
const point b = a;                           ⟷ let_point b = a;

point c = { 0, 1 };                          ⟷ let_ptr(point) c = ref_point({x = 0, y = 1});
f(c.x)                                       ⟷ f(get_int(c ⊡ x));
c.x = 2;                                     ⟷ set_int(c ⊡ x, 2);
c = a;                                       ⟷ set_point(c, a)
point* const q = &c;                         ⟷ let_ptr(point) q = c
int* const p = &c.x;                         ⟷ let_ptr(int) p = c ⊡ x

point* const u = malloc(9 * sizeof(point)); ⟷ let_ptr(point) u = heapAlloc_UninitArray_point(9)();
u[1] = c;                                    ⟷ set_point(u ⊞ 1, get_point(c));
u[1].x = 2;                                  ⟷ set_int((u ⊞ 1) ⊡ x, 2);
```

Figure 3.3: Additional example translations involving structures, as well as arrays of structures. The operator $p \boxdot x$, where p is the address of a structure, computes the address of the field x of this structure. In this figure, we made explicit the types for get and set operations, in particular to show that one operation can read or write entire structures.

show how we handle arrays of structures. However, in the version presented in this manuscript, we do not yet support structures containing arrays. This is naturally an expected extension for future work.

Style annotations We next explain the last key ingredient of our bidirectional translation: the use of *style annotations*. The issue stems from the fact that Optiλ features fewer language constructions than OptiC. For example, OptiC features the construct **if** without **else**, as well as C's ternary operator `b ? x : y`, whereas Optiλ only features a plain **if then else** construct. To enable going back from Optiλ to OptiC, we allow certain Optiλ terms to carry *style annotations* written $\langle t \rangle^A$ where t is an Optiλ term and A is the set of *style annotations*. For example, we can annotate an **if then else** construct with the annotation ~~**else**~~ to indicate that the **else** branch was absent in the input OptiC code. The example below shows additional examples of style annotations, with if-statements annotated using && or ?:, for keeping track of specific OptiC constructions.

```
void f(int* p) {
  if (p && *p == 0) {
    *p = 1;
  }
  if (p ? (*p == 2) : false) {
    *p = 3;
  } else {}
}
```

```
let f = fun(p : ptr(int)) ↦ {
  ⟨if (⟨if p then eq(get(p), 0) else false⟩^&&) then {
     set(p, 1)
  } else {}⟩^~~else~~
  if (⟨if p then eq(get(p), 2) else false⟩^?:) then {
     set(p, 3)
  } else {}
};
```

The key point is that a style annotation never alters the semantics of a construct. OptiTrust transformations do their best at preserving existing annotations. Yet, semantics preservation remains guaranteed even if any of the style annotation is dropped during a transformation.

R	$\ni$	|	$\textbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$	range for range-based **for** loops
π	$\ni$	|	**seq** | **par**	execution mode for range-based **for** loops
r	$\ni$	|	$\emptyset$ | x	result of a sequence
t	$\ni$	|	x	variables
		|	b | n	boolean values, and number values
		|	$\{t_1; ...; t_n; r\}$	sequence with result r
		|	**let** $x = t$	variable definition
		|	$\textbf{fun}(a_1, ..., a_n) \mapsto t$	function definition
		|	$t_0(t_1, ..., t_n)$	function call
		|	$\textbf{for}^{\pi}\ (i \in R)\ t$	possibly parallel, range-based **for** loop
		|	**if** t_0 **then** t_1 **else** t_2	conditional
		|	**type** $x = \{f_1 : \hat{\tau}_1, ..., f_n : \hat{\tau}_n\}$	definition of a structure type
		|	$\{f_1 = t_1, ..., f_n = t_n\}$	structure as values
		|	$t.f$	projection from struct values
		|	$t_1 \boxplus t_2$ | $t \boxdot f$	address computation

Figure 3.4: Grammar of Optiλ, the internal λ-calculus of OptiTrust. τ is a restriction of t to terms that can represent types, and $\hat{\tau}$ is a restriction of τ for types that correspond to an OptiC type. Some variables names x are predefined for built-in types (e.g. int) or functions (e.g. heapAlloc). Additionnally, we use infix notation for arithmetic operations internally represented as regular function calls (e.g. $x_1 + x_2 = \text{add}(x_1, x_2)$). The actual abstract syntax tree moreover features placeholders for carrying several kinds of annotations detailed later in this section.

3.2 Optiλ: OptiTrust's internal, imperative λ-calculus

This section describes Optiλ, the internal language of OptiTrust manipulated by transformations, which is based on an imperative λ-calculus.

Figure 3.4 gives the grammar of Optiλ. In this language, variables are bound by **let** bindings and by function arguments, and they are always immutable. Immutable variables allow for a straightforward implementation of substitution: variables may be substituted with values without concern on whether occurrences appear as right- or left-values. We next describe the grammar, starting with the less common features. The formal call-by-value semantics of Optiλ, may be found in appendix A.

***n*-ary functions** Standard λ-calculus usually only defines functions with exactly one argument and one return value. We say that such functions are *unary*. To express functions with more than one argument, λ-calculus usually relies on an encoding called *currying*. With such encoding, a function taking two arguments is written as a function returning a function (if the function takes two integers and returns an integer, this corresponds to the type written $\text{int} \to (\text{int} \to \text{int})$). To encode functions with zero arguments or not returning any value, standard λ-calculus define a unit type. This unit type has only one possible value, therefore it does not convey any information, and does not need any space in memory. Then, a function taking zero argument and returning nothing can be encoded as a function from unit to unit.

Although unary functions and currying are standard, Optiλ does not use those. Instead, the Optiλ construction for function definition allows an arbitrary number of arguments, and at most one return value. We say that such functions are *n-ary*, and this pattern usually occurs in imperative languages such as C.

This choice stems from the fact that we want Optiλ to be closer to the machine than standard λ-calculus. Semantically, unary functions are simpler, but the currying pattern creates a lot of intermediate function closures[1]. In OptiTrust, we want to reason about performance, and therefore we need the distinction between a function that returns a closure and a function that simply takes more than one argument, thus the support for n-ary functions.

1: Generally, compilers for languages with unary functions automatically eliminate the allocation of intermediate closures to avoid performance slowdowns

With those n-ary functions, Optiλ does not need a proper unit type, and instead handles expressions that do not return a value inside sequences[2].

2: That said, we will likely introduce a unit type in the future since such type can be very useful in presence of polymorphism.

Sequences A sequence is a term that consists of a list of subterms with side effects or **let** bindings, to be executed in order. Additionally, after the sequential execution of the subterms, a sequence may return a value. A sequence is written $\{t_1; ...; t_n; r\}$, where r denotes the optional return value for the sequence[3]. This return value may be absent, in that case we write it $\emptyset$. We enforce that the expression r does not perform side effects. In our current implementation, the result value r is syntactically restricted to be either $\emptyset$ or a variable. We translate a statement of the form **return** t that appears in terminal position of a C function into "**let** $x = t$; x" where x is a fresh variable name.

3: This presentation of sequences is similar to that found in, e.g., the Rust language.

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

Moreover, in Optiλ, we enforce that all the instructions in a sequence do not have a return value. To do so, we insert calls to the built-in function "ignore" around instructions that are not of type `void` in the OptiC code. Eliminating the implicitly ignored returned values coming from the user-facing language helps to simplify typechecking and transformations.

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

Manipulation of heap and stack cells To account for heap-allocated data, OptiTrust provides the following standard primitive functions: $\mathsf{heapAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}$ for allocating an uninitialized cell of type $\hat{\tau}$ on the heap, get for reading a cell, set for writing a cell, and free for freeing allocated cells. As usual, a read in an uninitialized memory cell is undefined behavior. More generally, heapAlloc can be used for matrix allocation. For example $\mathsf{heapAlloc}_{\mathsf{UninitMatrix2}_{\mathsf{int}}(5,8)}$ allocates an uninitialized matrix of 5×8 integers. Additionally, to account for stack-allocated variables, OptiTrust includes special functions. The operation $\mathsf{stackAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}()$ allocates a memory cell of type $\hat{\tau}$ on the stack without initializing its contents. The corresponding space is automatically reclaimed at the end of the surrounding sequence. Like for heapAlloc, stackAlloc can also be used to allocate matrices on the stack. The operation $\mathsf{ref}(t)$ also allocates a memory

cell on the stack but initializes it with t. These two special operations are meant to occur as part of a **let** binding, for example **let** $x = \text{ref}(3)$, occurring directly within a sequence. Note that a binding **let** $x = \text{ref}(t)$ is semantically equivalent to **let** $x = \text{stackAlloc}_{\text{Cell}_{\hat{\tau}}}(); \text{set}(x, t)$ where $\hat{\tau}$ is the type of t. The two stack-allocation operators, apart from their implicit-free behavior, are treated like other primitive functions.

Unbounded integers and infinitely precise reals In OptiTrust, the type int can accept infinitely large integers, like in Python. This greatly helps when proving properties about the code, and removes corner cases for arithmetic optimizations that should normally deal with possible overflows. In practice, such unbounded integers can have a significant performance cost, and therefore they should be eliminated at some point during the interactive compilation process. For now, we expect the user to choose a large enough data type after all the transformations manipulating integer arithmetic. The transformation that establishes this choice should insert assertions that prevent overflow or prove the absence of such overflows. However, this feature is left for future work, and we currently trust the choice of the user. In such future work, we plan to leverage function specifications and a value analysis to choose an actual bounded size for representing integers. To keep things simple, the formalization does not include fixed size integer types in the grammar.

Similarly, most arithmetic transformations only work with idealized real numbers instead of floating point computations performing rounding at every step. To handle that we use the type real that does never lose precision. Such infinite precision types require symbolic reasoning which does not correspond to actual CPU instructions. Therefore, like for integers, we currently trust the user for choosing a precise enough floating point approximation after all the transformations manipulating those idealized reals.

Possibly parallel range-based for loops The construct $\textbf{for}^{\pi}(i \in \textbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}}))\ t_{\text{body}}$ describes a *range-based* ***for*** *loop*. In such a loop, the immutable variable i denotes the loop index. The loop range consists of the loop bounds and the per-iteration step, that are evaluated only once before starting the loop. Following the convention used by Python and other languages, the index goes from the *start* value inclusive to the *stop* value exclusive. If the *step* value is negative, the loop index iterates downwards. The loop is tagged with an *execution mode* π that can be either **seq** of **par**. If π is set to **seq**, the loop is executed sequentially (i.e. one iteration at a time). If π to **par**, then the loop is treated as a parallel loop. The flag **par** corresponds to the directive: `#pragma openmp parallel`. The restrictions imposed by OpenMP on the ranges of parallel **for** loops essentially constrain them to fit the format $\textbf{range}(t_{\text{start}}, t_{\text{stop}}, t_{\text{step}})$, which is the format that we use for our range-based **for** loops. In this manuscript, we omit the execution mode of a loop when it is irrelevant.

Structured data Like all variables, mutable record and arrays are allocated by means of a call to the stackAlloc or heapAlloc functions. The construct $\{f_1 = t_1; \ldots; f_n = t_n\}$ can be used to build records as constant values, and in the current version, constant array values are not supported. OptiTrust features three operations to manipulate structured data.

If a corresponds to the address of an array, then the memory address of i-th cell of the array a can be computed by the operation $a \boxplus i$. This operation corresponds to the C pointer arithmetic operation `a+i`, that is more intuitively

written `&a[i]`. The contents of that cell may be retrieved by evaluating $\mathrm{get}(a \boxplus i)$.

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 corresponds to shifting the pointer r by the offset associated with the field f.

All these projection and address-shifting operations are here presented as constructs of the grammar. From the perspective of typechecking, however, we treat these operations like function applications for better factorization.

Other language constructs The other language constructs of Optiλ are standard. They include function calls and conditionals. Our implementation accounts for a diversity of literal types. For simplicity, we consider in the formalization only two kinds of literals: the metavariable b denotes a boolean literal (either **true** or **false**), and the metavariable n denotes an integer literal.

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

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

- the location of the subterm in the initial source code;
- user-placed marks allow referring to subterms by name in transformation script's targets;
- separation logic contracts for functions and loops;
- type information for all bindings, operators, and for every subterm;
- style annotations to guide the reverse translation from Optiλ to OptiC, as described in more details in the next subsection.

Implementation of the AST The Optiλ abstract syntax tree (AST) is represented as an immutable tree data structure. A program transformation takes as input such an immutable AST, and produces as output another AST, which may share subtrees with the input AST. There are two major benefits following a purely functional programming style using immutable trees. First, this approach avoids numerous bugs typically associated with inadvertent sharing of subtrees when modifying data structures in-place. Second, this approach, by enabling sharing, can lead to a more compact construction of complete execution traces, which are used for reporting to the user all the intermediate ASTs constructed during the evaluation of the user's transformation script.

Another implementation choice of this AST is the encoding of variables. In OptiTrust, we associate each variable binder with a unique ID. Then, we have

a procedure to propagate such IDs to variable occurrences, following scope and shadowing rules. These variable ID allows fast variable comparison, and interestingly prevent unintended scope or shadowing issues when moving around instructions in transformations. The only caveat is that transformations need to refresh binder IDs (and update the corresponding variable occurrences) when duplicating terms.

3.3 OptiC: a C-like, user-facing language

We strive to make the user-facing language of OptiTrust as close to C as possible, in order to make the tool accessible to most high performance programmers. This section describes the OptiC language by comparison with C, and without giving details about resource annotations that will be discussed in chapter 4.

Comparison with C Syntactically, OptiC is a subset of C with custom resource annotations and with a few extensions borrowed from C++. Our current implementation of OptiTrust parses OptiC code using Clang. Moreover, OptiTrust users can benefit from the C or C++ support of their IDEs to edit OptiC code. The exact grammar of the supported subset of C is not explicitly given since the next section details all the supported constructions along with their translation in Optiλ.

Semantically, OptiC admits a simpler semantics than C. Supporting all the features of the C language would be extremely challenging. To see why, it suffices to contemplate the size of the Rocq formalization of a significant subset of C [Kre15]. OptiC features fewer undefined behaviors than C, in particular with respect to evaluation order. Hence, it is incorrect to compile OptiC code using an arbitrary C-compliant compiler. Instead, either a prior translation to C is required, e.g. to bind intermediate expressions; or one should translate OptiC code directly into a lower-level language, such as CompCert's Clight or LLVM IR. Currently, the cleanup transformation we saw in chapter 2 tries to convert an OptiC input into C compatible code. As of today, this procedure is probably incomplete and is not verified. We leave for future work a more trustworthy backend for OptiC.

[Kre15]: Krebbers (2015), *The C standard formalized in Coq*

Strict order of evaluation for all operators In standard C, operators do not necessarily behave like calls to primitive functions. Indeed, the standard allows for a more liberal argument evaluation order. This is visible for pre-/post-increment/decrement operators such as `i++`. For example, it is undefined whether the instruction `u = u++;` increments `u` or not. However, when written as call-by-value function calls, `set(&u, getAndIncr(&u))`, it is obvious that it cannot increment `u`. Since code transformations might accidentally produce code such as `u = u++`, and we do not want to treat operators in a special way, the OptiC language exposes fewer undefined behaviors than standard C. In that case, we impose that operators behave like function calls. This means that we do not guarantee any evaluation order of the arguments, but we ensure that all arguments performed all their side effects before the execution of an operator syntactically higher in the AST. This is not a problem when importing C code because this only restricts the number of possible behaviors of any given piece of code.

Types for unbounded integers and reals In OptiC, we consider that type `int` corresponds to the unbounded Optiλ type. Considering that unbounded integers are the default specifically alleviates the burden of undefined behaviors when signed overflow occurs. We also support fixed size types such as `int32_t` or `uint64_t`, but we do not yet support arithmetic simplification on these types. However, we have no plans for supporting platform specific data types such as `short` or `long`. For real arithmetic, we add a specific data type `real` to OptiC, and we also support the classic floating point approximations `float` and `double` (again without arithmetic simplification). Neither `int` not `real` yields executable code, but OptiTrust users can transform them into fixed size integers or into regular floating point variables, as part of a user trusted step.

Variables marked `const` and function arguments are pure As we saw in section 3.1, it is important for our translation to understand whether a variable is pure or impure. Since transformations may react differently in presence of pure or impure variables, we want to syntactically distinguish the variables that we treat as pure. To reduce the amount of syntactic noise, our internal C-like language treats the keyword `const` as a request to handle the variable as a pure variable. Moreover, we made the design decision to always treat function arguments as pure variables. The mutation of function arguments is allowed in C, yet it is a rarely used feature, which can easily be avoided. We might, in future work, extend our translation to handle mutated arguments by introducing an auxiliary fresh local mutable variable, and turning the mutated argument into a constant argument.

Function types and function variables In C, a programmer can write pointers to functions such as `int (*fptr)(int);`. This type can only accept functions that do not capture their surrounding environment. The semantics of the operators & and $*$ when applied to C functions is not as simple as one may hope. In OptiC, we chose instead to use the C++ function types, such as `std::function<int(int)>`, which in this paper we write `fun<int(int)>`, where `fun` is defined as an alias for `std::function`. A local variable with a function type may be either pure or impure. However, all functions declared using the syntax of C function definitions (e.g. `int f(){ return 42; }`) are represented as pure variables.

Syntax extensions for translating back from Optiλ OptiTrust transformations may generate ASTs in Optiλ with arbitrary shapes. In particular, the grammar of Optiλ allows function abstractions and sequences with a return value to appear anywhere as subterms in the AST. This flexibility does not exist in the standard C. Yet, we wish to be able to display to the OptiTrust user the corresponding AST in OptiC syntax. To that end, we consider standard extensions of the C language; such extensions would probably be already familiar to the OptiTrust user. For sequences, we consider the GNU C extensions[4], which supports the syntax `({` u_1; ...; u_n; u_r; `})`, where u_r is an expression that corresponds to the return value. For functions, we borrow the C++ syntax for closures, written `[&](T_1 a_1, ..., T_n a_n){...}`. Moreover, we allow the nested functions from the GNU C extensions[5].

4: https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Statement-Exprs.html

5: https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Nested-Functions.html

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

We currently do not support structures containing arrays, this is due to the additional complexity of field access of array type. If `f` is a non-array field, the expression `a.f` reads the value at the address `&(a.f)`. However, if `f` is declared in the form `int f[5]`, then the expression `a.f` returns the address of the first cell without reading any memory. We could add support for these array fields in the future by extending our bidirectional translation.

The current version of OptiTrust only supports range-based **for** loops on a range fixed when entering the loop the first time. Indeed, as we discuss in chapter 4, the OptiTrust typechecker ensures program termination, and therefore more general loop patterns would require a more complex treatment. In the future, we plan to add a single form of **repeat** loop in Optiλ and use it to encode `while` loops and general forms of C `for` loops. Likewise, due to complication for termination checks, recursive functions are currently not supported and should be added in the future.

To handle abrupt termination, as triggered by `break`, `continue`, and non-final `return` statements, we would need a generalization of our typesystem. The treatment of abrupt termination in separation logic is well-understood—they are handled, for example, in the VST program verification framework for C programs [Cao+18]. That said, abrupt termination support introduces a fair amount of additional complexity, explaining why we have not covered it yet. Note that our implementation handles the case of a function ending with a conditional that has both branches ending with a `return`, but this case is not covered in this manuscript for simplicity.

[Cao+18]: Cao et al. (2018), *VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs*

3.4 Translation from OptiC to Optiλ

This section describes the translation from OptiC into OptiTrust's internal λ-calculus. We call this operation the *encoding*.

As exposed in the overview, the translation from the user-facing language to the internal λ-calculus crucially depends on the notion of pure variables. We can assume at this step that all pure variables are marked with the `const` keyword in OptiC (recall that it corresponds to `const register` in C). This means that pure variables can be considered immutable and without an address.

The essence of the encoding process is to eliminate the notion of left-value by replacing impure variables with their stack-allocated address. Then, the encoding wraps the accesses to values of impure variables with a get operation. Such a process of elimination of the left-values is commonly found in the implementation of compilers. However, compilers in general are not concerned with supporting a reverse translation.

Figures 3.5 and 3.6 define our translation from OptiC to Optiλ. We write $\lfloor u \rfloor$ the encoding of an OptiC term u, which could be a statement or an expression in right-value position. We write $\lfloor u \rfloor^{\&}$ the encoding of an OptiC term u appearing in left-value position.

The encoding operation builds a global set Π that contains the identifiers of all variables marked as pure. Recall that in particular, this includes the names of arguments that appear in function definitions. Every right-value occurrence of a variable that does not belong to Π becomes wrapped inside a call to get. Note that this encoding fails if the invariants imposed by the declared purity of a variable are not satisfied. Recall that this encoding adds style annotations to the Optiλ terms being produced.

$$
\begin{array}{lcl}
\lfloor u \rfloor^{\&} & = & t \quad \text{where } \lfloor u \rfloor \text{ is (guaranteed to be) of the form } \mathrm{get}(t) \\
\lfloor x \rfloor & = & \begin{cases} x & \text{if } x \in \Pi \\ \mathrm{get}(x) & \text{otherwise} \end{cases} \\
\lfloor b \rfloor & = & b \\
\lfloor n \rfloor & = & n \\
\lfloor u_1 + u_2 \rfloor & = & \mathrm{add}(\lfloor u_1 \rfloor, \lfloor u_2 \rfloor) \\
\lfloor \&u \rfloor & = & \lfloor u \rfloor^{\&} \\
\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 \texttt{ += } u_2 \rfloor & = & \mathrm{inplaceAdd}(\lfloor u_1 \rfloor^{\&}, \lfloor u_2 \rfloor) \\
\lfloor u_0(u_1, \ldots, u_n) \rfloor & = & \lfloor u_0 \rfloor(\lfloor u_1 \rfloor, \ldots, \lfloor u_n \rfloor) \\
\lfloor T \texttt{ const } x = u \rfloor & = & \mathbf{let}_{\lfloor T \rfloor^{\mathrm{typ}}}\ x = \lfloor u \rfloor \qquad (x \in \Pi) \\
\lfloor T\ x = u \rfloor & = & \mathbf{let}_{\mathrm{ptr}(\lfloor T \rfloor^{\mathrm{typ}})}\ x = \mathrm{ref}_{\lfloor T \rfloor^{\mathrm{typ}}}(\lfloor u \rfloor) \qquad (x \notin \Pi) \\
\lfloor T\ x \rfloor & = & \mathbf{let}_{\mathrm{ptr}(\lfloor T \rfloor^{\mathrm{typ}})}\ x = \mathrm{stackAlloc}_{\mathrm{UninitCell}_{\lfloor T \rfloor^{\mathrm{typ}}}}() \qquad (x \notin \Pi) \\
\lfloor \texttt{malloc(sizeof(}T\texttt{))} \rfloor & = & \mathrm{heapAlloc}_{\mathrm{UninitCell}_{\lfloor T \rfloor^{\mathrm{typ}}}}() \\
\lfloor u_0 \texttt{ ? } u_1 \texttt{ : } u_2 \rfloor & = & \langle \mathbf{if}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \lfloor u_2 \rfloor \rangle^{?:} \\
\lfloor \texttt{if(}u_0\texttt{) } u_1 \texttt{ else } u_2 \rfloor & = & \mathbf{if}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \lfloor u_2 \rfloor \\
\lfloor \texttt{if(}u_0\texttt{) } u_1 \rfloor & = & \langle \mathbf{if}\ \lfloor u_0 \rfloor\ \mathbf{then}\ \lfloor u_1 \rfloor\ \mathbf{else}\ \{\} \rangle^{\cancel{\mathbf{else}}} \\
\lfloor u_1 \texttt{ \&\& } u_2 \rfloor & = & \langle \mathbf{if}\ \lfloor u_1 \rfloor\ \mathbf{then}\ \lfloor u_2 \rfloor\ \mathbf{else}\ \mathbf{false} \rangle^{\&\&} \\
\lfloor u_1 \texttt{ || } u_2 \rfloor & = & \langle \mathbf{if}\ \lfloor u_1 \rfloor\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \lfloor u_2 \rfloor \rangle^{||} \\
\lfloor \texttt{for(int } x = u_1\texttt{; } x < u_2\texttt{; } x \texttt{ += } u_3\texttt{) } 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 \\
\lfloor \{u_1; \ldots; u_n;\} \rfloor & = & \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; \emptyset\} \\
\lfloor \{u_1; \ldots; u_n; \texttt{return;}\} \rfloor & = & \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; \langle \emptyset \rangle^{\mathrm{return}}\} \\
\lfloor \{u_1; \ldots; u_n; \texttt{return } u_r\texttt{;}\} \rfloor & = & \begin{cases} \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; x\} & \text{if } \lfloor u_r \rfloor \text{ is a variable } x \\ \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; \langle \mathbf{let}\ x = \lfloor u_r \rfloor \rangle^{\mathrm{res}}; x\} & \text{otherwise } (x \text{ is fresh}) \end{cases} \\
\lfloor (\{u_1; \ldots; u_n; u_r;\}) \rfloor & = & \begin{cases} \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; x\} & \text{if } \lfloor u_r \rfloor \text{ is a variable } x \\ \{\lfloor u_1 \rfloor^{\mathrm{void}}; \ldots; \lfloor u_n \rfloor^{\mathrm{void}}; \langle \mathbf{let}\ x = \lfloor u_r \rfloor \rangle^{\mathrm{res}}; x\} & \text{otherwise } (x \text{ is fresh}) \end{cases} \\
\lfloor u \rfloor^{\mathrm{void}} & = & \begin{cases} \lfloor u \rfloor & \text{if } u \text{ is of type } \texttt{void} \\ \mathrm{ignore}(\lfloor u \rfloor) & \text{otherwise} \end{cases} \\
\lfloor T_0\ f(T_1\ a_1, \ldots, T_n\ a_n)\ u_f \rfloor & = & \mathbf{let}_{(\lfloor T_1 \rfloor^{\mathrm{typ}} \times \ldots \times \lfloor T_n \rfloor^{\mathrm{typ}}) \to \lfloor T_0 \rfloor^{\mathrm{typ}}}\ f = \mathbf{fun}(a_1, \ldots, a_n) \mapsto \lfloor u_f \rfloor \\
\lfloor \texttt{[\&]}(T_1\ a_1, \ldots, T_n\ a_n)\ u_f \rfloor & = & \langle \mathbf{fun}(a_1, \ldots, a_n) \mapsto \lfloor u_f \rfloor \rangle^{\square} \\
\\
\lfloor T* \rfloor^{\mathrm{typ}} & = & \mathrm{ptr}(\lfloor T \rfloor^{\mathrm{typ}}) \\
\lfloor \texttt{int} \rfloor^{\mathrm{typ}} & = & \mathrm{int} \\
\lfloor \texttt{bool} \rfloor^{\mathrm{typ}} & = & \mathrm{bool} \\
\lfloor \texttt{real} \rfloor^{\mathrm{typ}} & = & \mathrm{real} \\
\lfloor \texttt{fun<}T_0\texttt{(}T_1, \ldots, T_n\texttt{)>} \rfloor^{\mathrm{typ}} & = & (\lfloor T_1 \rfloor^{\mathrm{typ}} \times \ldots \times \lfloor T_n \rfloor^{\mathrm{typ}}) \to \lfloor T_0 \rfloor^{\mathrm{typ}}
\end{array}
$$

Figure 3.5: Translation from OptiC to Optiλ. Translation rules for structured data are given in the following figure 3.6. A global, precomputed set Π contains the identifiers of all *pure* variables—in OptiTrust, variables carry unique identifiers in addition to their names. Superscripts on λ-terms represent style annotations for translating back to OptiC.

$$
\begin{array}{lcl}
\lfloor \texttt{typedef struct } \{T_1\ f_1; \ldots; T_n\ f_n;\}\ x \rfloor & = & \textbf{type}\ x = \{f_1 : \lfloor T_1 \rfloor^{\text{typ}}, \ldots, f_n : \lfloor T_n \rfloor^{\text{typ}}\} \\
\lfloor (T)\{u_1, \ldots, u_n\} \rfloor & = & \{f_1 = \lfloor u_1 \rfloor, \ldots, f_n = \lfloor u_n \rfloor\} \\
 & & \quad \text{where } f_i \text{ are the fields of the struct type } T \\
\lfloor u.f \rfloor & = & \begin{cases} x.f & \text{if } u = x \wedge x \in \Pi \\ \lfloor u \rfloor.f & \text{if } u \text{ is a constant structure} \\ \text{get}(\lfloor u.f \rfloor^{\&}) & \text{otherwise} \end{cases} \\
\lfloor u.f \rfloor^{\&} & = & \begin{cases} \langle \lfloor u' \rfloor \boxdot f \rangle^{\twoheadrightarrow} & \text{if } u \text{ is of the form } (*u') \\ \lfloor u \rfloor^{\&} \boxdot f & \text{otherwise} \end{cases} \\
\lfloor u \rightarrow f \rfloor & = & \text{get}(\lfloor u \rightarrow f \rfloor^{\&}) \\
\lfloor u \rightarrow f \rfloor^{\&} & = & \lfloor u \rfloor \boxdot f \\
\\
\lfloor T\ x[u] \rfloor & = & \textbf{let}_{\text{ptr}(\lfloor T \rfloor^{\text{typ}})}\ x = \text{stackAlloc}_{\lfloor u, T \rfloor^{\text{arr}}}() \qquad (x \in \Pi) \\
\lfloor \texttt{malloc(}u\texttt{* sizeof(}T\texttt{))} \rfloor & = & \text{heapAlloc}_{\lfloor u, T \rfloor^{\text{arr}}}() \\
\lfloor u_1[u_2] \rfloor & = & \text{get}(\lfloor u_1[u_2] \rfloor^{\&}) \\
\lfloor u_1[u_2] \rfloor^{\&} & = & \lfloor u_1 \rfloor \boxplus \lfloor u_2 \rfloor
\end{array}
$$

$$
\lfloor u, T \rfloor^{\text{arr}} \quad = \quad \begin{cases} \text{UninitMatrix1}_{\lfloor T \rfloor^{\text{typ}}}(\lfloor u_n \rfloor) & \text{if } u \text{ is of the form } \texttt{MSIZE1(}u_n\texttt{)} \\ \text{UninitMatrix2}_{\lfloor T \rfloor^{\text{typ}}}(\lfloor u_m \rfloor, \lfloor u_n \rfloor) & \text{if } u \text{ is of the form } \texttt{MSIZE2(}u_m\texttt{, }u_n\texttt{)} \\ \text{UninitArray}_{\lfloor T \rfloor^{\text{typ}}}(\lfloor u \rfloor) & \text{otherwise} \end{cases}
$$

Figure 3.6: Translation from OptiC to Optiλ for arrays and structures. A global, precomputed set Π contains the identifiers of all *pure* variables—in OptiTrust, variables carry unique identifiers in addition to their names. Superscripts on λ-terms represent style annotations for translating back to OptiC.

[BC25b]: Bertholon et al. (2025), *Bidirectional Translation between a C-like Language and an Imperative Lambda-calculus*

In this manuscript, we consider that the semantics of an OptiC program u is defined to be the same as the semantics of the corresponding Optiλ program $\lfloor u \rfloor$. Ideas for assigning a direct semantics to OptiC and proving that the translation is semantic-preserving are discussed in the workshop article [BC25b, §4-6] which inspired most of this chapter. We believe that such a direct semantics is not needed for formal reasoning and that a more interesting future work would be to verify another semantic-preserving translation to a language with an existing verified toolchain such as CompCert's Clight.

3.5 Translation from Optiλ back to OptiC

This section defines the reciprocal translation, which we call *decoding*. Figures 3.7 and 3.8 define this decoding operation. The notation $\lceil t \rceil$ denotes the decoding of an Optiλ term t. The notation $\lceil t \rceil^*$ denotes an auxiliary operation for decoding terms that appear in left-value contexts in the OptiC output.

As mentioned earlier, during the encoding, a number of style annotations are attached to the terms produced, in order to guide the decoding phase and ensure the round-trip property. Importantly, these annotations are always ignored by the semantics. It is therefore always safe to drop annotations in the OptiTrust AST. Ignoring style annotations may even be necessary. Consider for example a transformation that rewrites "t_0; $\langle$**if** t_c **then** $\{t_1\}$ **else** $\{\}\rangle^{\text{~~else~~}}$" into "$\langle$**if** t_c **then** $\{t_0; t_1\}$ **else** $\{t_0\}\rangle^{\text{~~else~~}}$", where the annotation ~~**else**~~ in the input term indicates that the *else* branch was absent from the OptiC code. There, the resulting term is a nonempty *else* branch, hence the annotation ~~**else**~~ must be discarded.

There is one limitation with the current style annotation system expressed by the notion of *spurious pattern*, which consists of an occurrence of an

$$
\begin{array}{lcl}
\lceil t \rceil^{*} & = & \begin{cases} u & \text{if } \lceil t \rceil \text{ is of the form } \&u \\ *\lceil t \rceil & \text{otherwise} \end{cases} \\
\lceil x \rceil & = & \begin{cases} x & \text{if } x \in \Pi \\ \&x & \text{otherwise} \end{cases} \\
\lceil b \rceil & = & b \\
\lceil n \rceil & = & n \\
\lceil \mathsf{add}(t_1, t_2) \rceil & = & \lceil t_1 \rceil + \lceil t_2 \rceil \\
\lceil \mathsf{get}(t) \rceil & = & \lceil t \rceil^{*} \\
\lceil \mathsf{set}(t_1, t_2) \rceil & = & \lceil t_1 \rceil^{*} = \lceil t_2 \rceil \\
\lceil \mathsf{inplaceAdd}(t_1, t_2) \rceil & = & \lceil t_1 \rceil^{*} \mathrel{+}= \lceil t_2 \rceil \\
\lceil t_0(t_1, \ldots, t_n) \rceil & = & \lceil t_0 \rceil(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) \\
\lceil \mathbf{let}_{\hat{\tau}}\ x\ =\ t \rceil & = & \lceil \hat{\tau} \rceil^{\mathsf{typ}}\ \texttt{const}\ x = \lceil t \rceil \qquad (x \in \Pi) \\
\lceil \mathbf{let}_{\mathsf{ptr}(\hat{\tau})}\ x = \mathsf{ref}(t) \rceil & = & \lceil \hat{\tau} \rceil^{\mathsf{typ}}\ x = \lceil t \rceil \qquad (x \notin \Pi) \\
\lceil \mathbf{let}_{\mathsf{ptr}(\hat{\tau})}\ x = \mathsf{stackAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}() \rceil & = & \lceil \hat{\tau} \rceil^{\mathsf{typ}}\ x \qquad (x \notin \Pi) \\
\lceil \mathsf{heapAlloc}_{\mathsf{UninitCell}_{\hat{\tau}}}() \rceil & = & \texttt{malloc(sizeof(}\lceil \hat{\tau} \rceil^{\mathsf{typ}}\texttt{))} \\
\lceil \langle \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ \mathbf{false} \rangle^{\&\&} \rceil & = & \lceil t_1 \rceil\ \&\&\ \lceil t_2 \rceil \\
\lceil \langle \mathbf{if}\ t_1\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ t_2 \rangle^{||} \rceil & = & \lceil t_1 \rceil\ ||\ \lceil t_2 \rceil \\
\lceil \langle \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ \{\} \rangle^{\cancel{\mathbf{else}}} \rceil & = & \texttt{if(}\lceil t_0 \rceil\texttt{)}\ \lceil t_1 \rceil \qquad \text{inside sequence} \\
\lceil \langle \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rangle^{?:} \rceil & = & \lceil t_0 \rceil\ ?\ \lceil t_1 \rceil\ :\ \lceil t_2 \rceil \\
\lceil \mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \rceil & = & \begin{cases} \texttt{if(}\lceil t_0 \rceil\texttt{)}\ \lceil t_1 \rceil\ \texttt{else}\ \lceil t_2 \rceil & \text{inside sequence} \\ \lceil t_0 \rceil\ ?\ \lceil t_1 \rceil\ :\ \lceil t_2 \rceil & \text{otherwise} \end{cases} \\
\lceil \mathbf{for}\ (x \in \mathbf{range}(t_1, t_2, t_3))\ t_4 \rceil & = & \texttt{for(int}\ x = \lceil t_1 \rceil; x < \lceil t_2 \rceil; x \mathrel{+}= \lceil t_3 \rceil\texttt{)}\ \lceil t_4 \rceil \\
\lceil \{t_1; \ldots; t_n; \langle \mathbf{let}\ x = t_r \rangle^{\mathsf{res}}; x\} \rceil & = & \begin{cases} \{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil; \texttt{return}\ \lceil t_r \rceil;\ \} & \text{as function body} \\ (\{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil; \lceil t_r \rceil;\ \}) & \text{otherwise} \end{cases} \\
\lceil \{t_1; \ldots; t_n; x\} \rceil & = & \begin{cases} \{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil; \texttt{return}\ x;\ \} & \text{as function body} \\ (\{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil; x;\ \}) & \text{otherwise} \end{cases} \\
\lceil \{t_1; \ldots; t_n; \langle \emptyset \rangle^{\mathsf{return}}\} \rceil & = & \{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil; \texttt{return};\ \} \qquad \text{as function body} \\
\lceil \{t_1; \ldots; t_n; \emptyset\} \rceil & = & \{\lceil t_1 \rceil; \ldots; \lceil t_n \rceil;\ \} \qquad \text{inside sequence} \\
\lceil \mathsf{ignore}(t) \rceil & = & \lceil t \rceil \\
\lceil \mathbf{let}_{(\hat{\tau}_1 \times \ldots \times \hat{\tau}_n) \to \hat{\tau}_0}\ f = \langle \mathbf{fun}(a_1, \ldots, a_n) \mapsto t_f \rangle^{A} \rceil & = & \lceil \hat{\tau}_0 \rceil^{\mathsf{typ}}\ f(\lceil \hat{\tau}_1 \rceil^{\mathsf{typ}}\ a_1, \ldots, \lceil \hat{\tau}_n \rceil^{\mathsf{typ}}\ a_n)\ \lceil t_f \rceil \qquad \text{if } \square \notin A \\
\lceil \mathbf{fun}(a_{1\,\hat{\tau}_1}, \ldots, a_{n\,\hat{\tau}_n}) \mapsto t_f \rceil & = & \texttt{[\&]}(\lceil \hat{\tau}_1 \rceil^{\mathsf{typ}}\ a_1, \ldots, \lceil \hat{\tau}_n \rceil^{\mathsf{typ}}\ a_n)\ \lceil t_f \rceil \\
\\
\lceil \mathsf{ptr}(\hat{\tau}) \rceil^{\mathsf{typ}} & = & \lceil \hat{\tau} \rceil^{\mathsf{typ}}* \\
\lceil \mathsf{int} \rceil^{\mathsf{typ}} & = & \texttt{int} \\
\lceil \mathsf{bool} \rceil^{\mathsf{typ}} & = & \texttt{bool} \\
\lceil \mathsf{real} \rceil^{\mathsf{typ}} & = & \texttt{real} \\
\lceil (\hat{\tau}_1 \times \ldots \times \hat{\tau}_n) \to \hat{\tau}_0 \rceil^{\mathsf{typ}} & = & \texttt{fun<}\lceil \hat{\tau}_0 \rceil^{\mathsf{typ}}\texttt{(}\lceil \hat{\tau}_1 \rceil^{\mathsf{typ}}, \ldots, \lceil \hat{\tau}_n \rceil^{\mathsf{typ}}\texttt{)>}
\end{array}
$$

Figure 3.7: Translation from OptiTrust's internal λ-calculus back to C. Translation rules for structured data are given in the following figure 3.8. A global, precomputed set Π contains the identifiers of all *pure* variables—in OptiTrust, variables carry unique identifiers in addition to their names. Style annotations are silently ignored if no rule matches.

$$
\begin{array}{lcl}
\lceil \textbf{type}\ x = \{f_1 : \hat{\tau}_1, ..., f_n : \hat{\tau}_n\} \rceil & = & \texttt{typedef struct } \{\lceil \hat{\tau}_1 \rceil^{\text{typ}}\ f_1; ...; \lceil \hat{\tau}_n \rceil^{\text{typ}}\ f_n;\}\ x \\
\lceil \{f_1 = t_1, ..., f_n = t_n\} \rceil & = & (T)\{\lceil t_1 \rceil, ..., \lceil t_n \rceil\} \qquad \text{where } T \text{ is the struct type with fields } f_i \\
\lceil t.f \rceil & = & \lceil t \rceil.f \\
\lceil \langle \text{get}(t) \boxdot f \rangle^{\rightarrow *} \rceil & = & \&((*\lceil t \rceil^*).f) \\
\lceil t \boxdot f \rceil & = & \begin{cases} \&(u \rightarrow f) & \text{if } \lceil t \rceil^* \text{ is of the form } *u \\ \&(\lceil t \rceil^*.f) & \text{otherwise} \end{cases} \\
\lceil \textbf{let}_{\text{ptr}(\hat{\tau})}\ x = \text{stackAlloc}_{C_{\hat{\tau}}}() \rceil & = & \lceil \hat{\tau} \rceil^{\text{typ}}\ x[\lceil C \rceil^{\text{size}}] \qquad (x \in \Pi) \\
\lceil \text{heapAlloc}_{C_{\hat{\tau}}}() \rceil & = & \texttt{malloc(}\lceil C \rceil^{\text{size}} * \texttt{ sizeof(}\lceil \hat{\tau} \rceil^{\text{typ}}\texttt{))} \\
\lceil t_1 \boxplus t_2 \rceil & = & \&(\lceil t_1 \rceil[\lceil t_2 \rceil]) \\
\\
\lceil \text{UninitArray}(t) \rceil^{\text{size}} & = & \lceil t \rceil \\
\lceil \text{UninitMatrix1}(t_n) \rceil^{\text{size}} & = & \texttt{MSIZE1(}\lceil t_n \rceil\texttt{)} \\
\lceil \text{UninitMatrix2}(t_m, t_n) \rceil^{\text{size}} & = & \texttt{MSIZE2(}\lceil t_m \rceil, \lceil t_n \rceil\texttt{)}
\end{array}
$$

Figure 3.8: Translation from Optiλ to OptiC for arrays and structures. A global, precomputed set Π contains the identifiers of all *pure* variables. Style annotations are silently ignored if no rule matches.

expression of the form `&*u` or `*&u`. Such spurious patterns are eliminated during the process of encoding a program and do not generate any annotation. Even though we could design a style annotation that preserves spurious patterns, we believe that it is not worth the extra work, because spurious patterns usually do not appear in human-written source programs.

We are now ready to state the round-trip property of our bidirectional translation.

Proposition 3.5.1: Round-trip for OptiC programs

If u is an OptiC program that does not contain spurious patterns, and such that its encoding $\lfloor u \rfloor$ is well-defined, then decoding gives back the original program: $\lceil \lfloor u \rfloor \rceil = u$.

An actual mechanized proof of this round-trip property is left for future work and should be doable by induction on the possible constructions for OptiC programs.

Note that this roundtrip theorem is defined at the level of the ASTs. In our current implementation, comments, blank lines, indentation, and macro expansions, are not part of the AST. Hence, they are not preserved by our translations. Indentation is not much an issue for users using code formatters. Comments and blank lines could be attached to AST nodes, as style annotation, if need be. Macros would be trickier to handle. One could imagine a best-effort algorithm, which folds back macros for parts of the AST that have not been altered by the transformations.

A very attentive reader might notice that the reverse round-trip (i.e. $\lfloor \lceil t \rceil \rfloor = t$) does not hold. For a pure variable x of type $\text{ptr}(\hat{\tau})$ where $\hat{\tau}$ is a structure type with a field f, the Optiλ terms $\text{get}(x \boxdot f)$ and $\text{get}(x).f$ are both decoded as $x \rightarrow f$. This is due to a fundamental ambiguity in C-like languages in presence of structure accesses: there is no distinction between first reading the full structure and extracting one field and directly reading only one field. In the encoding, we always prefer the latter (i.e. $\lfloor x \rightarrow f \rfloor = \text{get}(x \boxdot f)$) since it corresponds to the most efficient code by limiting memory accesses. In practice, we currently always normalize Optiλ terms of the form $\text{get}(x).f$ into $\text{get}(x \boxdot f)$, with an ad-hoc transformation, therefore this ambiguity of OptiC is not yet an issue. In the future, if we want to keep a term of the form

get$(x).f$ between two user interactions, we might change the semantics of OptiC by giving different meanings to $x{\rightarrow}f$ and $(*x).f$ and to $x.f$ and $(*\&x).f$.

Computing program resources: Contexts

4

As we claimed in the introduction, resource typing is a key ingredient in OptiTrust to obtain invariants expressed as separation logic resources between each line of code. Those invariants are then leveraged to justify the correctness of code transformations. This chapter describes the details of our separation logic resource typesystem.

Traditional typecheckers have a typing judgment of the form $\Gamma \vdash t : \tau$. Yet, the OptiTrust typechecker needs to account also for linear resources. Following the presentation of separation logic, OptiTrust's typing judgment is written as a *triple* of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$. The input context Γ decomposes as $\langle E \mid F \rangle$, where E consists of *pure resources* and F consists of *linear resources*. Symmetrically, the output context Γ' contains both pure and linear resources. The pure resources from Γ' typically correspond to ghost return values and to pure postconditions. We qualify as *ghost*, any entity that is useful during program typechecking but is erased in the final executable code. Triples will be later extended in chapter 5 to the form $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$, where Δ denotes a *usage map*, providing a summary explaining which resources are used by every subterm, and how they are used. This chapter presents the typing entities and the algorithmic typing rules, ignoring usage maps.

We designed the OptiTrust typechecker such that it can be used for checking shape-only or functional correctness properties, with only very minor variations. The key difference is that, with functional correctness assertions, the typechecker needs to keep track of a *model* for each accessible memory cell. Unless stated otherwise, everything that appears in this chapter applies both for shape-only and for functional correctness assertions. The typechecker makes no difference between full functional correctness and incomplete functional correctness. Indeed, in both cases the typechecker validates all the assertions written in the code.

The chapter is organized as follows. Section 4.1 presents the grammar of *pure resources* and *linear resources*. Section 4.2 presents the grammar of *contexts*. Section 4.3 presents the grammar of *function contracts* and *loop contracts*. Section 4.4 presents the *entailment relation*. Section 4.5 presents the *subtraction procedure*, which corresponds to an algorithmic implementation of the entailment relation. Section 4.6 presents the typing judgment for *logical expressions*. Section 4.7 presents our algorithmic typing rules, which define the judgment $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$. Finally, section 4.8 presents soundness results.

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

4.1 Grammar of resources

As mentioned earlier, a context Γ decomposes as $\langle E \mid F \rangle$, where E contains pure resources and F contains linear resources. A pure resource describes a fact that remains true until the end of the program, or describes a program variable permanently bound to a given value. Pure resources may be freely duplicated during typechecking. Linear resources describe the ownership

of a given subset of the memory. Each linear resource describes a fragment of memory. Two *full* linear resources that appear in a same context must describe disjoint parts of the memory. A given full linear resource may be split into *fractional* resources, in which case several fractional linear resources may cover the same parts of memory. Subsequently, resources that were split may be joined back together. In any case, a linear resource cannot be duplicated and cannot be silently dropped. We next describe the grammar of pure and of linear resources.

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

- "τ : Type" quantifies a type variable, useful for expressing polymorphism in Optiλ.
- "$x : \tau$" quantifies a variable of type τ; and "$x : \hat{\tau}$" quantifies a variable with the OptiC type $\hat{\tau}$.
- "$f : \tau_1 \xrightarrow{\text{logic}} \tau_2$" quantifies a logical function that takes an argument of type τ_1 and returns an argument of type τ_2. Logical functions corresponds to functions that are pure and terminating. "$f : \forall(x_1 : \tau_1)(x_2 : \tau_2),\ \tau_3$" quantifies a type dependent logical function with two arguments. In that example, x_1 can appear in τ_2 and τ_3, and x_2 can appear in τ_3. In practice the non-dependent logical function syntax "$(\tau_1 \times ... \times \tau_n) \xrightarrow{\text{logic}} \tau_0$" is syntactic sugar for "$\forall(x_1 : \tau_1)...(x_n : \tau_n),\ \tau_0$" with all x_i fresh from all τ_j.
- "P : Prop" quantifies an abstract proposition; and "$Q : \tau \xrightarrow{\text{logic}}$ Prop" quantifies an abstract logical predicate over values of type τ.
- "$p : P$" quantifies a proof witness of a proposition P; for example "$p : i > 0$" captures the assumption that i is positive.
- "p : Spec(f, [a_1, ..., a_n], γ)" describes a *function specification*[1] asserting that the function f expects arguments named a_i and admits the *function contract* γ.
- "H : HProp" quantifies an abstract heap predicate[2], and "I : int $\xrightarrow{\text{logic}}$ HProp" quantifies an abstract invariant parameterized by an integer.

1: Function contracts may appear in typing contexts, while typing contexts are involved in the statement function contracts. This form of *impredicativity* is standard in higher-order separation logic [Cha20a].

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

Linear resources The linear part of a typing context contains *resources*. A resource is described by a binding of the form "$y : H$", where H is a *heap predicate*, and where y is a name. For example, "$y : p \rightsquigarrow$ Cell" is a resource. This name y is used in particular to refer to resources in usage maps. A heap predicate H describes ownership of a memory region. When a linear context contains several resources, each resource must describe a disjoint part of the memory. Interestingly, heap predicates guarantee the absence of hidden aliasing.

Table 4.1 gives the grammar of the currently supported heap predicates. These heap predicates have already been discussed in chapter 2, but we introduce here math notations making the type annotations explicit, and give more details in the following paragraphs.

Table 4.1: Grammar of heap predicates. User-defined representation predicates are left to future work.

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

Table 4.2: Syntactic sugar for frequently used heap predicates.

Syntax in OptiC	Syntax in the theory	Desugared version
p `⇝ Array(`n`)`	$p \rightsquigarrow \mathsf{Array}_{\hat{\tau}}(n)$	$\bigstar_{i \in 0..n} p \boxplus i \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$
p `⇝ Matrix1(`n`)`	$p \rightsquigarrow \mathsf{Matrix1}_{\hat{\tau}}(n)$	$\bigstar_{i \in 0..n} p \boxplus \mathsf{mIndex1}(n, i) \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$
p `⇝ Matrix2(`m`,` n`)`	$p \rightsquigarrow \mathsf{Matrix2}_{\hat{\tau}}(m, n)$	$\bigstar_{i \in 0..m} \bigstar_{j \in 0..n} p \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$
p `⇝ Array(`n`,` M`)`	$p \rightsquigarrow \mathsf{Array}(n, M)$	$\bigstar_{i \in 0..n} p \boxplus i \mapsto M(i)$
p `⇝ Matrix1(`n`,` M`)`	$p \rightsquigarrow \mathsf{Matrix1}(n, M)$	$\bigstar_{i \in 0..n} p \boxplus \mathsf{mIndex1}(n, i) \mapsto M(i)$
p `⇝ Matrix2(`m`,` n`,` M`)`	$p \rightsquigarrow \mathsf{Matrix2}(m, n, M)$	$\bigstar_{i \in 0..m} \bigstar_{j \in 0..n} p \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow M(i, j)$
p `⇝ UninitArray(`n`)`	$p \rightsquigarrow \mathsf{UninitArray}_{\hat{\tau}}(n)$	$\bigstar_{i \in 0..n} p \boxplus i \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}$
p `⇝ UninitMatrix1(`n`)`	$p \rightsquigarrow \mathsf{UninitMatrix1}_{\hat{\tau}}(n)$	$\bigstar_{i \in 0..n} p \boxplus \mathsf{mIndex1}(n, i) \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}$
p `⇝ UninitMatrix2(`m`,` n`)`	$p \rightsquigarrow \mathsf{UninitMatrix2}_{\hat{\tau}}(m, n)$	$\bigstar_{i \in 0..m} \bigstar_{j \in 0..n} p \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}$

The resource $p \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$ corresponds to the ownership of a single cell of type $\hat{\tau}$, located at address p. Table 4.2 presents syntactic sugar constructions for commonly used heap predicates. For example, the resource $p \rightsquigarrow \mathsf{Array}_{\hat{\tau}}(n)$ is syntactic sugar for $\bigstar_{i \in 0..n} p \boxplus i \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$. This resource corresponds to the ownership of the set of all the cells in the array. The big-star symbol corresponds to the *iterated separating conjunction* of separation logic. Likewise, $p \rightsquigarrow \mathsf{Matrix1}_{\hat{\tau}}(m)$ is the syntactic sugar for one-dimensional matrices, and $p \rightsquigarrow \mathsf{Matrix2}_{\hat{\tau}}(m, n)$ is syntactic sugar for two-dimensional matrices. This pattern then generalizes for more dimensions.

We leave it to future work to provide mechanisms allowing the user to define *representation predicates* [Rey02] for custom data types.

[Rey02]: Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*

Alternative permissions with models For users wanting to work with functional correctness invariants, instead of heap predicates of the form $p \rightsquigarrow \mathsf{Cell}_{\hat{\tau}}$ we provide the more precise heap predicate $p \mapsto v$ which corresponds to the ownership of a cell located at address p and containing the value v. We call such value v a *model* for the cell. In practice, models might be variables in the pure context, or pure expressions describing a value, like we saw in section 2.3.2. Like for permissions without models, we define $p \rightsquigarrow \mathsf{Array}(n, M)$, $p \rightsquigarrow \mathsf{Matrix1}(n, M)$ and $p \rightsquigarrow \mathsf{Matrix2}(m, n, M)$ as syntactic sugar.

The rest of the heap predicates present in table 4.1 are common both with and without models and are explained in the following paragraphs.

Read-only fractions Following standard separation logic, we represent read-only resources using *fractional resources* [Boy03; Jun+18a]. Intuitively, possessing a non-zero fraction of a memory region gives read-only access to this region. Possessing the full fraction (i.e. 1) of a memory region gives read-write exclusive access to this region. These memory regions are abstracted by

[Boy03]: Boyland (2003), *Checking Interference with Fractional Permissions*
[Jun+18a]: Jung et al. (2018), *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*

heap predicates and therefore fractions can be used on those heap predicates and form read-only linear resources.

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

For simple enough heap predicates[3], such as all those built from constructions presented in table 4.1, possessing both αH and βH is equivalent to possessing the single resource $(\alpha+\beta)H$. In practice, OptiTrust currently only manipulates heap predicates such that this equivalence holds. This means that two fractions of the same heap predicate can always be merged together, to reconstitute bigger fractions.

3: Heap predicates using disjunction or existential quantifiers cannot be merged together in the general case. Indeed, the ressource $\frac{1}{2}(p \rightsquigarrow \text{Cell} \vee q \rightsquigarrow \text{Cell}) \star \frac{1}{2}(p \rightsquigarrow \text{Cell} \vee q \rightsquigarrow \text{Cell})$, does not imply a permission to write in either p or q, while a permission $p \rightsquigarrow \text{Cell} \vee q \rightsquigarrow \text{Cell}$ does. We discuss further in section 7.2 how we could accomodate in the future more complex heap predicates with disjunction or existential quantification.

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

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

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

To simplify carve, merge and unification operations, during typechecking we normalize heap predicates of the form $\bigstar_{i \in R} \alpha H_i$ into $\alpha(\bigstar_{i \in R} H_i)$. While this operation can sometimes be unsound in classical separation logic[4], we decided to use a variant called *unbounded separation logic* [DMS22] to circumvent this issue.

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

[DMS22]: Dardinier et al. (2022), *Fractional resources in unbounded separation logic*

Resources for uninitialized cells Separation logic can guarantee that a program never reads from an uninitialized memory cell.

In order to do so, allocation of a memory cell at address p does not produce the regular heap predicate $p \rightsquigarrow \text{Cell}$ or $p \mapsto v$ but an uninitialized predicate written $p \rightsquigarrow \text{UninitCell}$ in OptiTrust[5].

Then, the specification of the read operation requires a permission of the form $\alpha(p \rightsquigarrow \text{Cell})$ or $\alpha(p \mapsto v)$, neither of which can be extracted from a permission of the form $p \rightsquigarrow \text{UninitCell}$.

On the contrary, the specification of a write operation requires a permission of the form $p \rightsquigarrow \text{UninitCell}$, and produces the full permission $p \rightsquigarrow \text{Cell}$ (or alternatively a permission of the form $p \mapsto v$). In order to be able to execute a write operation when we already have a full permission, we allow a permission $p \rightsquigarrow \text{Cell}$ (or $p \mapsto v$) to be downgraded into $p \rightsquigarrow \text{UninitCell}$ at any time.

5: Some separation logic frameworks reuse the permission $p \mapsto v$ to encode uninitialized permissions by adding a special uninitialized value often denoted $\perp$. Then when reading a cell, one must prove that the value is not $\perp$. We believe that a dedicated uninitialized representation predicate is easier to handle in our case.

More generally, as detailed further on (in section 4.5), when our typechecker encounters a term that requires $p \rightsquigarrow \mathsf{UninitCell}$ in a context where the plain resource $p \rightsquigarrow \mathsf{Cell}$ or $p \mapsto v$ is available, it weakens that plain resource into $p \rightsquigarrow \mathsf{UninitCell}$ on-the-fly. This weakening can also be performed under separating conjunctions. We write $\mathsf{Uninit}(H)$ such weakening operation on an arbitrary heap predicate H. It returns a heap predicate when it succeeds and $\bot$ otherwise. If H is already an uninitialized resource, then $\mathsf{Uninit}(H) = H$.

Once again, we define uninitialized arrays and matrices as syntactic sugar for iterated separating conjunction of uninitialized cells as stated in table 4.2.

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

Magic wand permission As we saw in the case studies, the ghost operation focus that extracts a permission to a specific cell from a permission to the full array also generates a second permission representing the rest of the array. In separation logic, such remainder permission is usually modelled by a heap predicate called *magic wand*. A magic wand is denoted $H_1 \mathrel{-\!\!*} H_2$ where H_2 is a big heap predicate from which H_1 was extracted. The predicate $H_1 \mathrel{-\!\!*} H_2$ can then be recombined with H_1 to get H_2 back. In OptiTrust, such recombination of H_1 with $H_1 \mathrel{-\!\!*} H_2$ is never performed automatically, but can be performed by a ghost operation.

4.2 Construction and operations on typing contexts

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

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

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

$p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}}$. Internally, though, all context items are identified by a variable name.

Bindings of the special result variable In contexts, we use a special variable **res** as a canonical name to denote the result value of an expression. Therefore, if t has a non-void type τ then, in the triple $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, this variable **res** may be bound in Γ' as an alias. The variable **res** also appears in function contracts, to specify properties about the return value of the function. The use of a dedicated name such as **res** is common practice in program verification tools, such as ESC/Java [Fla+02] or Why3 [Fil03].

[Fla+02]: Flanagan et al. (2002), *Extended Static Checking for Java*

[Fil03]: Filliâtre (2003), *Why: a multi-language multi-prover verification tool*

Projection of context components We define two projection functions. For a context $\Gamma = \langle E | F \rangle$, the projection "$\Gamma$.pure" returns E, and the projection "Γ.linear" returns F.

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

Separated conjunction of two contexts We write $F_1 \star F_2$ the disjoint union of two sets of linear resources. Furthermore, for two contexts Γ_1 and Γ_2, we define $\Gamma_1 \circledast \Gamma_2$ as $\langle \Gamma_1.\mathsf{pure},\ \Gamma_2.\mathsf{pure} \mid \Gamma_1.\mathsf{linear} \star \Gamma_2.\mathsf{linear} \rangle$, assuming the variables in this result are well-scoped (that is, Γ_1 and Γ_2 have disjoint domains and the formulas in Γ_2 are well-scoped in $\Gamma_1.\mathsf{pure}$). Observe that $[E] \circledast F = \langle E \mid F \rangle$.

Iterated separating conjunction of a context Consider a linear context F of the form $(y_0 : H_0, \ldots, y_n : H_n)$. We define $\bigstar_{i \in R} F$ as $(y_0 : \bigstar_{i \in R} H_0, \ldots, y_n : \bigstar_{i \in R} H_n)$, that is, the iterated separating conjunction distributes pointwise over the set of linear resources.

We also define an iterated separating conjunction for contexts containing pure resources. This notion is especially useful when working with models. Before giving the formal definition, let us take a step back to understand what such an iterated separating conjunction means in presence of pure resources. Take the context $\Gamma_i = \langle n : \mathsf{int},\ 0 < n < 100 \mid p \boxplus i \mapsto n \rangle$ that asserts that there is an integer n between 0 and 100 such that $p \boxplus i$ points to n. Intuitively, the iterated separating conjunction $\circledast_{i \in R} \Gamma_i$ represents the fact that Γ_i holds on disjoint parts of memory for each i in range R. In that case, this means that each cell $p \boxplus i$ can point to a different integer n. Therefore, the conjunction must allow for a different choice of n for each i. In OptiTrust, we allow that by using a function instead of the scalar n in the conjunction. On our example, this gives $\circledast_{i \in R} \Gamma_i = \langle n : \mathsf{int} \xrightarrow{\mathsf{logic}} \mathsf{int},\ (\forall (i : \mathsf{int})(x_R : i \in R),\ 0 < n(i) < 100) \mid \bigstar_{i \in R} p \boxplus i \mapsto n(i) \rangle$.

Notice that, in $\circledast_{i \in R} \Gamma_i$, the resource n is modelled by a simple function of type $\mathsf{int} \xrightarrow{\mathsf{logic}} \mathsf{int}$ whose values are irrelevant outside the range R. We could alternatively consider modelling n with a dependent function of type $\forall (i : \mathsf{int})(\_ : i \in R),\ \mathsf{int}$, following the same pattern as the hypothesis $0 < n < 100$ in the example. Functions with irrelevant values outside the range simplify resource handling compared to dependent function with range checks by avoiding the need for providing a proof of range inclusion as an extra argument at every coefficient access. However, such use of irrelevant values outside the range only makes sense for bindings with

obviously inhabited types such as int for which irrelevant arbitrary values are easy to find. In this version of OptiTrust, we decided that obviously inhabited types are the program types and the type of fractions frac. In particular, all types in Prop (like $0 < n < 100$ in the example) are not obviously inhabited, and therefore use dependent functions for iterated separating conjunction.

In order to formally define a notion of iterated separating conjunction for contexts that also contain pure resources, we need to generalize slightly our iterated separating conjunction operator: the syntax $\bigstar_{i\in R}^{x_R} H$ allows using a variable x_R representing a proof of the fact that i is in range R (i.e. x_R has type $i \in R$) inside the expression H. This generalized version also distributes pointwise over sets of linear resources. Then, we define the iterated separating conjunction over a context as follows:

$$\circledast_{i\in R}^{x_R} \langle \varnothing \mid F\rangle = \bigstar_{i\in R}^{x_R} F$$

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

Fraction of linear contexts Consider a linear context F of the form $(y_0 : H_0, \ldots, y_n : H_n)$. We define αF as $(y_0 : \alpha H_0, \ldots, y_n : \alpha H_n)$, that is, the read-only fraction distributes pointwise over the set of linear resources.

For a context with pure resources, we can define $\alpha\langle E|F\rangle$ as $\langle E|\alpha F\rangle$. However, this does not give the expected properties for merging two context fractions since, for instance, in general, $\frac{1}{2}\Gamma \circledast \frac{1}{2}\Gamma$ is not the same as Γ. A specific instance where those differ is given by $\Gamma = \langle p : \mathsf{ptr}(\mathsf{int}) \mid p \rightsquigarrow \mathsf{Cell}_{\mathsf{int}}\rangle$. In that case, the pure binding p might refer to different locations in each occurrence of Γ. In the rest of this manuscript, we therefore never take a fraction of a context with pure resources.

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

Specialization of contexts In order to adapt a function contract for a specific call to that function, we introduce a specialization operation. At function calls, function contracts are specialized on the arguments, as well as on the ghost arguments, on which the function is applied. In case of a polymorphic function, type arguments are specialized as well. The specialization operation takes the form $\mathrm{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$. The definition of this operation is fairly technical, yet it is a direct generalization of the process of typechecking function applications in higher-order logics. Rather than presenting a technical definition here, let us illustrate the specialization operation on an example. The technical definition is described in appendix B.

Consider a function f whose input is described by a context $\Gamma = \langle A : \mathsf{Type}, C : \mathsf{Type}, n : \mathsf{int}, p : \mathsf{ptr}(A), b : A, c : C \mid p \rightsquigarrow \mathsf{Matrix1}_A(n)\rangle$, where A and C are type arguments, where p and n denote physical arguments,

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

Note that in general, $\mathsf{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ can fail, because for instance the arguments given for the function call do not have types matching the function contract, or because σ contains bindings that are not in $\Gamma.\mathsf{pure}$. In such cases, the specialization operation generates a typing error.

Renaming on contexts A renaming operation is involved when the programmer explicitly specifies the names to assign to the ghost variables obtained as part of the result of a function call. The operation $\mathsf{Rename}\{\rho\}(\Gamma)$ renames certain keys from Γ. Here, ρ denotes a map that associates resource names to other resource names. The keys from ρ may or may not be bound in Γ. The values from ρ must be fresh from Γ. For example, $\mathsf{Rename}\{x := x',\ y := y'\}(\langle E_1,\ x : \tau,\ E_2 \mid F\rangle)$, where y has no occurrence in E_1, E_2 or F, evaluates to $\langle E_1,\ x' : \tau,\ \mathsf{Subst}\{x := x'\}(E_2) \mid \mathsf{Subst}\{x := x'\}(F)\rangle$. As another example, $\mathsf{Rename}\{y := y'\}(\langle E \mid F_1,\ y : H,\ F_2\rangle)$ evaluates to $\langle E \mid F_1, y' : H, F_2\rangle$.

4.3 Grammar of contracts

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

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

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

OptiC syntax for function contracts As we saw in chapter 2, the OptiTrust user needs to provide function contracts in OptiC by using annotations. In practice a function contract annotation consists of a series of contract clauses, each adding one or several resources to the contract following the pattern shown in the following table:

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

Figure 4.1: Contracts assigned to key primitive functions; $\hat{\tau}$ denotes a C type; a and b denote program variables. $C_{\hat{\tau}}$ is either $\mathsf{UninitCell}_{\hat{\tau}}$, $\mathsf{UninitArray}_{\hat{\tau}}(n)$, $\mathsf{UninitMatrix1}_{\hat{\tau}}(n)$, or $\mathsf{UninitMatrix2}_{\hat{\tau}}(m, n)$, for size expressions m and n. When models are enabled, get and set get a more precise contract described in figure 4.2.

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

Every function contract can be expressed using exclusively a combination of `__requires`, `__ensures`, `__consumes` and `__produces` clauses. Although not strictly needed, the other four clauses `__modifies`, `__preserves`, `__reads` and `__writes` provide syntactic sugar for common patterns, to avoid useless repetitions in contracts. In practice, one clause can specify multiple resources at once. For example, `__requires("x1: int, x2: int");` is interpreted as `__requires("x1: int"); __requires("x2: int");`. In the version of OptiTrust presented in this PhD, the clauses `__modifies` and `__preserves` are identical, but they convey different intentions. In fact, using only one keyword everywhere would be counter-intuitive. The clause `__modifies` should be used for predicates without models because having for example $p \rightsquigarrow \mathsf{Cell}$ both in the pre- and the post-condition allows modifying the value stored in the cell without restoring the initial value at the end. Conversely, the clause `__preserves` should be used for predicates with models because having for example $p \mapsto v$ both in the pre- and the post-condition forces to restore the initial value of the cell at the end of the function.

Contracts for primitive functions Figure 4.1 gives the contracts that we axiomatize for the operations on heap cells—technically, we present not their contracts, but the triples derived from their contracts, to improve readability. These contracts illustrate key mechanisms of the formalism. A heap allocation produces an uninitialized permission over a single cell or a matrix and a permission to free these allocated cells. A write operation requires an uninitialized permission and returns a full permission. A read operation requires a read-only permission and returns it. A free operation requires a permission to free, the associated uninitialized permission and returns nothing. Recall that a full permission can be split into read-only resources, and that it may be downgraded at any time into an uninitialized permission. Additionally, we can see that bindings on **res** appear in output contexts.

When typechecking functional correctness assertions, we need to use a more precise version of the contracts for get and set, as described in the figure 4.2. In that case, a read specifies that the returned value is equal to the current

$$\{[\hat{\tau} : \mathsf{Type}, a : \mathsf{ptr}(\hat{\tau}), \alpha : \mathsf{frac}, v : \hat{\tau}] \circledast \alpha(a \mapsto v)\} \quad \mathsf{get}(a) \quad \{[\mathbf{res} := v : \hat{\tau}] \circledast \alpha(a \mapsto v)\}$$
$$\{[\hat{\tau} : \mathsf{Type}, a : \mathsf{ptr}(\hat{\tau}), b : \hat{\tau}] \circledast a \rightsquigarrow \mathsf{UninitCell}_{\hat{\tau}}\} \quad \mathsf{set}(a, b) \quad \{a \mapsto b\}$$

Figure 4.2: Contracts assigned to get and set primitive functions when models are enabled; $\hat{\tau}$ denotes a C type; a and b denote program variables.

model, and a write returns a full permission with the written value as the model.

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

Contracts for ghost functions In addition to contracts for primitive heap-manipulating functions, OptiTrust provides contracts for primitive ghost functions. For example, the ghost function swap_groups allows swapping two iterators (iterated separating conjunctions). It is involved for example in the loop-swap operation, which is used in our case studies (chapter 2), and which is presented further on in section 6.5. The transformation is specified as shown below, where H is a heap predicate that depends on the two indices i and j. The type range corresponds to the type of loop ranges.

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

The OptiTrust user can define custom ghost functions to factorize repetitive resource-manipulation patterns. Ghost functions are written and type-checked like regular C functions, but their bodies are composed only by calls to other ghost functions, sequences and range-based **for** loops. Importantly, the body of a ghost function does not need to be executed, and simply serves as a proof witness.

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

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

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

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

We call F_{reads} the *shared reads*. In practice this context consists of read-only resources. Resources in F_{reads} cannot refer to the loop index. Each loop

iteration receives a subfraction of each resource in F_{reads}, and must give it back at the end.

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

OptiC syntax for loop contracts Like with function contract annotations, loop contract annotations in OptiC consist of a series of loop contract clauses, some of which are syntactic sugar for common patterns. The loop contract clauses are described by the following table:

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

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

4.4 Entailment

We next introduce the *entailment* judgment, written $\Gamma \Rightarrow \Gamma'$. The entailment judgment can be used to assert that a context Γ obtained at a given program point corresponds to a context Γ' expected at that same point. For example, the context at the end of a function body must entail the context described by the postcondition of that function. The entailment judgment $\Gamma \Rightarrow \Gamma'$ is a declarative judgment, for which we will present our algorithmic implementation in the next section.

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

Usually, in separation logic, the entailment relation is defined on linear resources only. In OptiTrust we extend this standard relation to apply over contexts with pure resources. In the entailment $\langle E_1 \mid F_1 \rangle \Rightarrow \langle E_2 \mid F_2 \rangle$ variables

from the pure context E_1 can be used inside the resources in E_2 and F_2. Intuitively, the aforementioned entailment can be read as "In a context with pure resources E_1, is it possible to choose values for bindings in E_2 such that any memory region described by F_1 is also described by F_2". A formal semantic definition of this entailment relation can be found in definition C.9 in the appendix.

For example, the following entailments hold:

- $\langle x : \mathsf{loc} \mid x \rightsquigarrow \mathsf{Cell} \rangle \Rrightarrow \langle \alpha : \mathsf{frac} \mid \alpha(x \rightsquigarrow \mathsf{Cell}), (1-\alpha)(x \rightsquigarrow \mathsf{Cell}) \rangle$
- $\langle y : \mathsf{loc} \mid y \rightsquigarrow \mathsf{Cell} \rangle \Rrightarrow \langle \mid y \rightsquigarrow \mathsf{UninitCell} \rangle$
- $\langle A : \mathsf{loc},\ m : \mathsf{int},\ n : \mathsf{int} \mid \bigstar_{i \in 0..m} \bigstar_{j \in 0..n} A \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow \mathsf{Cell} \rangle \Rrightarrow \langle \mid \bigstar_{j \in 0..n} \bigstar_{i \in 0..m} A \boxplus \mathsf{mIndex2}(m, n, i, j) \rightsquigarrow \mathsf{Cell} \rangle$
- $\langle n : \mathsf{int},\ n\ \mathsf{even} \mid \rangle \Rrightarrow \langle m : \mathsf{int},\ n = 2m \mid \rangle$.

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

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

4.5 Subtraction

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

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

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

The subtraction operation is implemented following a standard scheme:

(1) The substitution map σ is initialized with bindings that associate each of the pure variables of Γ' to a fresh unification variable.

(2) Each of the linear resources from Γ' are syntactically matched against a corresponding resource from Γ. This process may trigger unifications, resulting in partial or total resolution of certain unification variables.

    If Γ' requests an uninitialized linear resource H' and if Γ contains a resource H such that $\mathsf{Uninit}(H)$ unifies with H', then our algorithm applies this on-the-fly weakening from H to $\mathsf{Uninit}(H)$ to instantiate H'.

(3) The items from Γ that remain at the end are assigned to the frame F.

The carving subtraction operation $\Gamma \ominus \Gamma'$ behaves almost like $\Gamma \boxminus \Gamma'$ but outputs a triple $(E_{\mathsf{frac}}, \sigma, F)$ where σ and F are the same as in core subtraction and E_{frac} is a pure context for generated fractions containing only bindings of the form $\alpha : \mathsf{frac}$. At step (1), E_{frac} is initialized as an empty environment. Compared to the core subtraction, the carving subtraction refines step (2) as follows. If Γ' requests a fractional resource αH, if α is an unconstrained unification variable that denotes a fraction, and if Γ contains a fractional resource $\beta H'$ for some fraction β and where H unifies with H', then our algorithm applies an on-the-fly splitting operation to convert $\beta H'$ into the conjunction of $\alpha' H'$ and $(\beta - \alpha') H'$ for a fresh α' added to E_{frac}. Our algorithm then adds the binding $\alpha := \alpha'$ into σ. The interest of extracting a carved fraction from βH rather than consuming the whole read-only permission βH is that the left-over fraction remains available in Γ, allowing to match other resources of the form $\alpha'' H$ that might appear in the other elements from Γ'. Note that when $\Gamma \ominus \Gamma' = (E_{\mathsf{frac}}, \sigma, F)$, then both the entailments $\Gamma \Rightarrow \Gamma' \circledast \langle E_{\mathsf{frac}} \mid F \rangle$ and $\Gamma \Rightarrow [E_{\mathsf{frac}}] \circledast \mathsf{Specialize}_{\Gamma}\{\sigma\}(\Gamma') \circledast F$ hold.

4.6 Typechecking of logical expressions

A *logical expression* is an expression that may appear in specifications and invariants; technically, a logical expression is an expression whose evaluation terminates and does not depend on the memory state. Logical expressions include program variables (which are always immutable in Optiλ), constant literals, logical propositions, heap predicates, C types, logical types, pure

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

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

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

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

Figure 4.3: Selected rules defining the typing judgment for logical expressions, written $E \vdash t : \tau$. Arithmetic operations such as $t_1 + t_2$ are viewed as functions calls and are therefore handled by the rule LogicApp.

function definitions, and pure function calls. Figure 4.3 shows the main typing rules for logical expressions. The judgment is written $E \vdash t : \tau$, where E is a pure context.

An arithmetic expression (e.g. $t_1 + t_2$) can be considered as a logical expression (e.g. the application of the logical function for addition) if its two arguments are pure. This allows us to express the contract for the primitive arithmetic operations by referring to the corresponding logical expression. For instance, the contract for addition is: $\{[a : \mathsf{int}, b : \mathsf{int}]\}\ (a + b)\ \{[\mathbf{res} := a \hat{+} b : \mathsf{int}]\}$, where for clarity we distinguished the addition operator from the programming language denoted $+$, and the addition operator from the logic denoted $\hat{+}$. Partial primitive arithmetic functions have a contract expressed with the corresponding logical expression, but those require an additional precondition. For example, the contract for division is: $\{[a : \mathsf{int}, b : \mathsf{int}, b \neq 0]\}\ (a/b)\ \{[\mathbf{res} := a \hat{/} b : \mathsf{int}]\}$ where $\hat{/}$ denotes the logical integer division operator. Following standard practice in proof assistants, the operator $\hat{/}$ is defined in the logic as a total function that returns unspecified results when the divisor is equal to zero.

4.7 Typechecking of terms

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

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

In particular, in the algorithmic typechecking, all the logical bindings (ghost variables and pure facts) from Γ are reproduced in Γ'. The pure bindings that appear in Γ' but not in Γ correspond either:

- to the binding for **res**, which denotes the result value produced by t, as explained in section 4.3; or
- to logical bindings (ghost variables and pure facts) that correspond to the pure part of the postcondition of t.

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

Figure 4.4 presents our typing rules. The typing rule for function application handles the particular case where the subterms are program variables (this syntactic restriction for function call is often called *A-normal form*)—the processing of effectful subterms depends on resource usage, and is explained further in section 5.5. The soundness of most of these rules stems from the fact that they correspond to an algorithmic reformulation of the standard

$$\frac{\Gamma.\text{pure} \vdash t : \tau \qquad t \text{ is a literal or a variable}}{\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma \circledast [\mathbf{res} := t : \tau]\}\!\}}\ \textsc{LitOrVar} \qquad \frac{\{\!\{\Gamma_0\}\!\}\ t\ \{\!\{\Gamma_1\}\!\} \qquad \Gamma_2 = \text{Rename}\{\mathbf{res} := x\}(\Gamma_1)}{\{\!\{\Gamma_0\}\!\}\ \mathbf{let}\ x = t\ \{\!\{\Gamma_2\}\!\}}\ \textsc{Let}$$

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

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

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

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

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

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

Figure 4.4: Algorithmic typing rules for establishing triples of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$. These rules are generalized in section 5.3 to derive triples the form $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$, where Δ describes the resource usage.

reasoning rules from separation logic. We next describe the rules individually.

Literals and variables Consider a term t that corresponds either to a program variable or to a literal. In its triple, of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, the output context Γ' is obtained by extending Γ with an alias binding from **res** to t itself. Alias bindings were defined in section 4.2. This is possible since for literals and variables, t is a logical expression and therefore can directly appear in contexts. The type of t is computed by means of the typing judgment for logical expressions, defined in section 4.6.

Let bindings Consider an instruction of the form **let** $x = t$. Recall from section 3.2 that such instructions only appear in sequences. The subexpression t produces a value, hence the output context Γ_1 associated with t binds the special variable **res**. The expression **let** $x = t$ itself does not produce a

value, hence its output context Γ_2 does not bind **res**. However, the output context Γ_2 is extended with a binding on x. Concretely, Γ_2 is obtained by replacing in Γ_1 the bound name **res** with the bound name x.

Sequence of instructions We decompose the treatment of sequences in two rules: a first rule named Seq for handling the sequence of instructions per se, and a second rule named Block for handling the disposal of stack-allocated variables. The rest of this paragraph describes the Seq rule. Consider a sequence $(t_1; ...; t_n; r)$. Starting from an input context Γ_0, each subterm t_i makes the context evolves from Γ_{i-1} to Γ_i. Recall from section 3.2 that each subterm t_i must have unit type (a.k.a. **void** type), else it would have been wrapped into a call to the "ignore" function. The sequence itself may return a value identified by the optional result variable r. If such a result variable is set, the final context is patched to include a **res** binding instead of the original r binding.

Scope blocks The typing rule Block is responsible for collecting the resources that corresponds to stack-allocated variables, when reaching the end of a sequence, that is, the end of their scope. Recall from section 3.2 that stack allocation takes the form **let** $x = \mathsf{ref}(t)$ or **let** $x = \mathsf{stackAlloc}_C()$, with such instructions occurring directly within a sequence[6]. The auxiliary function $\mathsf{StackAllocCells}(t_1, ..., t_n)$ synthesizes, based on the syntax of the terms t_i that appear in the sequence at hand, a separated conjunction of the uninitialized version of all the resources allocated in the sequence $t_1, ..., t_n$. Formally, StackAllocCells and its auxiliary function StackAllocCell are defined as follows:

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

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

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

These resources described by $\mathsf{StackAllocCells}(t_1, ..., t_n)$ are subtracted from the context available at the end of the sequence. Crucially, the subtraction operation checks that the resources indeed appear in the resource set after the execution of the sequence. Doing so ensures, in particular, that the ownership of a stack-allocated piece of data is not stored in another heap predicate at the end of the sequence.

We take a conservative approach for pure typing context scopes: when a sequence is exited, each immutable program variable that goes out of scope is generalized as a ghost variable, and all ghosts variables introduced in the scope are kept. Generalizing immutable program variables as ghost variable is a no-op in practice since all the program variables are already in the pure context. For now, only contract annotations on function, loops and on conditionals act as abstraction barriers that filter pure contexts[7].

7: We might allow specifying a contract on any block in the future to allow for intermediate abstractions, and avoid pure context growing up too much in size.

Function definition Consider a function definition $\textbf{fun}(a_1 : \hat{\tau}_1, ..., a_n : \hat{\tau}_n)_\gamma \mapsto t$, with arguments a_i of type $\hat{\tau}_i$, with body t, and with contract γ. Recall from section 4.3 that the function contract consists of a precondition $\gamma.\mathsf{pre}$ and a postcondition $\gamma.\mathsf{post}$, both described as contexts. The function is a closure that may capture free variables from the current context. In the rule, the pure variables from the current context are described as $\Gamma_0.\mathsf{pure}$.

Note, however, that the function is not allowed to capture linear resources. Hence, the body of the function is typechecked in an environment that consists of the conjunction of Γ_0.pure and γ.pre. Ultimately, the body of the function must produce a context Γ_r that entails the postcondition γ.post. The postcondition of the function definition itself binds **res** with the correct function type (**res** : $\hat{\tau}_f$) and gives its specification hypothesis (Spec(**res**, $[a_1, \ldots, a_n]$, γ)). As explained earlier in section 4.3, this hypothesis captures $\{\gamma.\text{pre}\}$ **res**$(a_1, \ldots, a_n)$ $\{\gamma.\text{post}\}$, which is indeed the triple intended for the function named **res**.

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

Two additional technicalities are involved in the statement of the App rule. They correspond to the handling of optional user-provided annotations, named σ and ρ, that may guide the typechecking of an application. Such annotations are commonly found both in proof assistants and in program verification frameworks. The map σ allows instantiating a subset of the ghost arguments. Indeed, there could be situations where the subtraction operation would fail to infer a unique possible instantiation, by the only means of the unification process. Hence, user annotations are required to resolve the instantiation. In all our case studies, σ is only used on ghost calls. The map ρ corresponds to a renaming map. Its purpose is to give a name to all the resources that are added to the context by the postcondition of the called function, thus avoiding name conflicts. In practice, the map ρ is partially given by user annotations or by transformations and is extended with fresh variable names for each missing entry during the first typechecking of each function application. In all our case studies, ρ is never manually provided, but it is manipulated by certain transformations on some ghost calls.

Note that, in the definition of a function f, recursive calls to f would not typecheck. Indeed, in that case, the context Γ_0 cannot contain a hypothesis of the form Spec$(f, [a_1, \ldots, a_n], \gamma)$. This fact explains how our typechecker guarantees termination. In the future, we could add support for guarded recursion, by extending the rule Fun for function definition with a variant annotation.

Range-based for loops Consider a possibly-parallel **for** loop of the form $\textbf{for}^{\pi}\ (i \in R)_{\chi}\ t$. The typechecking of such a loop is driven by the loop contract annotation χ. The loop body t is typechecked in a context that binds an index i of type int, a hypothesis of type $i \in R$, the variables from χ.vars, the resources χ.excl.pre, (subfractions of) the resources in χ.shrd.reads, and the resources in χ.shrd.inv at the index i. The loop body needs to produce the resources χ.excl.post and χ.shrd.inv at the next loop index, and it needs to give back the resources that it had received from χ.shrd.reads.

The expression R.start corresponds to the first index in the range R, R.step corresponds to the increment between two consecutive indices of the range R, and R.end corresponds to the one-past-end index of the range R. Note that this R.end can differ from the stop index written in the source code denoted by R.stop. For instance, **range**(0, 3, 2).stop = 3 but **range**(0, 3, 2).end = 4.

There are three complications. First, the shared-read resources, described by χ.shrd.reads, are split into $\frac{1}{R.\mathsf{len}}$ subfractions, where R.len denotes the number of iterations associated with the range R. Note that, when typechecking the body of the loop for a particular iteration $i \in R$, the denominator R.len can be assumed to be nonzero—indeed, $i \in R$ is equivalent to $0 \leq i < R.\mathsf{len}$. Second, like for function calls, the instantiation of the contract using the resources from the input environment Γ_0 is computed using a subtraction, involving a frame F as well as an instantiation map σ'. Also, like for function calls, the output context is obtained by invoking the CloseFracs operation. Third, loops, like functions calls, feature optional annotations σ and ρ, which we have omitted from the statement of the rule, for simplicity. The map σ guides how the contract is instantiated in the input environment Γ_0. The map ρ can be used to explicit the names associated with the resources produced by the loop. The two maps are handled in a similar way as in the App rule.

Conditionals Consider a conditional of the form **if** t_1 **then** t_2 **else** t_3. The condition t_1 is evaluated in the input context Γ_0 and produces a context Γ_1. Then, both branches t_2 and t_3 need to typecheck in the context Γ_1. This context needs to be patched to reflect the knowledge that t_1 evaluated to either **true** or **false**, depending on the branch. The patch is implemented by means of the operation Learn{**res** = b}(Γ). This operation applies the following three steps.

(1) If an aliasing binding of the form **res** := v : bool appears in Γ, then the operation replaces this binding with a conventional binding **res** : bool, and extends Γ with an equality [**res** = v].

(2) It specializes the variable **res** with b, that is, it removes the binding **res** : bool, and replaces all occurrences of **res** with the boolean value b.

(3) It applies basic simplifications on the expressions in which **res** has been substituted with b.

For example, assume t_1 is a test of the form `x == y`, and consider the evaluation of Learn{**res** = **true**}(Γ_1). The output context of t_1 contains the alias binding **res** := (x==y) : bool. At step (1), this alias binding is replaced with an equality **res** = (x==y). At step (2), **res** is replaced with **true**, hence the equality becomes **true** = (x==y). At step (3), this hypothesis is rewritten as the logical equality x = y.

The **then** branch t_2 produces an output context Γ_2, and likewise the **else** branch t_3 produce an output context Γ_3. What should be the output context of the entire conditional **if** t_1 **then** t_2 **else** t_3? It must be a context, call it Γ_r, that both Γ_2 and Γ_3 entail. This context Γ_r is usually called the *join* context in program logics. In general, there is no way to automatically infer join contexts—it is almost as hard as inferring contracts for loops. Therefore, typechecking and verification tools must resort to a combination of user-provided annotations and heuristics. For now, we assume join contexts to be provided by the user. In our box-blur case study (section 2.1), the conditionals appear in terminal position in the body of a function, hence our typechecker can simply instantiate the join context using the (user-provided)

postcondition of that function. We leave it to future work to devise heuristics well-suited for our typesystem, in order to reduce the number of situations where OptiTrust users need to provide annotations.

4.8 Type soundness

The purpose of this section is to present formal statements that reflect the design principles of our typesystem. This section may be safely skipped for a first read. A number of auxiliary definitions, such as the evaluation rules or the satisfaction of a linear resource by a heap fragment, may be found in the appendix.

We follow the standard approach of justifying soundness of a separation logic by providing a semantic interpretation of triples. The general pattern asserts that: "a triple holds if and only if, in any input state satisfying the precondition (i.e. the input context), the evaluation of the term terminates and produces an output state satisfying the postcondition (i.e. the output context)". This statement relies on two central ingredients. First, a definition of the semantics of a term. Second, a definition of what it means for a program state to satisfy a context.

We formalize the semantics using an *omni-big-step* evaluation judgment [Cha+23]. This judgment has been shown to simplify proofs of the frame rule of separation logic, and proofs of compiler correctness results. Concretely, the judgment $t/(s, m) \Downarrow Q$ asserts that the term t, in an input program state (s, m), evaluates to output program states that belong to the set Q. A program state, written (s, m), consists of an immutable stack s and a store m. If t produces an output value, then this value is bound in the output immutable stack to the dedicated name **res**. For simplicity, we focus on total correctness: $t/(s, m) \Downarrow Q$ asserts that all possible evaluations of the term t do terminate, without error[8]. The evaluation rules may be found in appendix A.

[Cha+23]: Charguéraud et al. (2023), *Omnisemantics: Smooth Handling of Nondeterminism*

8: As explained in the omnisemantics paper [Cha+23], the omni-big-step evaluation judgment is related to the standard big-step judgment via the following equivalence:

Theorem 4.8.1: Equivalence between omni-big-step and big-step reductions

$$t/(s, m) \Downarrow Q \iff \left(\begin{array}{l}\text{all possible executions of } t \\ \text{terminate without error}\end{array}\right) \wedge \left(\begin{array}{l}\forall (s', m'), \\ t/(s, m) \Downarrow (s', m') \\ \qquad \Longrightarrow\ (s', m') \in Q\end{array}\right)$$

More concretely, in program states, the immutable stack s is a map from program variables to values. The store m is a map from memory addresses to a possibly uninitialized value and a mode. Such mode can be either RO (read-only) when the memory location is shared between several threads and can only be read, or RW (read-write) when the memory location is only accessible from the current execution thread.

Let us now focus on context satisfaction. As usual in separation logic that involves fractional permissions (or more general forms of ghost state), one asserts that a program state satisfies a context if and only if there exists a *logic state*, which consists of this program state augmented with additional ("ghost") information, such that this logical state satisfies the context. A *logical* state is one that may *satisfy* a context Γ. We define further an elision function that extracts a program state from a logical state. We first describe the representation of a logical state.

A logical state consists of a logical stack, written σ, and a logical store, written μ. A logical stack is similar to a program stack except that it includes additional bindings for ghost variables. A logical store is similar to a program store except that every memory location is tagged with a positive fraction, written α instead of a mode. As standard in realizations of separation logic, a fraction less than one corresponds to a read-only permission. Since we use unbounded separation logic [DMS22] fractions greater than one can occur in logical stores, but a logical store containing such a fraction does not correspond to any program state.

[DMS22]: Dardinier et al. (2022), *Fractional resources in unbounded separation logic*

As said, a context Γ corresponds to a specification of a logical state. We say that a logical state (σ, μ) satisfies a context Γ of the form $\langle E \mid F \rangle$, and write $(\sigma, \mu) \in \Gamma$, if the bindings in σ have types that correspond to the bindings in E, and if the memory cells described by μ correspond to the linear resources described in F. If additionally, the logical store μ does not contain any fraction strictly greater than one (i.e. it corresponds to a program state), we say that μ is *bounded*, and we write $(\sigma, \mu) \overline{\in} \Gamma$. The technical details of the definitions of $(\sigma, \mu) \in \Gamma$ and $(\sigma, \mu) \overline{\in} \Gamma$ are given in appendix C.

To state the semantic interpretation of triples, we need a projection function for extracting a program state out of a logical state. We write $\sigma_{|\text{prog}}$ the operation that converts a logical stack σ into a program stack s by restricting the entries to program variables, or, equivalently said, by removing entries associated with ghost variables. We write $\mu_{|\text{prog}}$ the operation that turns a bounded logical store μ into a program store m by replacing fraction with the corresponding mode:

$$\mu_{|\text{prog}} = \left\{ l \mapsto (\mathcal{M}, v) \;\middle|\; \exists \alpha,\, \mu(l) = (\alpha, v) \wedge \mathcal{M} = \begin{cases} \text{RO} & \text{if } \alpha < 1 \\ \text{RW} & \text{if } \alpha = 1 \end{cases} \right\}$$

This operation is not defined on unbounded logical stores. By leveraging the two operations, we define $(\sigma, \mu)_{|\text{prog}}$ as $(\sigma_{|\text{prog}}, \mu_{|\text{prog}})$, to convert a logical state into a program state.

Before defining triples, we introduce $\mathsf{AcceptableStates}(\sigma, \mu, \Gamma')$ to denote the set of program output states satisfying the postcondition Γ' and satisfying certain constraints with respect to the input state (σ, μ). The set $\mathsf{AcceptableStates}(\sigma, \mu, \Gamma')$ corresponds to the set of states that are the projection of a logical state (σ', μ') such that:

1. the logical state (σ', μ') satisfies the specification Γ' with μ' bounded, and
2. the read-only restriction of μ' is identical to the read-only restriction of μ, and
3. the stacks in σ' and σ agree on the intersection of their domain.

To formalize the second constraint, we let $\mathsf{OnlyRO}(\mu)$ denote the restriction of the logical store μ to the cells that are tagged with a fraction strictly less than 1, that is, as $\{l \mapsto (\alpha, v) \mid \mu(l) = (\alpha, v) \wedge \alpha < 1\}$. We then define:

Definition 4.8.2: Acceptable states

$$\mathsf{AcceptableStates}(\sigma, \mu, \Gamma') \quad = \quad \left\{ (\sigma', \mu')_{|\text{prog}} \;\middle|\; \begin{array}{ll} & (\sigma', \mu') \overline{\in} \Gamma' \\ \wedge & \mathsf{OnlyRO}(\mu) = \mathsf{OnlyRO}(\mu') \\ \wedge & \forall x \in \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma'),\, \sigma(x) = \sigma'(x) \end{array} \right\}$$

We are now ready to define logical triples, written $\{\Gamma\}\ t\ \{\Gamma'\}$. Such a triple asserts that for any bounded logical state satisfying Γ, starting in the program state that corresponds to this logical state, all executions of t terminate and produce output states that belong to the set $\mathsf{AcceptableStates}(\sigma, \mu, \Gamma')$. The latter means that an output state must satisfy Γ', must preserve read-only entries, and must feature an output stack that agrees with the input stack.

Definition 4.8.3: Logical triples

$$\{\Gamma\}\ t\ \{\Gamma'\} \quad = \quad \forall (\sigma, \mu) \overline{\in} \Gamma,\, t/(\sigma, \mu)_{|\text{prog}} \Downarrow \mathsf{AcceptableStates}(\sigma, \mu, \Gamma')$$

The fundamental property of separation logic is the frame rule, which we prove correct for our logical triples in appendix D. The contexts involved here are dependently-typed, hence we need additional assumptions to ensure that the composed contexts are *well-typed*, in the sense that every variable that appears in a type or a resource is properly bound earlier in the context, and that all the types that appear in the context are themselves well-typed. (Well-typed contexts are formalized by definition D.3 in appendix.) The statement of the frame rule is thus as follows:

Theorem 4.8.4: Frame property for logical triples

$$\{\Gamma\}\ t\ \{\Gamma'\} \land \Gamma \circledast \Gamma'' \text{ is well-typed} \land \Gamma' \circledast \Gamma'' \text{ is well-typed} \implies \{\Gamma \circledast \Gamma''\}\ t\ \{\Gamma' \circledast \Gamma''\}$$

Our typing rules presented earlier on in this section are designed as algorithmic variants of the standard typing rules of separation logic. There is one of our rules for which our presentation is not quite standard: the rule For, which handles the typechecking of **for** loops by leveraging our loop contracts. For this rule, we show in appendix E that it can be derived from two standard separation logic rules: the rule for sequential loops with an invariant, and the rule for parallel loops that split resources across iterations. Overall, the soundness of our algorithmic typing rules stems from the soundness of the standard typing rules of separation logic. Soundness is formally stated as an implication from algorithmic triples to semantic triples:

Proposition 4.8.5: Soundness of the algorithmic typing rules

$$\{\!|\Gamma|\!\}\ t\ \{\!|\Gamma'|\!\} \implies \{\Gamma\}\ t\ \{\Gamma'\}$$

We leave to future work the completion of a mechanized proof of this statement.

Computing program resources: Usage maps

5

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

The second goal of this chapter is to formalize the *triple minimization* operations, which plays a central role in the typechecking of function calls involving effectful subexpressions. Triple minimization will also be useful later on to minimize the loop contracts produced by transformations. Section 5.4 presents the triple minimization procedure. Section 5.5 presents the typing rule for subexpressions—this typing rule applies as a preprocessing before the App rule presented earlier. Section 5.6 presents formal statements about the contents of usage maps.

5.1 Grammar of usage maps

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

- "x : required" means that x is a pure resource in Γ that was used during the typing of t.
- "x : ensured" means that x is a pure resource added to the context Γ' during the typing of t. In such a situation, x is not bound in Γ.
- "y : full" can arise when Γ contains a linear resource "$y : H$", for some predicate H. The usage "y : full" means that this resource is consumed during the typing of t. As a result y is not bound in Γ'. Even if t produces a linear resource with the same predicate H, this new occurrence of H is assigned a fresh name, distinct from y.
- "y : uninit" is similar to "y : full" but moreover captures the information that t needs not read the original contents of the memory cells associated with the resource named y. In particular, if t performs a write operation on all cells described by y before any read operation on y, then the usage of y is uninit.
- "y : splittedFrac" can arise when Γ contains a linear resource "$y : H$", for some predicate H. The usage "y : splittedFrac" means that t uses an unspecified subfraction of this resource. In such a situation, the name y is bound both in Γ and in Γ'. It may be the case, however, that the resource named y carries different fractions in Γ and Γ'.
- "y : joinedFrac" can arise when Γ contains a linear resource of the form "$y : (\alpha - \beta_1 - \ldots - \beta_n)H$". The usage "$y$: joinedFrac" means that:

1. the linear resource named y is not used by t
2. t produced a resource of the form $(\beta_i - \gamma_1 - \ldots - \gamma_m)H$
3. these two resources are merged, and the result appears in Γ' under the name y

If a single merge operation is applied, then the resulting resource is $y : (\alpha - \beta_1 - \ldots - \beta_{i-1} - \gamma_1 - \ldots - \gamma_m - \beta_{i+1} - \ldots - \beta_n)H$. (Recall section 4.1.)

- "y : produced" means that the linear resource y has been produced by t. In this case, y is the name of a linear resource in Γ', and does not occur in Γ.
- If a resource name is bound in Γ but not in Δ, then its absence indicates that the corresponding resource is not touched by t. Such a resource is bound under the same name in Γ and Γ'.

5.2 Operations on usage maps

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

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

Intersection and filtering We define:

$$\begin{aligned} \Delta_1 \cap \Delta_2 &= \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) \\ \Gamma\restriction\cdot\Delta &= \Gamma\restriction\cdot\mathsf{dom}(\Delta) \\ \Gamma\backslash\Delta &= \Gamma\restriction\cdot(\mathsf{dom}(\Gamma)\backslash\mathsf{dom}(\Delta)) \end{aligned}$$

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

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

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

$\Delta_1; \Delta_2$	$\emptyset$	required	ensured
$\emptyset$	$\emptyset$	required	ensured
required	required	required	$\bot$
ensured	ensured	ensured	$\bot$

$\Delta_1; \Delta_2$	$\emptyset$	full	uninit	splittedFrac	joinedFrac	produced
$\emptyset$	$\emptyset$	full	uninit	splittedFrac	joinedFrac	produced
full	full	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
uninit	uninit	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
splittedFrac	splittedFrac	full	full	splittedFrac	splittedFrac	$\bot$
joinedFrac	joinedFrac	full	uninit	splittedFrac	joinedFrac	$\bot$
produced	produced	$\emptyset$	$\emptyset$	produced	produced	$\bot$

Figure 5.1: Tables for sequential composition of two usage maps, for pure and for linear resources. For example, in the second table, the cell on the row "splittedFrac" and on the column "full" expresses that if "x : splittedFrac" is a binding from Δ_1 and "x : full" is a binding from Δ_2, then "x : full" is a binding in $\Delta_1; \Delta_2$. The input or output $\emptyset$ corresponds to cases where the usage map contains no binding for the resource name considered. The output $\bot$ corresponds to cases that cannot arise according to our typechecking rules.

Formally, the usage composition operation $\Delta_1; \Delta_2$ is defined by merging the two usage maps pointwise by resource name, using the table shown in figure 5.1 to compute the *combined usage* in case a same resource name is bound both in Δ_1 and Δ_2.

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

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

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

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

5.3 Computing usage maps

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

- One entry required for each pure variable of Γ_1 mentioned in σ.
- One entry uninit or full for each linear resource of Γ_1 that was unified with a resource of Γ_2. The entry is uninit if the resource in Γ_2 is composed only of uninitialized cells. Otherwise, it is a full.

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

Usage of a CloseFracs When closing fractions, we need to add entries to the usage map to account for the modifications on the context. We try to do so in a way that preserves as much information as possible. When CloseFracs finds a possible reduction on two resources $y_1 : (\alpha - \beta_1 - ... - \beta_n)H$ and $y_2 : (\beta_i - \gamma_1 - ... - \gamma_m)H$ it keeps the name y_1 for the merged resource $(\alpha - \beta_1 - ... - \beta_{i-1} - \gamma_1 - ... - \gamma_m - \beta_{i+1} - ... - \beta_n)H$. On the one hand, the resource y_2 disappears from the context. Therefore, we have to put the usage y_2 : full in the usage map. On the other hand, the resource y_1 remains in the context. Since the absence of y_1 would not have blocked the typechecking, it gets the usage t_1 : joinedFrac. Note this is currently the only way joinedFrac usage are generated. Note also that the order of reduction does not matter for the final usage map (all the fractions that disappear will have a usage full, and all the fractions that got bigger will have a usage joinedFrac).

Computing usage during term typing In order to produce triples of the form $\{\!\{\Gamma\}\!\}\ t\ \{\!\{\Gamma'\}\!\}$, we need to patch our typing rules to record the usage information.

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

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

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

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

For the interested reader, we now explain how usage maps are computed in practice. Instead of rewriting each typing rule with explicit usage maps, which would be quite verbose, we simply explain how the rules are extended. We reuse the variables names of the rules described in figure 4.4.

- For the rule Seq, if each instruction t_i has a usage map Δ_i, the usage map of the sequence Δ is given by:

$$\Delta = \begin{cases} \text{Rename}\{r := \textbf{res}\}(\Delta_1; ...; \Delta_n) & \text{if } r \neq \emptyset \\ (\Delta_1; ...; \Delta_n) & \text{if } r = \emptyset \end{cases}$$

- For the rule Block, if we name Δ_r the usage map of the sequence, and Δ_c the usage map of the subtraction of StackAllocCells, the usage map of the whole block is $(\Delta_r; \Delta_c)$.

- For the rule Fun, if we name Δ_1 the usage map of the function body, Δ_2 the usage map of the subtraction $\Gamma_1 \boxminus \gamma.\mathsf{post}$, and S the generated specification hypothesis, then the usage map of the function definition is $((\Delta_1; \Delta_2)|\cdot\mathsf{dom}(\Gamma_0.\mathsf{pure})) \cup \{x : \mathsf{required} \mid x \in \mathsf{fv}(\gamma)\} \cup \{\mathbf{res} : \mathsf{ensured}, S : \mathsf{ensured}\}$. Indeed, viewed from outside the only dependencies of the function definition are the pure resources captured from the surrounding context.

- For the rule App, if Δ_σ is a usage map containing an entry required for each x_i and each pure resource from Γ_0 mentioned in σ, Δ_p is the usage map of the subtraction on Γ_0, Δ_q is a usage map containing one produced (resp. ensured) for each linear (resp. pure) resource in Γ_q, and Δ_f the usage map of the CloseFracs operation, the usage map of the application is $(\Delta_\sigma; \Delta_p; \Delta_q; \Delta_f)$.

- For the rule For, only the outer contract instantiation and the required pure variables needed to typecheck the loop body are considered for computing the usage map. If $\Delta^{\mathsf{out}}_{\mathsf{pre}}$ is the usage map of the subtraction $\Gamma_0 \ominus \Gamma^{\mathsf{out}}_{\mathsf{pre}}$, $\Delta^{\mathsf{in}}_{\mathsf{body}}$ is the usage of the body of the loop, $\Delta^{\mathsf{in}}_{\mathsf{post}}$ is the usage of the subtraction on $\Gamma^{\mathsf{in}}_{\mathsf{post}}$, $\Delta^{\mathsf{out}}_{\mathsf{post}}$ is a usage map containing one produced (resp. ensured) for each linear (resp. pure) resource in $\Gamma^{\mathsf{out}}_{\mathsf{post}}$, and Δ_r is the usage of the CloseFracs operation, then $(\Delta^{\mathsf{out}}_{\mathsf{pre}}; ((\Delta^{\mathsf{in}}_{\mathsf{body}}; \Delta^{\mathsf{in}}_{\mathsf{post}})|\cdot\mathsf{dom}(\Gamma_0.\mathsf{pure})); \Delta^{\mathsf{out}}_{\mathsf{post}}; \Delta_r)$ is the usage map of the **for** loop. Note that the $((\Delta^{\mathsf{in}}_{\mathsf{body}}; \Delta^{\mathsf{in}}_{\mathsf{post}})|\cdot\mathsf{dom}(\Gamma_0.\mathsf{pure}))$ part of this usage map correspond to the usage of pure resources from outside the loop in the body of the loop (they all have a required usage kind).

- For the rule If, applied to a conditional $\mathbf{if}_{\Gamma_r}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$, it is always sound (though possibly imprecise) to combine the usage map Δ_0 of the condition expression t_1 to another usage map Δ_1 that gives a full usage to each linear resource in Γ_1 (the output context of t_1) and a usage map Δ_r that contains a produced usage for each linear resource of Γ_r. For the usage of pure resources, we name Δ_2 (resp. Δ_3) the required usage from t_2 (resp. t_3). Then, we take all the pure facts from Γ_r that are not in Γ_1 as ensured in a usage map Δ'_r. In summary, we compute the usage map of the whole conditional as $(\Delta_0; \Delta_1; \Delta_2; \Delta_3; \Delta_r; \Delta'_r)$.

5.4 Minimization of triples

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

- $\hat{F}$ is the *minimized linear precondition*: a linear context containing resources from $\Gamma.\mathsf{linear}$ that are needed to typecheck t.

- $\hat{F}'$ is the *minimized linear postcondition*: a linear context produced after typechecking t if we give only $\hat{F}$ as the linear precondition.

- F^{framed} is the *maximal frame*: a linear context of resources from $\Gamma.\text{linear}$ that were superfluous in the typechecking of t. It means resources in F^{framed} can be framed during the typechecking of t. Since these resources are not touched by t, they must also occur in $\Gamma'.\text{linear}$.
- E^{fracs} is the *generated fraction set*: a set of pure fractions that are created by the Minimize algorithm to give only an arbitrary subfraction of the resource in $\Gamma.\text{linear}$ in $\hat{F}$ when such a fractional resource suffices to typecheck t.

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

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

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

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

5.5 Typechecking of order-irrelevant subexpressions

We next explain how to leverage the minimization procedure for typechecking functions calls that are not in A-normal form, but possibly include effectful subexpressions. In C, and in our subset OptiC, the arguments of a function call may be evaluated in an arbitrary order. The fact that the order is not fixed is interesting because it leaves additional flexibility for optimizations. Our typesystem checks that, for well-typed OptiTrust programs, the order of evaluation is indeed irrelevant. To that end, we consider a sufficient condition: that the arguments can be evaluated in parallel, in the sense that the side effects performed by one argument should not interfere with the evaluation of any other argument.

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

even more flexibility for transformations compared to undefined evaluation order.

The rule Subexpr reduces the typechecking of a term with possibly effectful subexpressions to the typechecking of a term whose subexpressions are program variables. In particular, the rule may be used to compute the output context associated with a call of the form $f(t_1, ..., t_n)$ in an input context Γ_0, by reducing the problem to the typechecking of a call of the form $f(x_1, ..., x_n)$, in an input context Γ_p that binds the fresh variables x_i.

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

The goal of the rule Subexpr is to distribute the linear resources from the input context Γ_0 between the subterms t_i. If several subterms read the same resource, then this resource needs to be split. If one subterm reads a resource and another subterm modifies that same resource, the rule must fail to apply. The key idea is to typecheck the subterms one after the other, taking advantage of the Minimize operation to remove the minimal amount of resources from the input context, thereby leaving as many resources as possible for the remaining subterms.

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

Appendix G presents an example application of this rule.

5.6 Formal properties of usage maps

To conclude this section, we present three propositions that specify the contents of usage maps computed by our typing algorithm. These propositions have guided all our definitions. We claim that these propositions hold by design; we leave to future work a thorough mechanized proof of the claims.

Consider an algorithmic triple $\{\!\{\Gamma\}\!\}\ t^{\Delta}\ \{\!\{\Gamma'\}\!\}$, where Γ decomposes as $\langle E \mid F\rangle$ and Γ' decomposes as $\langle E' \mid F'\rangle$. The first proposition explains how F and F' are partitioned by the usage map.

Proposition 5.6.1: Decomposition by usage

$$F = F\text{⌊}\cdot\Delta.\text{full} \star F\text{⌊}\cdot\Delta.\text{uninit} \star F\text{⌊}\cdot\Delta.\text{splittedFrac} \star F\text{⌊}\cdot\Delta.\text{joinedFrac} \star F\backslash\Delta$$
$$F' = F'\text{⌊}\cdot\Delta.\text{produced} \star F'\text{⌊}\cdot\Delta.\text{splittedFrac} \star F'\text{⌊}\cdot\Delta.\text{joinedFrac} \star F'\backslash\Delta$$

The second proposition explains how E' extends E, and how the frame resources from F are preserved in F'. Besides, the proposition captures the fact that a resource with usage splittedFrac or joinedFrac in F must also appear in F', albeit with a possibly different fraction.

Proposition 5.6.2: Preserved parts of typing contexts

$$
\begin{array}{ll}
 & \{\!\{\langle E \mid F\rangle\}\!\}\ t^{\Delta}\ \{\!\{\langle E' \mid F'\rangle\}\!\} \\
\Longrightarrow & E' = (E,\ E'\restriction\cdot\Delta.\mathsf{ensured}) \\
\wedge & F'\backslash\Delta = F\backslash\Delta \\
\wedge & \left(\forall y,\ \forall H,\ \begin{array}{l}(\exists\alpha,\ (y : \alpha H) \in F\restriction\cdot\Delta.\mathsf{splittedFrac}) \\ \quad\Longleftrightarrow\ (\exists\beta,\ (y : \beta H) \in F'\restriction\cdot\Delta.\mathsf{splittedFrac})\end{array}\right) \\
\wedge & \left(\forall y,\ \forall H,\ \begin{array}{l}(\exists\alpha,\ (y : \alpha H) \in F\restriction\cdot\Delta.\mathsf{joinedFrac}) \\ \quad\Longleftrightarrow\ (\exists\beta,\ (y : \beta H) \in F'\restriction\cdot\Delta.\mathsf{joinedFrac})\end{array}\right)
\end{array}
$$

The third proposition explains that the entries of the usage map Δ imply that the term t may be well-typed in a context with smaller footprint. If a resource H appears in F but is not used, then it is omitted. If a resource H appears in F but is used only as uninit (i.e. the corresponding cells are written before being read), then the resource is replaced with its uninitialized counterpart. If a resource H is only read, then it is replaced with a fractional resource αH, where α is a constant that can be chosen arbitrarily small. These operations are formally captured in the following statement, which also covers additional complications related to the case where a set of input resources are split or merged together for producing certain output resources. Below, $\{\hat{\Gamma}\}\ t\ \{\hat{\Gamma}'\}$ corresponds to a semantic triple, a notion introduced in section 4.8 and the $\hat{F}$ variables are explained afterwards.

Proposition 5.6.3: Minimization with usage maps

$$
\begin{array}{ll}
 & \{\!\{\langle E \mid F\rangle\}\!\}\ t^{\Delta}\ \{\!\{\langle E' \mid F'\rangle\}\!\} \\
\Longrightarrow & \forall\alpha,\ \exists\hat{F}^{\mathsf{SP}},\ \exists\hat{F}^{\mathsf{ST}},\ \exists\hat{F}^{\mathsf{JS}},\ \exists\hat{F}^{\mathsf{JF}}, \\
 & \quad \mathsf{let}\ \hat{\Gamma} = \left\langle E\restriction\cdot\Delta.\mathsf{required} \;\middle|\; \begin{array}{l} F\restriction\cdot\Delta.\mathsf{full} \\ \star\ \mathsf{IntoUninit}(F\restriction\cdot\Delta.\mathsf{uninit}) \\ \star\ \alpha(F\restriction\cdot\Delta.\mathsf{splittedFrac})\end{array}\right\rangle\ \mathsf{in} \\
 & \quad \mathsf{let}\ \hat{\Gamma}' = \left\langle \begin{array}{l} E\restriction\cdot\Delta.\mathsf{required}, \\ E'\restriction\cdot\Delta.\mathsf{ensured}\end{array} \;\middle|\; \begin{array}{l} F'\restriction\cdot\Delta.\mathsf{produced} \\ \star\ \hat{F}^{\mathsf{SP}} \star \hat{F}^{\mathsf{JS}} \star \hat{F}^{\mathsf{JF}}\end{array}\right\rangle\ \mathsf{in} \\
 & \qquad \{\hat{\Gamma}\}\ t\ \{\hat{\Gamma}'\} \\
 & \wedge\ \alpha(F\restriction\cdot\Delta.\mathsf{splittedFrac}) \Leftrightarrow \hat{F}^{\mathsf{SP}} \star \hat{F}^{\mathsf{ST}} \\
 & \wedge\ F'\restriction\cdot\Delta.\mathsf{splittedFrac} \Leftrightarrow (1-\alpha)(F\restriction\cdot\Delta.\mathsf{splittedFrac}) \star \hat{F}^{\mathsf{SP}} \star \hat{F}^{\mathsf{JS}} \\
 & \wedge\ F'\restriction\cdot\Delta.\mathsf{joinedFrac} \Leftrightarrow (F\restriction\cdot\Delta.\mathsf{joinedFrac}) \star \hat{F}^{\mathsf{JF}}
\end{array}
$$

We explain the role of the $\hat{F}^X$ variables at a high level, by means of example:

- Assume a resource $y : (\beta - \gamma)H$ from F with usage joinedFrac in Δ meaning that t does not read this resource. It must be the case that t produces (directly or indirectly) a resource γH that is immediately merged into y. This produced resource appears in $\hat{F}^{\mathsf{JF}}$, short for *joined-framed*.
- Assume a resource $y : (\beta - \gamma)H$ from F with usage splittedFrac in Δ meaning that t reads this resource. Assume moreover t produces a resource γH that is immediately merged into y. This produced resource appears in $\hat{F}^{\mathsf{JS}}$, short for *joined-split*.
- Assume a resource $y : \beta H$ from F with usage splittedFrac in Δ. In the minimized triple for t, which takes an arbitrarily-small fraction α of the splittedFrac resources, (up to) two subfractions of $\alpha\beta H$ may

be involved. A first subfraction corresponds to a subresource of y that t does not alter; this subfraction appears in $\hat{F}^{\mathsf{SP}}$, short for *split-preserved*. A second subfraction corresponds to a subresource of y that t alters; this subfraction appears in $\hat{F}^{\mathsf{ST}}$, short for *split-transformed*. The line $\alpha(F|\cdot\Delta.\mathsf{splittedFrac}) \Leftrightarrow \hat{F}^{\mathsf{SP}} \star \hat{F}^{\mathsf{ST}}$ captures that the splittedFrac resources from F divide between $\hat{F}^{\mathsf{SP}}$ and $\hat{F}^{\mathsf{ST}}$.

Again, we leave it to future work to carry out a mechanized proof of these propositions.

Implementation of trustworthy transformations

6

In this chapter, we explain how OptiTrust leverages resource typing information to check the correctness of the transformations requested by the programmer. The aim of this section is not to cover all the transformations implemented in OptiTrust, but to present a representative subset thereof. We focus in particular on transformations that leverage the resource information in an interesting way. All the transformations presented in this section are invoked multiple times in our case studies from chapter 2.

In OptiTrust, most code transformations support two modes of operation: *semantic preservation* and *specification preservation*. The main difference between those modes comes from the trust model and the complexity of the correctness analysis. With semantic preservation, transformations rely on a conservative *correctness criterion* checked by the transformation implementation. With specification preservation, transformations themselves do not need to check anything, they are not allowed to change the specification of the top-level functions, and they fail if their output does not typecheck. This specification preservation mode only makes sense in presence of full functional correctness invariants, because transformations are considered correct whenever their output satisfies the same top-level specification. In both cases, the transformations are responsible for preserving annotations so that their output code typechecks whenever their conditions of applicability are met.

Also recall that, in any case, we only need to check the correctness of *basic* transformations, because *combined* transformations are defined as the composition of basic transformations. Unless said otherwise, all transformations presented in this section have the same implementation in semantic and specification preservation modes. In specification preservation mode, the correctness criterion checks are skipped and replaced by a typechecking of the output code. That said, in semantic preservation mode, a successful typechecking of the output code may or may not be required by the correctness criterion. Nevertheless, even when typechecking is not required for checking the correctness, OptiTrust needs to re-typecheck the program after every transformation, in order to allow the application of subsequent transformations.

A number of *basic* transformation might seem "simple" to the reader. This simplicity is precisely a strength of OptiTrust. As explained in the introduction, we aim to minimize the trusted code base, by considering the simplest possible *basic* transformations and by implementing as many transformations as possible as composition of *basic* transformations. Other transformations are more involved. In fact, for certain loop transformations, we have considered only simplified correctness criterions, which we could further generalize in future work.

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

Notation for well-typed programs Transformations leverage typing information, not only for checking correctness, but also for guiding the generation of the output code. Recall from the previous section that our typechecking algorithm computes, for every subterm t, its input context Γ_1, its output context Γ_2, and its usage map Δ, establishing triples of the form $\{\!\{\Gamma_1\}\!\}\ t^{\Delta}\ \{\!\{\Gamma_2\}\!\}$. In this section, we use an alternative syntax, better-suited for describing the input of transformations. If t denotes an instruction, we write $\Gamma_1\ t;\ \Delta\ \Gamma_2$ as straight-line syntax for $\{\!\{\Gamma_1\}\!\}\ t^{\Delta}\ \{\!\{\Gamma_2\}\!\}$, where any of Γ_1, Δ, or Γ_2 can be omitted if not needed by the transformation.

Groups of instructions Some transformations operate on groups of consecutive instructions. We let the meta-variable T range over a (possibly empty) group of instructions. We generalize our alternative syntax by writing $\Gamma_1\ T;\ \Delta\ \Gamma_2$, where Γ_1 and Γ_2 denotes the initial and final contexts, and Δ denotes the *composition* of the usages from the group of instructions, as defined in section 5.3:

$$\boxed{\Gamma_0\ T;\ \Delta\ \Gamma_n} \quad = \quad \boxed{\Gamma_0\ t_1;\ \Delta_1\ \Gamma_1\ t_2;\ \Delta_2\ \Gamma_2\ \ldots\ \Gamma_{n-1}\ t_n;\ \Delta_n\ \Gamma_n}$$

where $T = t_1; t_2; \ldots; t_n$
and $\Delta = \Delta_1; \Delta_2; \ldots; \Delta_n$

Program contexts Transformations generally apply to a program subterm, that is, apply under a *program context*. Unlike evaluation contexts, program contexts can reach subterms that are not in evaluation position. We let the meta-variable $\mathcal{E}$ range over program contexts. For example, evaluating a subexpression $1 + 1$ that appears in a program context $\mathcal{E}$ is described as the transition from $\mathcal{E}[1 + 1]$ to $\mathcal{E}[2]$. We also allow program contexts to denote a hole in the middle of a sequence. For example, swapping two instructions that appear inside a sequence is described as the transition from $\mathcal{E}[t_1; t_2]$ to $\mathcal{E}[t_2; t_1]$, to be interpreted as a transition from $\mathcal{E}'[\{T_0; t_1; t_2; T_3\}]$ to $\mathcal{E}'[\{T_0; t_2; t_1; T_3\}]$, where $\mathcal{E}'$ denotes the program context associated with the outer sequence that contains $t_1; t_2$. We will only explicitly mention the surrounding program context $\mathcal{E}$ for the first few transformations.

Evaluation contexts Some transformations operate on subexpressions that appear inside an instruction. For those, we may need to restrict the form of the program contexts in which the subexpression may appear, to avoid nontrivial control-flow arising from, e.g., a conditional. Recall from section 5.5 that an *evaluation context*, written $\hat{\mathcal{E}}$, denotes a program context whose holes (possibly just one) are in evaluation position. For example, $f(g(\square, 2),\ g(3,\ a + 4))$ is an evaluation context with a single hole written $\square$.

One key property is that the rewrite is correct for any evaluation context $\hat{\mathcal{E}}$:

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

The reciprocal rewrite holds only if $\hat{\mathcal{E}}[t]$ is well-typed in our typesystem.

The validity of this rewrite rule, and more generally the interest of evaluation contexts for transformations, crucially relies on the hypothesis that the input code typechecks against our typing rules. Indeed, the Subexpr

rule ensures that, if a function has multiple arguments, then the available resources are distributed across the arguments—only read-only resources can be distributed onto several arguments. For example, $f(g_1(), g_2(), g_3())$ is equivalent to **let** $x = g_2()$; $f(g_1(), x, g_3())$ because, if the former term is well-typed, then the effects of $g_2()$ do commute with the effects of $g_1()$ and $g_3()$.

Notation for introducing ghost calls Recall that a call to a ghost function is an instruction that semantically behaves as a no-op, yet updates the context available. In the output of transformations, we write **ghost**($\Gamma \longrightarrow \Gamma'$) to mean the insertion of an appropriate ghost call $g()$, such that g admits Γ as precondition, and that the call to g adds the resources in Γ' to the context[1]. Concretely, the effect of **ghost**($\Gamma \longrightarrow \Gamma'$) is to consume the resources Γ then to produce the resources Γ'.

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

1: Technically, the ghost function g admits a context Γ'' as its postcondition and the call inserted by the transformation specifies a renaming map ρ such that $\Gamma' = \text{Rename}\{\rho\}(\Gamma'')$. Recall that these renaming maps are used by the typing rule App described in section 4.7.

6.1 Transformations on sequences of instructions

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

$$\boxed{\mathcal{E}[T_1;\ \Delta_1;\ T_2;\ \Delta_2]} \longmapsto \boxed{\mathcal{E}[T_2;\ T_1]} \qquad \text{correct if: } \begin{cases} \Delta_1.\text{alter} \cap \Delta_2 = \emptyset \\ \Delta_2.\text{alter} \cap \Delta_1 = \emptyset \end{cases}$$

The expression Δ_1.alter denotes the resources that T_1 adds or removes (consumes, produces, or ensures). It excludes resources that remained unaltered (carving or merging a fraction does not count as an alteration). The property $\Delta_1.\text{alter} \cap \Delta_2 = \emptyset$ captures the idea that if a resource is altered by T_1, then T_2 must not use it (this includes "write after read" dependencies), otherwise swapping T_1 and T_2 might not be correct. (The resource intersection operator $\cap$ was defined in section 5.2.) The second property, namely $\Delta_2.\text{alter} \cap \Delta_1 = \emptyset$, captures the symmetrical property: if a resource is altered by T_2, then T_1 must not use it (this includes "read after write" dependencies). When both conditions are met, the only resources that both T_1 and T_2 depend on are accessed in read-only mode, and T_1 and T_2 may be safely swapped without impacting their evaluation result.

Recall that such correctness criterion needs not to be checked in specification preservation mode that can be used in presence of full specifications.

Also note that the current version of Optiλ does not feature primitives performing external side effects such as input/output system calls. In order to guarantee that such side effects are preserved by transformations, those

side effects will need to be captured by resources and therefore appear in the usage map.

Annotations on the left represent resource information read by the transformation (here the usage maps Δ_1 and Δ_2, in green), and annotations on the right represent resource annotations written by the transformation (here no information is produced, it would be colored in orange).

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

Intuitively, the deletion operation preserves program semantics if the resources altered by T are not observed by the rest of the program or required by the postcondition. This intuition holds because our typechecker ensures that the deleted sequence of instructions always terminates. More precisely, if T has been typechecked as $\Gamma\ T; \Delta$, then we start with the resources Γ corresponding to not executing T, then forget the contents of the resources that might be different when not executing T. The resources to "uninitialize" Γ_m are computed by the filtering operation $\Gamma\restriction\cdot\Delta$.alter. (Filtering was defined in section 5.2.) Finally, we typecheck the auxiliary program $\mathcal{E}[G]$, in which the T is replaced with a ghost instruction G casting the Γ_m resources into their corresponding "uninitialized form", as performed by the IntoUninit operator. If a resource H is consumed by T, then G consumes H and produces $\mathsf{Uninit}(H)$.

The transformation can therefore be formalized as follows:

$$\boxed{\mathcal{E}[\Gamma\ T; \Delta]} \longmapsto \boxed{\mathcal{E}[\emptyset]}$$

correct if
$\mathcal{E}[\mathbf{ghost}(\Gamma_m \longrightarrow \mathsf{IntoUninit}(\Gamma_m))]$
typechecks, where $\Gamma_m = \Gamma\restriction\cdot\Delta$.alter

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

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

$$\boxed{\mathcal{E}[\emptyset]} \longmapsto \boxed{\mathcal{E}[T]}$$

correct if:

1. the program $\mathcal{E}[T]$ typechecks as $\mathcal{E}[\Gamma\ T; \Delta]$ for some Γ and Δ
2. the program $\mathcal{E}[\mathbf{ghost}(\Gamma_m \longrightarrow \mathsf{IntoUninit}(\Gamma_m))]$ typechecks, where $\Gamma_m = \Gamma\restriction\cdot\Delta$.alter, for the above values of Γ and Δ

Thereafter, for brevity, we omit the program context surrounding the code snippets, previously written $\mathcal{E}$.

Idempotent terms A number of transformations depend on the notion of idempotence. In the C23 standard, an expression is said to be "idempotent" if, intuitively, evaluating this expression multiple times in immediate sequence produces the same results. In OptiTrust, we leverage our resource analysis to capture a practical sufficient condition for idempotence.[2] A term can be considered idempotent if all the resources that this term produces correspond either:

- to uninitialized resources that were consumed by this term; or
- to read-only resources that the term consumes and returns with the exact same fraction.

These criterions may be formalized as follows:

A term T that appears in a program $\mathcal{E}[\Gamma_1\ T;\ \Delta\ \Gamma_2]$ is *idempotent* iff:

$$\begin{cases} \Delta.\mathsf{full} = \emptyset \\ (\Gamma_2 !\cdot \Delta.\mathsf{produced}) \boxminus (\Gamma_1 !\cdot \Delta.\mathsf{uninit}) = (\sigma, \emptyset) \\ \qquad\qquad \text{for some } \sigma \\ \Gamma_1 !\cdot \Delta.\mathsf{reads} = \Gamma_2 !\cdot \Delta.\mathsf{reads} \end{cases}$$

In particular, these criterions rule out terms that consume full resources, or produce resources they did not consume. To understand this definition, recall that a heap predicate H from the context can be downgraded on-the-fly into $\mathsf{Uninit}(H)$ when computing context subtractions. Therefore, a term that only consumes $\mathsf{Uninit}(H)$, and produces H is considered idempotent according to our definition, and can indeed be executed twice in a row without changing the result. To give more concrete examples, `x = y+1`, which reads `y` and assigns `x` is idempotent; however `x++`, which modifies `x`, is *not* idempotent. One key property that holds for an idempotent term t is that the following program equivalence holds:

$$\boxed{\begin{array}{l}\textbf{let } x = t; \\ \textbf{let } y = t; \\ \mathcal{E}[x, ..., y, ...]\end{array}} \leftrightarrow \boxed{\begin{array}{l}\textbf{let } x = t; \\ \mathcal{E}[x, ..., x, ...]\end{array}}$$

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

$$\boxed{T;} \leftrightarrow \boxed{\begin{array}{l}T; \\ T;\end{array}} \qquad \text{correct if } T \text{ is idempotent}$$

Similarly, if a term t is idempotent, then after the instruction **let** $x = t$, an instruction **let** $y = t$ may be inserted or removed, for a fresh variable y. The corresponding transformations are named `Instr.dup_let` and `Instr.dedup_let`.

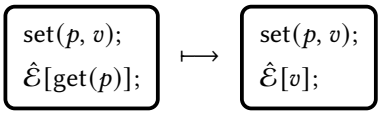
$$\boxed{\textbf{let } x = t;} \leftrightarrow \boxed{\begin{array}{l}\textbf{let } x = t; \\ \textbf{let } y = t;\end{array}} \qquad \text{correct if } t \text{ is idempotent and } y \text{ fresh}$$

2: The C23 standard defines a number of related notions. In particular, an expression is said to be "effectless" iff "any store operation that is sequenced during the execution is the modification of an object that synchronizes with the call". An expression is said to be "reproducible" iff it is both effectless and idempotent. Reproducibility is essentially equivalent to the notion of pure expression in GCC's terminology [AG22]. Due to our resource typing discipline, all OptiTrust terms can be considered "effectless". Hence, in the context of OptiTrust, "idempotent" and "reproducible" are equivalent.

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

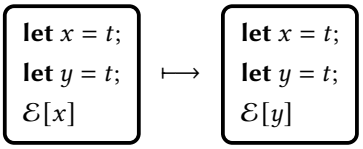
6.2 Transformations exploiting equalities

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



correct if $\hat{\mathcal{E}}$ is an evaluation context and v is a logical expression

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

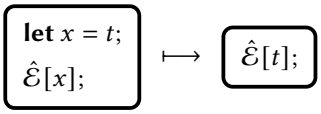


correct if $\mathcal{E}$ is a program context and t is idempotent.

6.3 Transformations on bindings

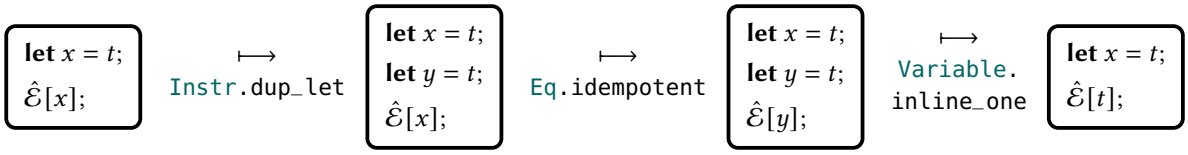
Inlining/binding for logical expressions The basic transformation `Variable.inline_pure` eliminates a binding of the form **let** $x = v$, where v is a logical expression, by replacing all occurrences of x with v. This transformation is always correct and requires no check. The reciprocal transformation,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

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



correct if the evaluation context $\hat{\mathcal{E}}$ contains no other occurrence of x than the one in its hole, and the output program typechecks

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



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

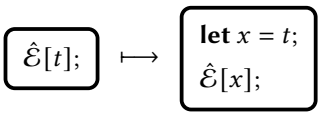
- If x has no occurrences, the effects of t need to be irrelevant to the rest of the program.
- If x has exactly one occurrence, then the effects of t needs to commute with all the instructions located between the binding on x and the occurrence of x.
- If x has several occurrences, then, in addition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

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

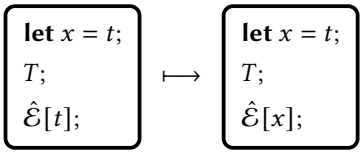
$$
\begin{array}{lll}
 & \textbf{let}\ x = t;\ g();\ \mathrm{set}(a, x);\ \mathrm{set}(b, x); & \\
\longmapsto & g();\ \textbf{let}\ x = t;\ \mathrm{set}(a, x);\ \mathrm{set}(b, x); & \texttt{(Instr.swap)} \\
\longmapsto & g();\ \textbf{let}\ x = t;\ \mathrm{set}(a, t);\ \mathrm{set}(b, x); & \texttt{(Variable.inline\_dup)}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

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

Binding introduction The basic transformation `Variable.bind_one` is essentially the reciprocal of `Variable.inline_one`.



Folding for additional occurrences The combined transformation `Variable.bind_dup` is essentially the reciprocal of `Variable.inline_dup`. (We implement it as a combination of `Variable.bind_one`, `Instr.swap`, `Eq.idempotent`, and `Instr.delete`.)



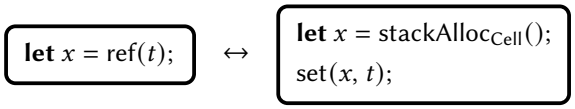
Common subexpression elimination The combined transformation `Variable.bind` is essentially the reciprocal of `Variable.inline`. Internally, it exploits the transformations `Variable.bind_one` and `Variable.bind_dup` to introduce a binding that factorizes the evaluation of common subexpressions. For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

6.4 Transformations on storage

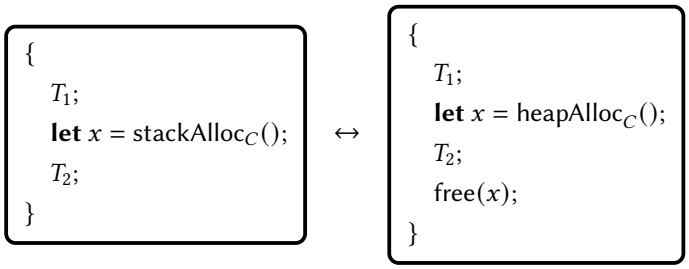
The purpose of this section is to present transformations for introducing, eliminating, and converting between various forms of storage. We present transformations operating on single cells, and omit from the discussion the generalizations to arrays and N-dimensional matrices.

Recall from chapter 3 that a pure program variable written `const int x = 3` in OptiC is represented in the manipulated Optiλ AST as **let** $x = 3$, that a non-pure stack-allocated variable `int x = 3` is represented as **let** $x =$ ref(3), and that an uninitialized variable `int x` is represented as **let** $x =$ stackAlloc$_{\text{Cell}}()$. For stack-allocated data, the resources produced by stackAlloc are automatically reclaimed at the end of the scope. For heap-allocated data, the resources produced by heapAlloc are consumed by the matching call to free.

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

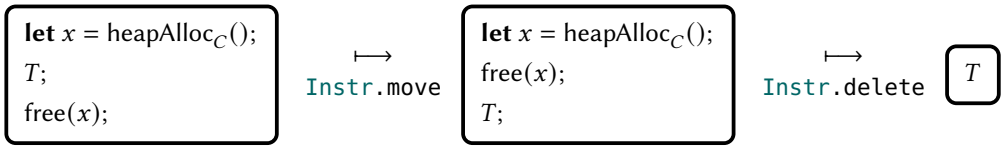


Converting between stack and heap allocation The basic transformation `Variable.to_heap` transforms an uninitialized stack-allocated storage into a corresponding heap-allocated storage. The transformation takes as optional argument the target at which the free instruction should be inserted; by default,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



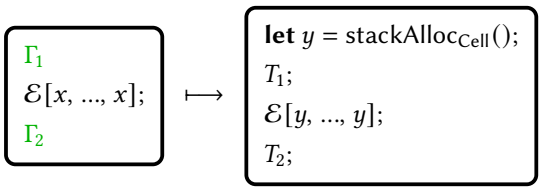
Removal of unused storage If a stack-allocated storage is never used, it may be removed by means of the operation `Instr.delete`. Concretely, the instruction **let** x = stackAlloc$_{\text{Cell}}()$ may be deleted if x has no occurrences, and the instruction **let** x = ref(e) may be deleted if moreover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

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



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

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



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

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

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

correct if the program to the right typechecks successfully, where H_x is:

- $x \rightsquigarrow \mathsf{Cell}$
- $\alpha(x \rightsquigarrow \mathsf{Cell})$
- or $x \rightsquigarrow \mathsf{UninitCell}$

depending on what appears in Γ_1

let y = stackAlloc$_{\mathsf{Cell}}()$;
T_1;
ghost$(\langle \emptyset \mid H_x \rangle \longrightarrow \langle H_g : \mathsf{HProp} \mid H_g, (H_g \mathrel{-\!\!*} H_x) \rangle)$;
$\mathcal{E}[y, ..., y]$;
ghost$(\langle \emptyset \mid H_g, (H_g \mathrel{-\!\!*} H_x) \rangle \longrightarrow \langle \emptyset \mid H_x \rangle)$;
T_2;

Figure 6.1: Description of the basic transformation `Variable.local_name`. In this figure, $\alpha < 1$ and all $x \rightsquigarrow \mathsf{Cell}$ can be replaced by $x \mapsto v$ for any v.

transformations, for example to store the transposed of a matrix in a cache friendly way (as in section 2.3), or to enable vectorization.

The transformation is described in figure 6.1. There, the group of instructions T is represented as $\mathcal{E}[x, ..., x]$, i.e. as a program context with multiple occurrences of x. The typing context Γ_1 describes the resources available before T, and Γ_2 the resources available after T. This typing information is used not only for checking the correctness criterion, but also for guiding the generation of the output code.

The correctness criterion appears at the bottom of figure 6.1. An essential aspect of this criterion is to check that, during the execution of T, the resource H_x corresponding to the full permission on x is "frozen" (i.e. made unavailable) in order to ensure that no operation may be performed on x via potential aliases of this pointer. The first ghost call uses a standard technique for enforcing such a "freeze" in separation logic: introducing a magic wand operator ($\mathrel{-\!\!*}$), guarded by a token named H_g in the rest of the sequence. The heap predicate H_g admits the type HProp, which is the type of all heap predicates in separation logic. This heap predicate H_g serves the role of a *key* for unfreezing H_x at the desired point—here, the end of the scope on which y is used in place of x, where the second ghost call is placed.

6.5 Transformations on loops

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

for $(i \in R_i)_{\chi_i}$ {
  T;
}
$\longmapsto$

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

where:

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

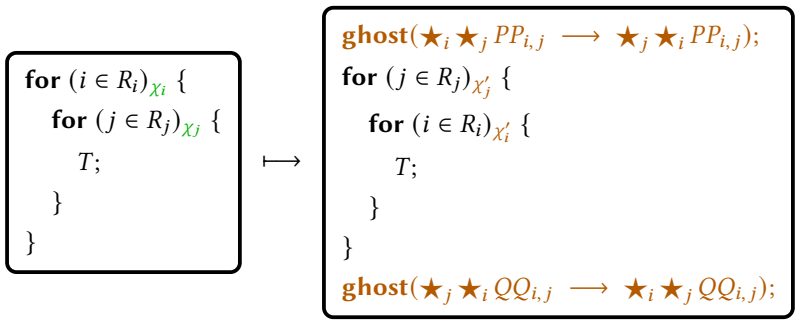
with the following possible instantiations for the ranges:

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

Figure 6.2: Description of the 4 variants of the basic transformation `Loop.tile`.

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

The 4 variants supported by `Loop.tile` are described in figure 6.2. The ranges of the three loops are written R_i, R_j and R_k, respectively. Recall that a range is of the form **range**$(start, stop, step)$. The notation $start..stop$ is a shorthand for **range**$(start, stop, 1)$. In particular, $0..n$ describes the range of values from 0 inclusive to n exclusive. The contracts for the three loops involved are written χ_i, χ_j and χ_k, respectively. To typecheck the output code, *ghost tiling* operations need to be inserted, as materialized before and



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

The contracts from the input code are decomposed as follows:

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

The contracts for the output code are built as follows:

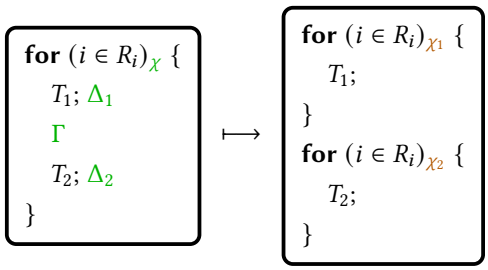
$$\chi'_j = \begin{cases} \text{vars} = \chi_i.\text{vars},\ \chi_j.\text{vars} \\ \text{shrd} = \{\text{inv} = (\bigstar_i IP_{i,j} \star IR),\ \text{reads} = (\bigstar_i RP_i \star RR)\} \\ \text{excl} = \{\text{pre} = (\bigstar_i PP_{i,j} \star PR_j),\ \text{post} = (\bigstar_i QQ_{i,j} \star PR_j)\}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

Figure 6.3: The basic transformation `Loop.swap`, in the particular case where the outer loop is parallelizable.

after the produced loops in the figure. Indeed, the loop on i consumes, in particular, the resource $\bigstar_{i \in R_i} \chi_i$.excl.pre whereas the loop on j consumes instead $\bigstar_{j \in R_j} \bigstar_{k \in R_k} \chi_k$.excl.pre.

Loop interchange The basic transformation `Loop.swap` allows interchanging (i.e. swapping) two loops. It is described at the top of figure 6.3. There exists a general criterion capturing when two loops may be swapped, however this criterion requires reasoning about the resources required by specific iterations, e.g. all pairs of iterations i, j and i', j' with $i' > i$ and $j > j'$. Instead, we focus on two conditions that are simpler yet sufficient for many practical situations: if at least one of the outer loop or the inner loop is parallelizable, then swapping the two loops is correct. Figure 6.3 describes the case where the outer loop is parallelizable. The case where the inner loop is parallelizable, not shown, is treated with just a few changes.

The first step is to partition the resources from the inner loop contract depending on where they come from relative to the resources from the outer loop. We name partitions by using the first letter to denote its inner loop origin, and the second letter to denote its outer loop origin. We use I for invariant, R for shared reads, P for exclusive precondition and Q for exclusive postcondition. For example, the inner shared reads are partitioned into RP_i that comes from the outer precondition, and RR that comes from the outer shared reads. Internally, we rely on the fact that our typechecker stores contract instantiations to compute partitions on χ_j.excl.pre and χ_j.shrd, and we rely on the subtraction operation $\boxminus$ to partition χ_j.excl.post.



correct if:

1. i is not free in χ.shrd
2. χ.shrd$|_i \cdot (\Delta_1.\text{alter} \cap \Delta_2) = \emptyset$
3. χ.shrd$|_i \cdot (\Delta_2.\text{alter} \cap \Delta_1) = \emptyset$
4. the output program typechecks

with:

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

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

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

Figure 6.4: The basic transformation `Loop.fission`.

Then, we appropriately place the resources obtained from the partitioning in the contracts χ_i' and χ_j' associated with the swapped loops. Compared with χ_j, the new contract χ_j' essentially adds a $\bigstar_i$ operator to certain components. Compared with χ_i, the new contract χ_i' removes occurrences of the $\bigstar_j$ operators. Note that the loop on i remains parallelizable. Around the new loop nest, a pair of ghost operations is inserted for swapping groups of resources—a necessary step to match the resources required by the new loop nest.

Loop fission The transformation `Loop.fission`, in its *basic* version, breaks a loop with body T_1; T_2 into two loops over the same range, a first loop with body T_1, and a second loop with body T_2. The transformation is described in figure 6.4. As for loop swapping, there exists a general correctness criterion expressed using inequalities on indices, but for now we focus on a simpler yet practical criterion.

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

There remains to explain how to synthesize the contracts χ_1 and χ_2, associated with the two generated loops, from the original contract χ. For shrd resources, we project the subsets of χ.shrd resources used by T_1 and T_2. For excl resources, we need to synthesize the resources at the cut point, written F_{cut}. The first loop takes the exclusive resources from χ.excl.pre to F_{cut}, whereas the second loop takes the exclusive resources from F_{cut} to χ.excl.post. At a high level, F_{cut} is computed by subtracting the shared resources as well as the local allocations from T_1, described by χ.shrd and StackAllocCells(T_1), from the typing context Γ computed by our typechecker at the location just between T_1 and T_2.

$$\boxed{\begin{array}{l}\textbf{for } (i \in R_i)_{\chi} \{ \\ \quad \Gamma_1\ T_1;\ \Delta_1 \\ \quad \Gamma_2\ T_2;\ \Delta_2 \\ \}\end{array}} \longmapsto \boxed{\begin{array}{l} T_1; \\ \textbf{for } (i \in R_i)_{\chi'} \{ \\ \quad T_2; \\ \}\end{array}}$$

with:

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

correct if:

1. i does not occur in T_1
2. T_1 is idempotent
3. $\Delta_1 \frown \Delta_2.\text{alter} = \emptyset$
4. R_i is nonempty, or the program on the right typechecks successfully with $F = \Gamma_2 \cdot \Delta_1.\text{produced}$

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

Figure 6.5: The basic transformation `Loop.move_out`.

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

Loop invariant code motion The basic transformation `Loop.move_out` applies to a loop with body $T_1; T_2$, where T_1 performs instructions that are redundant at every iteration. It produces as output code that first executes T_1, exactly once, then executes a loop with body T_2. The transformation is formalized in figure 6.5. We assume for simplicity the loop range to be provably nonempty, or T_1 to be provably deletable. Alternatively, T_1 could be wrapped into a conditional.

The key properties to check are that T_1 is the same for all iterations (it does not depend on i), can be safely deduplicated (it is idempotent as required by `Instr.dedup`), and does not interfere with the remaining instructions of the loop, described by T_2 (that is, the condition $\Delta_1 \frown \Delta_2.\text{alter} = \emptyset$). Note that, contrarily to the `Instr.move` criterion, it is safe for T_2 to read resources modified by T_1.

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

- `Loop.fusion` (reciprocal of `Loop.fission`): fuse two consecutive loops into a single one.
- `Loop.collapse` (reciprocal of `Loop.tile`): collapse two nested loops into a single one.
- `Loop.hoist_alloc`: hoist a variable allocated inside a loop into an array allocated outside the loop; more generally, it hoists a matrix of dimension N allocated inside a loop into a matrix of dimension $N + 1$ allocated outside the loop.

- `Loop.shift_range`: reindex a loop by applying a positive or negative offset to its values.
- `Loop.scale_range`: reindex a loop using an index that takes either smaller or larger steps.
- `Loop.extend_range`: extend the range of a loop by wrapping its body in a conditional.
- `Loop.unroll`: unroll a loop whose range is statically known.
- `Loop.parallel`: set (or unset) a parallel flag on a loop using our parallelizable criterion.

6.6 Transformations on annotations

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

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

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

This kind of issue with misplaced annotations is very common whenever transformations are chained together like in OptiTrust. Therefore, we developed a set of auxiliary transformation to deal with such cases, and hide those apparent issues for the user. In practice, for example, the combined transformation version of `Loop.swap` succeeds on the previous example.

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

```
__ASSERT(tile_div_check_i, "1024 == 64 * 16");
for(int j = 0; j < 64; ++j) {
  __xconsumes("for i in 0..1024 ⟶ &A[j, i] ⇝ UninitCell");
  __xproduces("for bi in 0..64 ⟶ for i in 0..16 ⟶ &A[j, bi * 16
    + i] ⇝ UninitCell");
```

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

Then, combined `Loop.swap` use a dedicated basic transformation to transform the loops over ghost instructions into ghost instruction themselves. Finally, the combined `Loop.swap` applies the basic `Loop.swap`, which produces the desired code.

The rest of this section first discusses the correctness criterion for annotation-only transformations and then shows a few examples of those annotation-only transformations.

Correctness criterion of annotation-only transformations The semantics of a program is fully determined by its *proper* Optiλ code: it does not depend in any way on the *ghost* code nor on the function and loop contracts. Therefore, loop contracts may be freely modified, and ghost instructions may be freely inserted, deleted, or modified. The requirement is to reach, after one or several updates, a set of annotations for which the typechecking of the code with updated annotations succeeds.

Note that, contrary to loop contracts, all the currently implemented annotation-only transformations preserve the top-level function contracts. This is due to the fact that changing the contract of a top-level function is generally unsound. In specification preservation mode, the correctness crucially depends on those top-level contracts being preserved, and in semantic preservation mode, changing e.g. the precondition can reduce the set of acceptable function inputs checked by subsequent semantic-preserving transformations.

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

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

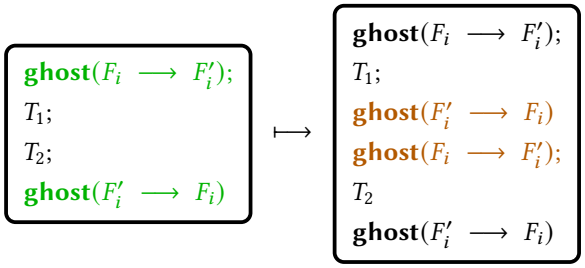
$$\boxed{\textbf{for } (i \in R_i)_{\chi} \ \{T;\ \Delta\}} \longmapsto \boxed{\textbf{for } (i \in R_i)_{\chi'} \ \{T\}}$$

Intuitively, the contract χ' is obtained by filtering out and by weakening resources from χ, depending on their usage in Δ. First, if a resource is unused by T and thus is absent from Δ or has usage joinedFrac, then it is excluded from χ'. As a result, certain variables that were quantified in χ might no longer have occurrence in χ', hence they can be removed as well. Second, if a resource appears with fraction 1 in χ, yet this resource is marked as splittedFrac is Δ, then this resource is replaced with a read-only version of it. Technically, an additional fraction variable must be quantified in χ', and this fraction variable is used for describing the resource as read-only. Internally, the implementation of contract minimization reuses our *minimization of triple* procedure (section 5.4 and appendix F). Details may be found in appendix H.

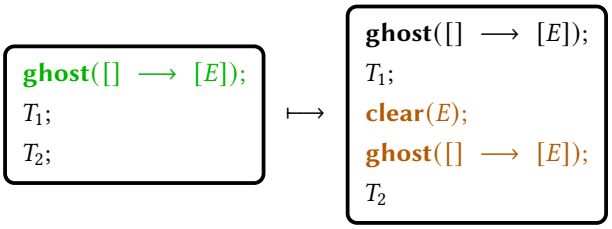
Moving and cancelling ghost instructions OptiTrust includes a transformation that attempts to remove pairs of ghost instructions that cancel each other. Indeed, the sequence $\textbf{ghost}(H \longrightarrow H')$; $\textbf{ghost}(H' \longrightarrow H)$ is equivalent to a no-op. More generally, the user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

Splitting ghost scopes Quite often, ghost instructions are needed to locally change the view on one resource between two program points in a sequence. For instance, if we have a permission to read an entire array, we may want to temporarily extract the permission to read a specific cell, and recompose a permission over the full array later. This pattern is so frequent that we decided to create in OptiC the syntax `__ghost_begin` and `__ghost_end` that we saw in chapter 2 to simplify writing the ghost pairs that change the view over one resource before an instruction in a sequence, and restore the initial resources later in the same sequence. We call *ghost scope* such code region delimited by pairs of cancelling ghost instructions.

In order to avoid issues with missing resources when manipulating instructions in a sequence with ghost scopes, it may be necessary to split a ghost scope into two independent ghost scopes. For example, when moving an instruction that is inside one or several ghost scopes, one can split the ghost scopes before and after the instruction and then move the instruction with its tightly surrounding ghost scopes.



We extended this strategy to avoid ghosts blocking transformations in even more cases. Indeed, ghost instructions producing only pure resources can be treated as if they were the beginning of a ghost scope ending at the end of the surrounding sequence. Then, to split such virtual ghost scope, our ghost scope splitting transformation inserts a special **clear** ghost instruction to forget the pure resource before the split point to avoid name conflicts and unwanted dependencies.



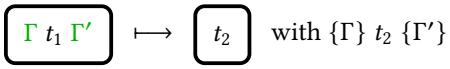
Loop over a ghost instruction into a ghost instruction We saw in the introductory example of this section that when manipulating ghost instructions using transformations such as `Loop.fission` to handle ghost code, these transformations sometimes create loops that only contain ghost code. Since such loops contain no executable code, we usually want to eliminate them, to avoid any possible runtime cost.

To do so, we developed a transformation that transforms a loop of ghost instruction into a ghost instruction itself. In practice, we remember inside the new ghost instruction the original contract and body of the loop to allow typechecking such new ghost with the typing rule for regular **for** loop.

6.7 Correctness of transformations

As we said in the introduction of this chapter, in OptiTrust, transformations have two modes of operation with a specific model of correctness: specification preservation and semantic preservation. This section describes more formally the model of correctness of both those modes.

Specification preservation Specification preservation is a mode that makes sense only with full specifications. In this mode, transformations guarantee that their output still respects the same specification as their input. More formally, every transformation in the specification preservation mode can be seen as an instance of the following generic transformation that allows transforming any typechecked term into another term with the same triple:



with {Γ} t_2 {Γ′}

With this model, simply running the typechecker on the output program to get $\{\!\{\Gamma\}\!\}\ t_2\ \{\!\{\Gamma''\}\!\}$ and checking that $\Gamma'' \Rightarrow \Gamma'$ is enough to justify the specification preservation of the transformation on a specific example. Therefore, we do not need to trust the transformation implementation because we can simply validate the transformed code (t_2) using our typechecker. Actually, with this model, we can see the full chain of user-guided transformations as one big transformation and simply validate the final code.

In practice, the majority of transformations are applied inside a function body whose specification does not change. In this case, we do not even need to check the final entailment $\Gamma'' \Rightarrow \Gamma'$ to prove specification preservation. Indeed, we can derive the following generic transformation that turns any function into a function with the same specification but a with different implementation:

$$\boxed{\mathbf{fun}(x_1, ..., x_n)_\gamma \mapsto t_1} \longmapsto \boxed{\mathbf{fun}(x_1, ..., x_n)_\gamma \mapsto t_2}$$

correct if output program typechecks

The fact that γ is the same in both sides, forces the typechecker to find the exact same context after the code on the left and the code on the right.

Note that, in specification preservation mode, the generic transformations presented above can also be manually applied by the user, and allows arbitrary rewriting that preserve the top level specifications. These generic transformations are specific to the specification preservation mode since they are not semantic-preserving.

Semantic preservation Semantic preservation is a mode in which transformations need to guarantee that they preserve the *observable behaviors* of the initial code in the transformed code. This mode is useful because it can be used even in presence of incomplete specifications. The rest of this paragraph defines formally the notion of observable behaviors in OptiTrust.

In general, compilation proofs show that behaviors of the output are all included in the set of behaviors of the input. In the case of OptiTrust, since we want to exploit incomplete specification information in our transformation, this is not fully satisfying for two reasons:

- First, and most importantly, we are not interested in behaviors that are never observed because they are ruled out by a precondition in the incomplete specification of the input code. Said differently, we want to preserve the semantics knowing that the precondition holds. Restricting ourselves to cases where the precondition holds is crucial to allow exploiting the incomplete specifications in the transformations. This restriction to valid inputs is standard in semantic preservation proofs using a refinement approach (such as Simuliris [Gäh+22]).
- Second, it is not important to preserve the content of cells that are described as uninitialized resources according to the postcondition of the transformed code. Indeed, by definition, this content is irrelevant and cannot be inspected in the rest of the code according to the incomplete specification.

[Gäh+22]: Gäher et al. (2022), *Simuliris: a separation logic framework for verifying concurrent program optimizations*

To tackle the second issue, we say that observable output states of the target code must correspond to an equivalent output state of the initial code. Equivalent output states are characterized with an operator EquivStates. Suppose $Q \subseteq \left\{(\sigma, \mu)_{|\text{prog}} \;\middle|\; (\sigma, \mu) \Subset \Gamma\right\}$, we define EquivStates($Q$, Γ) as a superset of Q, that characterizes states equivalent to another state in Q with

respect to an output context Γ. States are equivalent if they are the same up to the values of the cells covered by uninitialized permissions in Γ. Formally, we define OnlyUninit and OnlyInit to separate uninitialized and initialized resources with the two following definitions:

Definition 6.7.1: Uninitialized part of a context

$$\begin{aligned}\mathsf{OnlyUninit}(\Gamma) &= \mathsf{OnlyUninit}(\Gamma.\mathsf{linear})\\ \mathsf{OnlyUninit}((y : H) \star F) &= \begin{cases} y : H \star \mathsf{OnlyUninit}(F) & \text{if } H = \mathsf{Uninit}(H)\\ \mathsf{OnlyUninit}(F) & \text{otherwise}\end{cases}\\ \mathsf{OnlyUninit}(\varnothing) &= \varnothing\end{aligned}$$

Definition 6.7.2: Initialized part of a context

$$\begin{aligned}\mathsf{OnlyInit}(\Gamma) &= \langle \Gamma.\mathsf{pure} \mid \mathsf{OnlyInit}(\Gamma.\mathsf{linear})\rangle\\ \mathsf{OnlyInit}((y : H) \star F) &= \begin{cases} y : H \star \mathsf{OnlyInit}(F) & \text{if } H \neq \mathsf{Uninit}(H)\\ \mathsf{OnlyInit}(F) & \text{otherwise}\end{cases}\\ \mathsf{OnlyInit}(\varnothing) &= \varnothing\end{aligned}$$

Then, we can define EquivStates in the following way[3]:

Definition 6.7.3: Equivalent output states with respect to a context

$\mathsf{EquivStates}(Q, \Gamma)$ is defined as:

$$\left\{ (\sigma, \mu^I \uplus \mu_2^U)_{|\mathsf{prog}} \;\middle|\; \exists \mu_1^U, \begin{array}{ll} & (\sigma, \mu^I \uplus \mu_1^U)_{|\mathsf{prog}} \in Q\\ \wedge & (\sigma, \mu^I) \in \mathsf{OnlyInit}(\Gamma)\\ \wedge & \mathsf{Subst}\{\sigma\}(\mathsf{OnlyUninit}(\Gamma)) \models \mu_1^U\\ \wedge & \mathsf{Subst}\{\sigma\}(\mathsf{OnlyUninit}(\Gamma)) \models \mu_2^U \end{array} \right\}$$

Now, let us define a form of behavior inclusion that solves our three aforementioned issues, by considering a restriction to the behaviors that can be observed knowing a given precondition and postcondition. Consider two terms t_1 and t_2 satisfying the same triple with precondition Γ and postcondition Γ'. We write $\{\Gamma\}\ t_1 \supseteq t_2\ \{\Gamma'\}$ to capture the fact that behaviors of t_2 are included in the behaviors of t_1, starting from any state satisfying the precondition Γ, and allowing mismatch on the parts of the output states that correspond to uninitialized permissions in Γ'.

Definition 6.7.4: Behaviour inclusion under context

$\{\Gamma\}\ t_1 \supseteq t_2\ \{\Gamma'\}$ is defined as:

$$\begin{array}{ll} & \{\Gamma\}\ t_1\ \{\Gamma'\} \quad \wedge \quad \{\Gamma\}\ t_2\ \{\Gamma'\}\\ \wedge & \left(\begin{array}{l} \forall (\sigma, \mu) \overline{\in} \Gamma, \forall Q \subseteq \mathsf{AcceptableStates}(\sigma, \mu, \Gamma'),\\ \quad t_1/(\sigma, \mu)_{|\mathsf{prog}} \Downarrow Q \implies t_2/(\sigma, \mu)_{|\mathsf{prog}} \Downarrow \mathsf{EquivStates}(Q, \Gamma') \end{array} \right) \end{array}$$

With this definition, we can say that a transformation is semantic-preserving when it is an instance of the following generic transformation:

$$\boxed{\Gamma\ t_1\ \Gamma'} \longmapsto \boxed{t_2} \quad \text{with } \{\Gamma\}\ t_1 \supseteq t_2\ \{\Gamma'\}$$

In general, proving that a transformation is semantic-preserving is complex, as it relies on the correctness criterion checked by the transformation

3: Depending on the choice of memory model, it may or may not be necessary to also consider equivalent two states that are the same up to some kind of bijection on memory locations. Note that, in any case, the semantics of OptiTrust described in appendix A makes non-deterministic choices for allocated locations. It appears that, the use of the omni-big-step semantics removes the need for introducing a bijection over locations when paired with a block memory model such as the one in CompCert [Ler+12].

implementation. We leave for future work a formal proof that the transformations presented in this chapter, are indeed semantic-preserving when their correctness criterions are satisfied.

That said, we can handle here the simple case of annotation-only transformations. Indeed, since the semantics of a program does not depend on the resource annotation, two programs differing only by their annotations have equivalent behaviors in all contexts. If we write $t_1 =_{\text{code}} t_2$ the fact that t_1 and t_2 have the same code when forgetting all annotations, this leads to the following theorem:

Theorem 6.7.5: Behaviour inclusion for programs with the same code

$$t_1 =_{\text{code}} t_2 \quad \wedge \quad \{\Gamma\}\ t_1\ \{\Gamma'\} \quad \wedge \quad \{\Gamma\}\ t_2\ \{\Gamma'\} \quad \Longrightarrow \quad \{\Gamma\}\ t_1 \sqsupseteq t_2\ \{\Gamma'\}$$

7 Perspectives

In this PhD, we described the ingredients needed to create an interactive compiler with trustworthy source-to-source transformations. These transformations exploit invariants expressed in separation logic to guarantee their correctness. Moreover, these transformations preserve contract and ghost annotations in the code that can be checked by our resource typechecker. Crucially, this typechecker can deduce the invariants needed for the next source-to-source transformation, therefore enabling the user to execute chains of user-guided transformations. We focused on developing OptiTrust to support three real world case studies mentioned in chapter 2. Naturally, there are still many language features, logic features and transformations one could add in the future to extend the domain of applicability of our interactive compiler.

Another direction of improvement for OptiTrust, is to increase the trustworthiness of the tool further by reducing the trusted code base (TCB). As we have showed, we already support different levels of transformation trustworthiness. On the one hand, if the initial program is annotated with incomplete specifications, the implementations of the transformations are responsible for checking their own correctness criterion and therefore are in the TCB. On the other hand, if the initial program is annotated with full specifications, the final source code alone can be checked against this specification, and the implementations of the transformations are not in the TCB. In both cases, the implementation of the OptiTrust typechecker is in the TCB. Moreover, in order to actually compile the output, a currently unverified extraction mechanism to C code, and the C compiler that is then used to compile the code are also included in the TCB.

This chapter gives perspectives of extensions that can be developed after this PhD work to overcome the current limitations of OptiTrust and ideas on how we could reduce further the trusted code base. These perspectives are sorted thematically and not by relevance.

7.1 Language extensions

Recursive functions and while loops Most interactive compilers that we know of (TVM, Halide, Exo, Alpinist, ...) focus on programs without language constructions that may not terminate such as **while** loops or recursive functions. The current version of OptiTrust is not an exception in that regard, but we naturally expect to support **while** loops and recursive functions in the future.

Usual formally-verified compilers support language constructions that introduce non-termination. When trying to justify their correctness, those compilers require that terminating source programs are compiled to terminating target programs, and reciprocally that programs that do not terminate are compiled to programs that do not terminate. This approach is called *equitermination*. To reason about those non-terminating programs, non-termination can be characterized by co-induction [Ler06]. However, in general, non-terminating programs are not very useful in absence of observable side effects. To model those side effects in presence of possibly infinite executions, more complex semantics using co-inductive structures

[Ler06]: Leroy (2006), *Coinductive Big-Step Operational Semantics*

such as *interaction trees* [Xia+19] can be used, and allow proving properties about the trace of externally observable side effects. In any case, to support this equitermination approach in OptiTrust, all transformations will need to preserve the (non-)termination of programs. To reach this goal, the grammar of loop and function contracts will need extensions to handle these possibly non-terminating executions, and the typechecker and usage system will need to be updated accordingly.

[Xia+19]: Xia et al. (2019), *Interaction trees: representing recursive and impure programs in Coq*

Another solution to introduce **while** loops and recursive functions in Optiλ is to disallow programs that do not terminate by relying on techniques that check termination. One way to check such termination is to ask the user to add a *variant* annotation on every **while** loop and recursive function. A variant is an expression which evaluates to a natural number that must decrease between two iterations or recursive calls. Then, the OptiTrust typechecker could verify that such variant hold to guarantee that in any case all Optiλ terms that typecheck always terminate. This strategy is for instance used inside Why3 [FP13].

[FP13]: Filliâtre et al. (2013), *Why3—Where Programs Meet Provers*

Another path with weaker theoretical guarantees, but that may still be valuable in practice, is to ignore the termination issues and instead use a *partial correctness* model. With partial correctness, the compiler only guarantees that programs that do terminate are correct and do not tell anything about executions that loop indefinitely. Ignoring the non-terminating executions may look like cheating. Indeed, partial correctness allows a badly written transformation to transform a terminating program into a non-terminating one. However, remember that the execution time of an algorithm is generally not guaranteed by compilers, and that some terminating programs may take an absurdly long time to return, therefore having the same practical value as a non-terminating program. The partial correctness model makes sense in presence of full specifications: when executing the target code, if such code terminates then the output is guaranteed to respect the specification. On the other hand, it is very hard to find a meaning for partial correctness with transformations that should preserve the semantics of an initial possibly non-terminating program (i.e. what should the program compute if it terminates more often than the initial program?). Moreover, in practice, partial correctness is not so easy: allowing infinite loops in ghost code or specifications creates soundness issues, so there will still be a need for termination checks in the logic. Such partial correctness model is the default in the Iris separation logic framework, which uses a method called *step indexing* to avoid logical infinite loops while allowing infinite recursion in the program.

Overall, I think that partial correctness is a less promising path than variant annotations and equitermination reasoning, but all three approaches deserve further exploration.

Functional programming language features Currently, OptiTrust supports structures and arrays, as the only constructions to create complex data types. For someone with a functional programming background, this is very limiting. In particular, algebraic sum types and pattern matching are missing. Adding support for those language constructions is already the subject of ongoing extension work.

Adding sum types and pattern matching to Optiλ should not be difficult by itself. The interesting part lies in adapting the typechecking rules for the pattern matching and in implementing relevant transformations around algebraic data types and pattern matching (e.g. [Els24]).

[Els24]: Elsman (2024), *Double-Ended Bit-Stealing for Algebraic Data Types*

Note however that there is no natural syntax for such functional constructions in OptiC since sum types are not naturally supported in C and must use a rather complex low-level encoding using unions[1].

1: I believe that if OptiTrust goes in that direction, keeping the user facing language close to C is not ideal, and we should either detach OptiC from the C syntax or develop another more functional user-facing language in OptiTrust.

SIMD instructions For simplicity, OptiC currently do not yet have a proper support for SIMD instructions. Instead, parallel **for** loops can carry a flag that asks the extraction mechanism to place an OpenMP SIMD directive on its extracted C code which then asks the backend C compiler to emit SIMD code for the loop. This is not satisfactory for two reasons. First, the external C compiler is not guaranteed to interpret the OpenMP directive in the best way to produce the best code and manual tweaking of the selected SIMD instruction is not possible. Second, we are reusing the notion of parallel **for** loop in the typechecker to check that vectorization is possible. This check is too restrictive, and therefore OptiTrust might currently refuse to add the SIMD flag on loops where such flag does not change the semantics. In the future, OptiTrust should, like in Exo [Ika+22], use dedicated data types to represent SIMD registers and provide primitive instructions to manipulate those registers.

[Ika+22]: Ikarashi et al. (2022), *Exocompilation for productive programming of hardware accelerators*

Language support for accelerators As said in the introduction, high performance code may benefit from the use of specific hardware accelerators, and sometimes from the collaboration between accelerators and the CPU. In the long term, it should be possible to add support for programs executing specific functions or loops on an accelerator. Supporting code for those accelerators might require specific data types, language constructions or transformations. For example, tools such as Alpinist [Sak+22] include a GPU-specific transformation named *kernel-fusion* that requires a thread synchronization primitive to solve data dependencies issues.

[Sak+22]: Sakar et al. (2022), *Alpinist: An Annotation-Aware GPU Program Optimizer*

7.2 Program logic extensions

Support for user-defined representation predicates OptiTrust's linear resource grammar is currently very limited. Indeed, it is only possible to express properties about single cells, records (a.k.a. `struct` in OptiC), and matrix-like structures. Other separation logic frameworks usually give the possibility to their user to define custom representation predicates. Typically, in a framework such as CFML, a linked list heap predicate could be defined as:

$$
\begin{aligned}
p \rightsquigarrow \mathsf{List}(L) = \ & \textbf{match } L \textbf{ with} \\
& \mid \mathsf{nil} \rightarrow [p = \mathsf{null}] \\
& \mid x :: L' \rightarrow \exists q : \mathsf{loc},\ \ (p \boxdot \mathsf{head} \mapsto x) \star (p \boxdot \mathsf{tail} \mapsto q) \star (q \rightsquigarrow \mathsf{List}(L'))
\end{aligned}
$$

The challenge is to integrate those custom representation predicates with the rest of the OptiTrust framework without causing soundness issues, especially with read-only fractions. In particular, as we said in section 4.1, two fractions of a heap predicate using disjunction or existential quantifiers cannot be merged together in the general case. Many very common heap predicates such as the predicate for linked list shown above require some kind of existential quantification. Supporting those predicates is therefore an issue as the OptiTrust typesystem currently relies on automatic splitting and merging of read-only fractions anywhere during the typechecking process.

[JP08]: Jacobs et al. (2008), *The VeriFast Program Verifier*

[DMS22]: Dardinier et al. (2022), *Fractional resources in unbounded separation logic*

An interesting idea from the VeriFast framework [JP08], or from the unbounded separation logic [DMS22] is to syntactically check whether predicates are compatible with recombination of fractions, even in presence of existential quantification. Then, we could adjust the typechecker to disallow taking fractions of predicates that cannot be recombined later.

Specification of higher-order functions Technically, higher-order functions (that is functions taking other functions as arguments) are currently supported in Optiλ. However, in practice there is no mechanism to define specifications for higher order functions that depend on and constrain the specification of their function argument.

Theoretically, functions contracts can already contain the specification of functions, but we have no syntax for this in OptiC and currently those specifications must be fully syntactically equal during unification. In order to express and typecheck higher order function specifications, we should first add a mechanism to instantiate or unify an abstract predicate of type HProp with a resource set. Then, we could extend our typesystem to allow unification of function contracts with other function contracts containing a potentially different number of resources but at least one unresolved HProp. With such system we hope to typecheck code such as:

```
void repeat(int n, fun<void()> f) {
  __requires("H: int → Hprop");
  __requires("Spec(f(), { i: int | H(i) }, { | H(i+1) })");
  __consumes("H(0)");
  __produces("H(n)");
  for(int i = 0; i < n; ++i) {
    __spreserves("H(i)");
    f();
  }
}

void test(int n, int* p) {
  __writes("p ↦ n");
  *p = 0;
  repeat(n, [&]{
    __requires("i: int");
    __consumes("p ↦ i"),
    __produces("p ↦ i+1");
    *p += 1;
  });
}
```

External side effects Currently, there is no mechanism in OptiTrust for specifying external side effects such as printing in the console or reading a file. In order for OptiTrust to guarantee that such external side effects are preserved across transformations, we need to model those in the typesystem and include them in the usage maps. In the future, we could integrate such external effects by keeping resources that model those external interactions. For example a function that outputs to the console would take and return a linear resource modelling the standard output channel.

More advanced concurrency As said in section 1.3, fractions in concurrent separation logic can model resources shared across threads that are more complex than the currently supported read-only permissions. For instance, fractions can also be used to model permission to perform a certain class of atomic operation over a memory cell, such as concurrent atomic

fetch-and-add. With more implementation work, a future version of OptiTrust could easily support those other kinds of primitive atomic operations and the corresponding fraction modes.

A more complex, but also more powerful, possibility is to support data structures with atomic operations that are not supported directly by the CPU, but that are instead linearization properties of the data structure interface. For example, concurrent hash sets are not natively supported by CPUs. However, it is possible to implement a concurrent hash set data structure that is shared across several threads that can concurrently add elements inside the set [SB14].

[SB14]: Shun et al. (2014), *Phase-concurrent hash tables for determinism*

An interesting future work would be to find a practical solution in OptiTrust to encode those complex linearization properties and manipulate the underlying concurrent data structures using fractions. In order to implement complex concurrent data structures, one might need to use locks for creating synchronization points, and temporarily obtain exclusive access to otherwise shared data. A lot of work around locks and concurrent data structures in separation logic has been formalized inside the Iris framework (e.g. [SK24]). For integration inside OptiTrust however, I expect issues with the preservation of termination when manipulating such code using locks. An easier path could be to find a way to formally integrate those concurrent data structures proven in Iris as black boxes in OptiTrust.

[SK24]: Somers et al. (2024), *Verified Lock-Free Session Channels with Linking*

Properties beyond functional correctness In general, formal verification can be used to obtain guarantees beyond functional correctness. The question is whether OptiTrust can exploit or preserve these other kind of guarantees. Let us consider a few examples.

One property that may be crucial is the time complexity of an algorithm. Using separation logic, one can use *time credits* to count the number of basic operations performed between two program points. From this basic building block, it is possible to deduce worst-case or asymptotic bounds for time complexity [Gué19]. Similarly, *space credits* [Moi24] can be used to express the space complexity of an algorithm. In the same spirit as time and space complexity properties, previous works have been conducted to express security invariants that can for instance guarantee the absence of secret information leaks [EM19] (and more generally hyper-properties).

[Gué19]: Guéneau (2019), *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*

[Moi24]: Moine (2024), *Formal Verification of Heap Space Bounds under Garbage Collection*

[EM19]: Ernst et al. (2019), *SecCSL: Security Concurrent Separation Logic*

It is an open question to find whether it is possible to leverage time bound, space bounds, or hyper-properties to guide the choice of transformations. Independently, adapting OptiTrust to create transformations that preserve such properties beyond functional correctness is an interesting challenge. We may be able to take inspiration from CakeML, where there is already some support to preserve space guarantees across transformations [Góm+20].

[Góm+20]: Gómez-Londoño et al. (2020), *Do you have space for dessert? a verified space cost semantics for CakeML programs*

Proof automation OptiTrust currently requires the programmer to annotate the input program with ghost operations as well as function and loop contracts. One may wonder the extent to which such annotations could be automatically inferred, at least for reasonably simple programs.

The experience from other practical separation logic frameworks (like Viper [MSS17]) is that heuristics can be devised to significantly reduce the number of ghost operations that need to be explicitly provided by the programmer. For example, if we have at hand no other permissions on an array than a permission covering a range of its cells, then when facing read operations on a particular cell from this array, isolating this cell from the range at hand is the only way in which typechecking could succeed.

[MSS17]: Müller et al. (2017), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

We could also take inspiration from the goal directed proof search in RefinedC [Sam+21], to reduce the number of ghosts needed by adding more information in the function and loop annotations.

[Sam+21]: Sammler et al. (2021), *RefinedC: automating the foundational verification of C code with refined ownership types*

Software verification frameworks also usually leverage SMT solvers to automate proof search. Integrating one of those SMT solvers in OptiTrust should be possible, but some care should be taken to always identify the hypotheses from the context that are required to prove each proof goal to keep the ability to validate the final proof and compute usage maps on every term.

Inference is not limited to ghost operations: certain contracts may also be automatically inferred. For example, previous works [JM18] show that, by leveraging abstract interpretation, for functions such as matrix-multiplication or similar linear algebra operations, full functional correctness specifications can be automatically computed. Besides, *bi-abduction* [Spi+24] is a technique for inferring function contracts, at the heart of the *Infer* automated program analysis tool [Cal+19].

[JM18]: Journault et al. (2018), *Inferring functional properties of matrix manipulating programs by abstract interpretation*

[Spi+24]: Spies et al. (2024), *Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq*

[Cal+19]: Calcagno et al. (2019), *Go Huge or Go Home: POPL'19 Most Influential Paper Retrospective.*

7.3 Transformation extensions

Data layout transformations In this PhD, we did not consider transformation over the data layout of structures. Prior versions of OptiTrust with transformations that did not check any kind of correctness, support some transformations on the data layout. We could adapt those legacy transformations in the more trustworthy system presented in this manuscript.

A very common optimization in that data layout category transforms an array of structures to a structure of arrays, and can sometimes significantly improve cache locality. Another common optimization is bit-stealing that allows some struct fields to use unused bits from other fields of the same structure. A third interesting transformation is the insertion and removal of pointer indirection inside structures.

Loop with complex dependencies between iterations Recall from section 6.5 that we chose not to support fully general correctness criterion for loop transformations. In particular, currently, compilers based on the polyhedral model (such as for instance Tiramisu [Bag+19] or Clay [Bag+16]) accept more transformations than we do in OptiTrust, by finely modelling dependencies between iterations. One particular case managed by the polyhedral model where OptiTrust struggles is when the same array is read and written by different iterations of the same loop. Conversely, the polyhedral model is not capable of reasoning about pointer indirections, that are fully supported in OptiTrust. Supporting in OptiTrust the same kind of dependency between loop iterations as the polyhedral model would create the best of both worlds.

[Bag+19]: Baghdadi et al. (2019), *Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code*

[Bag+16]: Bagnères et al. (2016), *Opening Polyhedral Compiler's Black Box*

Such goal of supporting in OptiTrust all the loop transformation valid in the polyhedral model is currently hard to achieve, mostly because we do not yet have support for detecting and manipulating those complex dependencies between iterations. To get an intuition of why it is hard to detect those dependencies between iterations in the general case, take the following C code:

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    if (i > 0) {
      T[i][j] += T[i-1][j];
    }
```

```
    if (j > 0) {
      T[i][j] += T[i][j-1];
    }
  }
}
```

In this code, the computation of the cell `T[i][j]` depends on the cells `T[i-1][j]` and `T[i][j-1]`. A possible annotated OptiC version of this algorithm looks like:

```
for (int i = 0; i < n; ++i) {
  __smodifies("T ⇝ Matrix2(n,n)");
  for (int j = 0; j < n; ++j) {
    __smodifies("T ⇝ Matrix2(n,n)");
    __ghost_begin(split_i, group_split, "i");
    __ghost_begin(split_j, group_split, "j,
      fun j' → &T[MINDEX2(n,n,i,j')] ⇝ Cell");
    if (i > 0) {
      __ghost_begin(focus_im, group_ro_focus, "i-1, fun i' → for
    j' in 0..n → &T[MINDEX2(n,n,i',j')] ⇝ Cell");
      __ghost_begin(focus_j, group_ro_focus, "j, fun j' → &T[
    MINDEX2(n,n,i-1,j')] ⇝ Cell");
      T[MINDEX2(n,n,i,j)] += T[MINDEX2(n,n,i-1,j)];
      __ghost_end(focus_j);
      __ghost_end(focus_im);
    }
    if (j > 0) {
      __ghost_begin(focus_jm, group_ro_focus, "j-1, fun j' → &T[
    MINDEX(n,n,i,j')] ⇝ Cell");
      T[MINDEX2(n,n,i,j)] += T[MINDEX2(n,n,i,j-1)];
      __ghost_end(focus_jm);
    }
    __ghost_end(split_j);
    __ghost_end(split_i);
  }
}
```

Let us suppose we want to swap loops over i and j in this example. If we try to directly support this kind of dependencies between iteration, the `Loop.swap` transformation would need to inspect the loop body and analyze the `group_split` and `group_ro_focus` annotations to extract the iteration dependency pattern. Then after checking that it is applicable, the transformation would need to regenerate a similar-looking but slightly different set of annotations on its output code.

Doing this kind of analysis and annotation reconstruction on every loop transformation seems to require too much work. To extend loop transformations while avoiding this workload, I see two possible ways forward. On the one hand, we could search for a way to model those ghost operations in a more abstract representation that is easy to manipulate (like polyhedrons). On the other hand, we could instead extend the notion of loop contract to directly describe iteration dependency patterns.

Preservation of typechecking after arithmetic rewriting In the current version of OptiTrust, the arithmetic rewriting transformation breaks the typechecking in many cases even if it preserves the semantics. To understand why, let us take an example. Suppose that a function f has a contract that mentions a resource of the form $p \rightsquigarrow \mathsf{Matrix1}(n+m)$. Suppose that f is called in a context where p is allocated with size $5+1$. Finally, suppose that the user wants to simplify the expression $5+1$ that appears in the code into 6. In this case, the typechecker cannot succeed to typecheck the call to f

anymore as 6 is not an expression of the form $n + m$. This example shows that in order to preserve typechecking, arithmetic rewriting transformation should be extended to insert rewriting ghost instructions whenever those are needed.

Finding where the missing rewriting ghost instructions should be placed remains unclear to me at the moment. One simple but ugly solution is to always immediately cancel the arithmetic rewriting just after the execution of the rewritten expression. With such solution, proofs after a rewriting will never mention the (probably simpler) rewritten expression. The main issue with such solution is that subsequent transformations cannot leverage the new expression to justify their correctness or generate new proof annotations.

Arithmetic rewriting with finite representation of numbers In the current version of OptiTrust, arithmetic rewriting is only supported for idealized integer and reals. Such arithmetic rewritings are currently not supported on fixed size integers or floating point values. Regarding fixed size integers, reasoning about arithmetic requires proving the absence of overflows and therefore requires sufficiently precise range information. Regarding floating point, arithmetic rewritings are almost always false but most applications such as numerical simulation accept approximated answers. In order to justify that floating-point computations give an acceptable approximation, we should add support for reasoning about the precision of floating-point computations, as formalized in tools such as Flocq [BM11] and Gappa [DLM11]. All of these new specifications and transformation extensions could theoretically be supported in the future, but might require a lot of work.

[BM11]: Boldo et al. (2011), *Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq*

[DLM11]: Dinechin et al. (2011), *Certifying the Floating-Point Implementation of an Elementary Function Using Gappa*

One reasonable workflow to handle finite numerical values in OptiTrust transformation scripts could be to start by transforming the code using idealized numbers ignoring precision and overflows, and then at the end of the transformation script use a transformation that realizes those idealized computations with appropriate sizes and arithmetic expressions. We could, for example, take inspiration from Odyssey [Mis+23], to help the user interactively choose a precise-enough floating point expression that approximates an idealized computation over reals.

[Mis+23]: Misback et al. (2023), *Odyssey: An Interactive Workbench for Expert-Driven Floating-Point Expression Rewriting*

More automatic combined transformations Recall from case studies that our scripts are written using combined transformations, which can analyze the code and then call basic transformations. Currently, even our combined transformations give full control to the user who needs to precisely describe which transformation must be applied and where.

In the future, we could try to design heuristics that exploit our resource and usage annotations, in order to make even more automated transformations. Those heuristics can automate the transformation process in two different ways:

- Heuristics could be used to automatically determine where to apply a fixed transformation. For instance, we could create a transformation that leverage usage annotations to remove all the instructions that write to a location that is never observed later.
- Heuristics could be used to create transformation that apply at a given target position but try to automatically choose a good transformation candidate. For example, such automated transformations could try to find heuristically a good way to reorganize a loop nest.

Automating both the choice of the target position and the choice of the kind of transformation creates fully automatic transformations. Those fully automatic transformations can probably encode a lot of heuristically driven optimization patterns from traditional optimizing compilers.

The interest in importing those automatic transformations in a source-to-source interactive compiler such as OptiTrust, is that the user can still manipulate the code produced by the automatic transformation by applying any other (manual or automatic) transformation afterwards.

7.4 Reducing the trusted code base

Soundness of the algorithmic typing rule In section 4.8, we presented a formal description of the semantic meaning of our algorithmic typing rules. These rules are variants of rather standard separation logic typing rules, or can be derived from standard separation logic rules. However, to gain more confidence in this formalization, we should add a mechanized proof (e.g. validated in Rocq) of the soundness of the algorithmic typing rules.

Verify the implementation of the typechecker Proving the rules sound, however, is not enough to remove the implementation of our typechecker from the trusted code base. Indeed, we also would need to prove that the typechecker's output respects the formal algorithmic typing rules. Here, we have two options:

- Either we develop a mechanized proof that the typechecker implementation always produces a valid tree of algorithmic typing rules. This would be the most natural approach, but it may be highly technical because it would involve a deep embedding of the logic, and reasoning about unification inside this deep embedding.

- Or we can use a validation approach. The current output of the typechecker contains enough information to extract the corresponding tree of algorithmic typing rule as a proof certificate (e.g. a Rocq proof term). Such proof certificate that can be imported and typechecked by a proof assistant in which the soundness of typing rules is established. A priori, this option is more simple, but only provides guarantees about the typechecking of specific program instances and never proves that the typechecker is correct for all programs.

Formally verified backend With OptiTrust, a user can obtain OptiC code that is verified and optimized. In practice, some users would be interested instead in machine code that is verified and optimized. Currently, the path from OptiC to machine code relies on two unverified components: our unverified extraction from OptiC to C; and the use of an unverified C compiler. One more trustworthy path would be to use a formally verified extraction from our OptiC code into one of the intermediate languages of an existing formally verified compiler. For example, we could verify an extraction mechanism from OptiC to the Clight intermediate language of CompCert. Then, we can leverage the chosen existing formally verified compiler to create a verified toolchain from OptiC down to machine code.

Proving semantic-preserving transformations Another question we could ask ourselves is whether we can prove that transformations are correct. With full functional correctness specifications, transformations can be executed in specification preserving mode. In that case, proving transformations correct is unnecessary since the typechecking of their output alone is enough to validate that the final code respects the same semantics as the initial code. Said differently, when preserving full functional correctness specifications, the implementation of transformations is already absent from the trusted code base. With incomplete specification, we cannot simply rely on the soundness of our typechecker to validate transformations because those transformations must also preserve the semantics of the initial code. Therefore, we would need some extensions to remove the implementation of the transformations from the trusted code base.

One path towards a validation of the semantic preserving transformations could be to design a procedure that can encode the behaviors of the initial code as full functional correctness specifications. If this is feasible, we could transform all incomplete specifications into full specifications and then use the specification-preserving transformations everywhere.

Another path can be to formally verify the implementation of the semantic-preserving transformations. This solution seems to require a significant amount of work. Indeed, most transformations leverage information from the computed intermediate invariants and usage maps. Therefore, all this information computed by the typechecker and soundness theorems about the correctness of this information must be used in such transformation proof. We tried proving the `Instr.swap` transformation with incomplete specification is semantic-preserving on paper. However, such proof was too long to make for a too hard to read output, and therefore I decided to abandon the proof before completing it. A fresh look at the subject might find shortcuts that make such approach feasible in a reasonable time.

7.5 Framework engineering

Improving user interaction loop OptiTrust is an interactive tool and as such, user interaction is very important. The current version already has a non-negligible amount of tooling to visualize a transformation trace tree, and code diff between each step. That said, this interactivity can be improved in a lot of ways. To name only a few examples, I can think of computing diffs that follow better the transformation intent, provide a mechanism to automatically generate a target from the cursor position, or display information computed by the typechecker on any node by clicking on that node. Actual user feedback from high-performance code experts could also help to make a more ergonomic tool.

Scaling issues On the case studies presented in this manuscript, OptiTrust takes a few minutes to fully execute the transformation scripts from top to bottom. This is mainly due to the fact that currently we run the typechecker on the full program after each basic transformation. This is not a theoretical limitation of our approach. Indeed, both the typesystem and the transformations are fully modular (i.e. they do not require inlining function calls to execute).

In the near future, we should make an *incremental* version of the typechecker (like e.g. in [BDG19]) that is able to skip the typechecking of unmodified code

[BDG19]: Busi et al. (2019), *Using Standard Typing Algorithms Incrementally*

and annotations by reusing contexts that were previously computed. Moreover, most of the time, a transformation only exploits resource information until a given point in the code. Therefore, instead of always typechecking the full code between every transformation, OptiTrust could only typecheck the smallest prefix of the code needed for the next transformation with an *on-demand* interface. Combining on demand and incremental analysis is a known technique for reducing analysis costs in automatic verification tools [SCS24]. We can also speed up contract instantiation by sorting linear resources for a faster unification in the typechecker.

[SCS24]: Stein et al. (2024), *Interactive Abstract Interpretation with Demanded Summarization*

Interactive proof mode to insert annotations On complex algorithms with complex invariants, it can be easier to use interactive software verification tools instead of writing annotations directly inside the source code. As said in the introduction, preliminary works on this PhD included a setup for annotating a program using an interactive proof mode inspired by CFML. In the end, we decided that this idea was too disconnected from the main topic of the PhD to be worth implementing during the thesis. However, integrating such interactive proof mode inside OptiTrust remains an interesting addition for the project.

Appendix

A Semantics

As said in section 4.8, we formalize the semantics of Optiλ using an omni-big-step evaluation judgment in call by value style. The judgment $t/(s, m) \Downarrow Q$ asserts that the term t, in a program stack s and in a program store m, evaluates to result states that belong to the set Q. The result states in Q are of the form (s', m') where s' is a program stack and m' a program store. Program stacks maps program variables to values, and program stores maps each location to a mode and a value. Modes in program stores, denoted $\mathcal{M}$, are either RW (read-write) or RO (read-only). A location in mode RO is shared between several threads and therefore cannot be written to. A location in mode RW is exclusively manipulated by the current execution thread and have no such restriction. The values, denoted v, can be logical expressions, locations, function closures of the form $\mathbf{fun}^s(x_1, ..., x_n) \mapsto t$, and the special uninitialized value $\bot$.

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

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

Figure 1 gives the semantic rules of Optiλ. The evaluation contexts consist of function arguments and ranges of **for** loops.

This semantics rules are standard except maybe for the rule Seq to handle sequences with optional result value. The rule Seq encode the fact that a sequence creates a lexical scope by restoring the program stack after its execution. The result value (if there is one) is bound in the output stacks.

By design, like all omni-big-step judgments, the judgment $t/(s, m) \Downarrow Q$ is preserved when enlarging Q. This property named consequence will be used in the proof of the frame rule.

Theorem A.1: Consequence property for omnisemantics

$$t/(s, m) \Downarrow Q \quad \wedge \quad Q \subseteq Q' \quad \Longrightarrow \quad t/(s, m) \Downarrow Q'$$

We refer to the omnisemantics paper [Cha+22] for the inductive proof pattern.

[Cha+22]: Charguéraud et al. (2022), *Omnisemantics: Smooth Handling of Nondeterminism*

Moreover, this semantics also ensures that the content of read-only variables in the store is preserved by the evaluation, and that these read-only variables can be promoted to read-write both in the input and the output store without changing the computed values.

Theorem A.2: Relaxing mode preserves evaluation

$$t/(s, m \uplus \mathsf{IntoRO}(m'')) \Downarrow Q \implies t/(s, m \uplus m'') \Downarrow \{(s', m' \uplus m'') \mid (s', m' \uplus \mathsf{IntoRO}(m'')) \in Q\}$$

This theorem can be proven by induction over the derivation tree of $t/(s, m \uplus \mathsf{IntoRO}(m'')) \Downarrow Q$ by noticing that none of the rules in figure 1 require a variable in the store to have mode RO, and that the only variables in the store whose value is modified must have mode RW.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1: Semantics of the Optiλ internal language in omni-big-step style as explained in section 4.8. Other arithmetic built-in functions follow the pattern of Add or InplaceAdd.

This omni-big-step semantics is enough for our current needs in OptiTrust. However, we keep in mind that in order to handle concurrency patterns that are more complex that shared read-only, we will ultimately need to change our approach. We will have to either integrate traces to model concurrent effects on the same memory location, or we can replace this omni-big-step semantics with small-step semantics with explicit thread interleaving.

Note also that the semantic rules presented in figure 1 do not specify the memory model (i.e. how are arrays and structs represented in the program store, what is inside location values, and how $\boxplus$ and $\boxdot$ are evaluated). We leave the definition of a proper memory model for future work.

B Specialization of contexts

In section 4.2, we introduced the operator $\text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ to adapt a function contract for a specific call, but we did not define formally that operator. This section contains the formal definition of the operator Specialize. The definition is quite technical—the reader may safely skip over the details.

We formally define $\text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)$ by using an auxiliary recursive function of the form $\text{Specialize}'_{E_0}\{\sigma\}(E_1, \Gamma)$, where E_0 denotes the pure part of Γ_0, and where E_1 denotes an accumulator. The operation assumes $\text{dom}(\sigma)$ to be included in set of keys of Γ.pure. The definition of Specialize′ relies itself on two auxiliary operators. The operator $\text{TypeOf}(v, E)$ returns the unique type τ such that the pure expression v has type τ in the pure context E (this corresponds to the judgment written $E \vdash v : \tau$ detailed in section 4.6). The operator $\text{Unify}(E, \tau, \tau')$ tries to unify the types τ and τ' by treating variables in E as unification variables in τ'. If it succeeds, it returns a map between the resolved unification variables to their values found by unification. Some variables from E may remain unresolved and do not appear in this map.

$$
\begin{aligned}
\text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma) &= \text{Specialize}'_{\Gamma_0.\text{pure}}\{\sigma\}(\varnothing, \Gamma) \\
\text{Specialize}'_{E_0}\{\emptyset\}(E_1, \langle E_2 \mid F\rangle) &= \langle (E_1, E_2) \mid F\rangle \\
\text{Specialize}'_{E_0}\{\sigma\}(E_1, \langle x : \tau, E_2 \mid F\rangle) &= \begin{cases}
\text{if } x \notin \text{dom}(\sigma): \\
\quad \text{Specialize}'_{E_0}\{\sigma\}((E_1, x : \tau), \langle E_2 \mid F\rangle) \\
\text{otherwise } \sigma \text{ decomposes as } (x := v) \uplus \sigma': \\
\quad \text{let } \tau_0 = \text{TypeOf}(v, E_0) \text{ in} \\
\quad \text{let } \sigma'' = \text{Unify}(E_1, \tau_0, \tau) \text{ in} \\
\quad \text{let } E'_1 = \text{Specialize}_{E_0}\{\sigma''\}([E_1]).\text{pure in} \\
\quad \text{let } \Gamma' = \text{Subst}\{\sigma'' \uplus (x := v)\}(\langle E_2 \mid F\rangle) \text{ in} \\
\quad \text{Specialize}'_{E_0}\{\sigma'\}(E'_1, \Gamma')
\end{cases}
\end{aligned}
$$

C Context satisfaction

In section 4.8, we introduced the judgment $(\sigma, \mu) \in \Gamma$ to assert that a logical state (σ, μ) satisfies a context Γ of the form $\langle E \mid F\rangle$. This section formally defines this judgment. Doing so involves two auxiliary judgments $\sigma : E$ and $F \models \mu$ that we define below.

First, $\sigma : E$ is a characterization of the fact that bindings in σ have types that correspond to the bindings in E. It allows σ to have more bindings than E. Recall that the operator $\text{TypeOf}(v, E)$ returns the unique type τ such that

the pure expression v has type τ in the pure context E. In this definition, we enforce all values in σ to be well-typed in an empty environment.

Definition C.1: Pure context satisfaction

We define $\sigma : E$ with the following rules:

$$\frac{}{\sigma : \varnothing} \qquad \frac{\mathsf{TypeOf}(\sigma(x), \varnothing) = \tau \qquad \sigma : \mathsf{Subst}\{x := \sigma(x)\}(E)}{\sigma : (x : \tau, E)}$$

$F \models \mu$ is a characterization of the fact that memory cells described by μ correspond to the linear resources described in F. Before giving its formal definitions, we need to introduce additional operators on logical stores. These definitions are essentially standard in separation logic.

We denote by $\mu_1 \uplus \mu_2$ the *compatible* union between two logical store. We denote by $\mu_1 \perp \mu_2$ the fact that two logical stores are compatible. Two logical stores are compatible if and only if, on their intersection, all the bindings have the same value.

Definition C.2: Compatibility of logical stores

$$\mu_1 \perp \mu_2 \quad = \quad \begin{array}{c} \forall l \in \mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2), \exists \alpha_1, \exists \alpha_2, \exists v, \\ \mu_1(l) = (\alpha_1, v) \wedge \mu_2(l) = (\alpha_2, v) \end{array}$$

When two logical stores are compatible, their compatible union is defined as follows:

Definition C.3: Compatible union of logical stores

Assume $\mu_1 \perp \mu_2$. Then:

$$\mu_1 \uplus \mu_2 \quad = \quad \left\{ l \mapsto (\alpha, v) \;\middle|\; \begin{array}{ll} & (\mu_1(l) = (\alpha, v) \quad \wedge \quad l \notin \mathsf{dom}(\mu_2)) \\ \vee & (\mu_2(l) = (\alpha, v) \quad \wedge \quad l \notin \mathsf{dom}(\mu_1)) \\ \vee & \left(\begin{array}{ll} & \mu_1(l) = (\alpha_1, v) \\ \wedge & \mu_2(l) = (\alpha_2, v) \\ \wedge & \alpha = \alpha_1 + \alpha_2 \end{array} \right) \end{array} \right\}$$

In program and logical stores, we allow a special value $\perp$ for variables that are not initialized. We define the fact that a linear resource H models a logical store μ recursively as follows:

Definition C.4: Linear resource satisfaction

We define $H \models \mu$ with the following rules:

$$\frac{}{l \mapsto v \models \{l \mapsto (1, v)\}} \qquad \frac{v \neq \perp}{l \rightsquigarrow \mathsf{Cell} \models \{l \mapsto (1, v)\}}$$

$$\frac{}{l \rightsquigarrow \mathsf{UninitCell} \models \{l \mapsto (1, v)\}} \qquad \frac{\forall i \in R, H_i \models \mu_i}{\bigstar_{i \in R} H_i \models \biguplus_{i \in R} \mu_i}$$

$$\frac{H \models \mu}{\alpha H \models \biguplus_{l \mapsto (\beta, v) \in \mu} \{l \mapsto (\alpha \cdot \beta, v)\}}$$

$$\frac{\forall \mu_1,\ \mu_1 \perp \mu \ \wedge\ H_1 \models \mu_1 \implies H_2 \models \mu_1 \uplus \mu}{H_1 \mathrel{-\!\!\star} H_2 \models \mu}$$

Above, all the occurrences of the operator $\biguplus$ must be well-defined.

We say that a linear context F models a logical store μ and write $F \models \mu$ if and only if the disjoint union of all resources in F models μ. Formally:

Definition C.5: Linear context satisfaction

Consider F of the form $H_0, \ldots, H_{n-1}$. The predicate $F \models \mu$ holds if and only if $(\bigstar_{i \in 0..n} H_i) \models \mu$.

With the two relations $\sigma : E$ and $F \models \mu$ defined above, we define $(\sigma, \mu) \in \Gamma$ in the following way:

Definition C.6: Context satisfaction

$$(\sigma, \mu) \in \langle E \mid F \rangle \quad = \quad \begin{array}{ll} & \sigma : E \\ \wedge & \mathsf{Subst}\{\sigma\}(F) \models \mu \end{array}$$

In order to additionally express that a context can correspond to a program state, we define the bounded context satisfaction $(\sigma, \mu) \mathrel{\overline{\in}} \Gamma$ in the following way:

Definition C.7: Bounded context satisfaction

$$(\sigma, \mu) \mathrel{\overline{\in}} \Gamma \quad = \quad \begin{array}{ll} & (\sigma, \mu) \in \Gamma \\ \wedge & (\forall l, \alpha, v,\ \mu(l) = (\alpha, v) \implies \alpha \leq 1) \end{array}$$

From these definitions of context satisfaction, we can give a semantic characterization of the separating conjunction operator $\circledast$ over two contexts. This characterization directly follows from the definitions above.

Theorem C.8: Semantic characterization of context separating conjunction

$$(\sigma_0, \mu_0) \in (\Gamma_1 \circledast \Gamma_2) \iff \begin{array}{ll} & \exists \sigma_1, \mu_1, \sigma_2, \mu_2, \\ & \quad \sigma_0 = \sigma_1 \uplus \sigma_2 \\ \wedge & \mu_0 = \mu_1 \uplus \mu_2 \\ \wedge & (\sigma_1, \mu_1) \in \Gamma_1 \\ \wedge & (\sigma_2, \mu_2) \in \mathsf{Subst}\{\sigma_1\}(\Gamma_2) \end{array}$$

We can now also give a formal definition for the entailment between two contexts:

Definition C.9: Entailment between two contexts

$$\Gamma_1 \Rrightarrow \Gamma_2 \quad = \quad \begin{array}{ll} & \forall (\sigma_1, \mu) \in \Gamma_1,\ \exists \sigma_2, \\ & \quad (\forall x \in \mathsf{dom}(\Gamma_1.\mathsf{pure}), \sigma_1(x) = \sigma_2(x)) \\ \wedge & (\sigma_2, \mu) \in \Gamma_2 \end{array}$$

D Proof of the frame rule

This section gives a proof of the frame rule for logical triples. This proof is divided in two steps. First, we need to prove correct the frame property with respect to the semantics of our language for omni-big-step evaluation judgments. Then, we can use this property to show that the frame rule for logical triples holds.

Before formally stating the frame property for omni-big-step evaluation judgment, we need one technical definition to take the compatible union

of two sets of program states. Two program stacks (respectively program stores) are compatible, and we write $s \perp s'$ (resp. $m \perp m'$), if their domain is disjoint. In that case, we write $s \uplus s'$ (resp. $m \uplus m'$) their disjoint union. We can define the compatible union of two set of program states as follows:

Definition D.1: Compatible union of program states

$$Q \uplus Q' = \{(s \uplus s', m \uplus m') \mid (s, m) \in Q \land (s', m') \in Q' \land s \perp s' \land m \perp m'\}$$

Then, the frame property for omni-big-step evaluation judgments reads as follows:

Theorem D.2: Frame property for omnisemantics

$$t/(s, m) \Downarrow Q \implies \forall s' \perp s, \forall m' \perp m,\ t/(s \uplus s', m \uplus m') \Downarrow (Q \uplus \{(s', m')\})$$

The proof sketch of this property is given in the omnisemantics paper [Cha+23, §5.4].

[Cha+23]: Charguéraud et al. (2023), *Omnisemantics: Smooth Handling of Nondeterminism*

Before expressing the frame property for logical triples, we need one last technical definition to characterize typing contexts that are well-typed. Recall that since E is a telescope, bindings defined in E can be used in the following bindings of the typing context.

Definition D.3: Well-typed contexts

A typing context $\Gamma = \langle E \mid F \rangle$ is *well-typed* if and only if there is no name conflict in E or in F and for any $x : \tau$ in E, τ is of type Type and for any $y : H$ in F, F is of type HProp.

We can now prove the frame property for logical triples:

Theorem 4.8.4: Frame property for logical triples

$$\{\Gamma\}\ t\ \{\Gamma'\} \land \Gamma \circledast \Gamma'' \text{ is well-typed} \land \Gamma' \circledast \Gamma'' \text{ is well-typed} \implies \{\Gamma \circledast \Gamma''\}\ t\ \{\Gamma' \circledast \Gamma''\}$$

Proof.

- Suppose we have $\{\Gamma\}\ t\ \{\Gamma'\}$. Let $(\sigma_0, \mu_0) \mathrel{\overline{\in}} \Gamma \circledast \Gamma''$. By definition of triples, we have to prove $t/(\sigma_0, \mu_0)_{|\text{prog}} \Downarrow \text{AcceptableStates}(\sigma_0, \mu_0, \Gamma' \circledast \Gamma'')$.
- By theorem C.8, there is a decomposition $\sigma_0 = \sigma \uplus \sigma''$ and $\mu_0 = \mu \uplus \mu''$ such that $(\sigma, \mu) \in \Gamma$ and $(\sigma'', \mu'') \in \text{Subst}\{\sigma\}(\Gamma'')$. Since μ_0 is bounded, we can deduce $(\sigma, \mu) \mathrel{\overline{\in}} \Gamma$.
- By definition of $\{\Gamma\}\ t\ \{\Gamma'\}$ applied to $(\sigma, \mu) \mathrel{\overline{\in}} \Gamma$, we have $t/(\sigma, \mu)_{|\text{prog}} \Downarrow \text{AcceptableStates}(\sigma, \mu, \Gamma')$.
- We have $(\sigma \uplus \sigma'')_{|\text{prog}} = \sigma_{|\text{prog}} \uplus \sigma''_{|\text{prog}}$ and $\sigma''_{|\text{prog}} \perp \sigma_{|\text{prog}}$.
- We pose $\hat{\mu} = \mu \backslash \text{dom}(\mu'')$, $\hat{\mu''} = \mu'' \backslash \text{dom}(\mu)$, $\mu_\cap = \mu_{|\cdot \text{dom}(\mu'')}$, and $\hat{\mu_\cap} = (\mu \uplus \mu'')_{|\cdot (\text{dom}(\mu) \cap \text{dom}(\mu''))}$. We have $\mu = \hat{\mu} \uplus \mu_\cap$, and $\mu \uplus \mu'' = \hat{\mu} \uplus \hat{\mu_\cap} \uplus \hat{\mu''}$.
- We pose $m = \hat{\mu}_{|\text{prog}}$, $m'' = \hat{\mu''}_{|\text{prog}}$, and $m_\cap = \hat{\mu_\cap}_{|\text{prog}}$. By disjointness of domains, we have $(\mu \uplus \mu'')_{|\text{prog}} = m \uplus m_\cap \uplus m''$, and $(\hat{\mu} \uplus \hat{\mu_\cap})_{|\text{prog}} = m \uplus m_\cap$, and $\mu_{|\text{prog}} = m \uplus \mu_{\cap|\text{prog}}$

- Let us show that $\mu_{\cap|\text{prog}} = \text{IntoRO}(m_\cap)$.

  First, $\text{dom}(\mu_{\cap|\text{prog}}) = \text{dom}(\mu_\cap) = \text{dom}(\mu) \cap \text{dom}(\mu'')$, and $\text{dom}(\text{IntoRO}(m_\cap)) = \text{dom}(m_\cap) = \text{dom}(\hat{\mu_\cap}) = \text{dom}(\mu) \cap \text{dom}(\mu'')$, thus $\text{dom}(\mu_{\cap|\text{prog}}) = \text{dom}(\text{IntoRO}(m_\cap))$.

  Take $l \in \text{dom}(\mu) \cap \text{dom}(\mu'')$, we need to show that $\mu_{\cap|\text{prog}}(l) = (\text{IntoRO}(m_\cap))(l)$. By definition, $\mu_{\cap|\text{prog}}(l) = \mu_{|\text{prog}}(l)$. Similarly, $(\text{IntoRO}(m_\cap))(l) = \text{IntoRO}(\hat{\mu_\cap}_{|\text{prog}})(l) = \text{IntoRO}((\mu \uplus \mu'')_{|\text{prog}})(l)$.

  Let us write $\mu(l) = (\alpha, v)$, and $\mu''(l) = (\alpha'', v'')$. By definition of $\mu \uplus \mu''$, $(\mu \uplus \mu'')(l) = (\alpha + \alpha'', v)$. Since additionally $\mu \uplus \mu''$ is bounded, we know that $\alpha + \alpha'' \leq 1$. Thus, there exist $\mathcal{M}$ such that $(\mu \uplus \mu'')_{|\text{prog}} = (\mathcal{M}, v)$. By definition of IntoRO, we obtain $(\text{IntoRO}(m_\cap))(l) = (\text{RO}, v)$.

  Like all fractions, $\alpha'' > 0$. Therefore, from the inequality $\alpha + \alpha'' \leq 1$, we can deduce $\alpha < 1$ and thus $\mu_{|\text{prog}}(l) = (\text{RO}, v)$.

  Thus, we indeed have $\mu_{\cap|\text{prog}} = \text{IntoRO}(m_\cap)$.

- By theorem A.2 applied to $t/(\sigma, \mu)_{|\text{prog}} \Downarrow \text{AcceptableStates}(\sigma, \mu, \Gamma')$ where $\mu_{|\text{prog}} = m \uplus \text{IntoRO}(m_\cap)$, we obtain $t/(\sigma_{|\text{prog}}, m \uplus m_\cap) \Downarrow Q$, where:

$$Q = \left\{ (s', m' \uplus m_\cap) \;\middle|\; \begin{array}{c} (s', m' \uplus \text{IntoRO}(m_\cap)) \in \\ \text{AcceptableStates}(\sigma, \mu, \Gamma') \end{array} \right\}$$

- By the frame property for omni-big-step (theorem D.2) applied on $t/(\sigma_{|\text{prog}}, m \uplus m_\cap) \Downarrow Q$, ${\sigma''}_{|\text{prog}} \perp \sigma_{|\text{prog}}$, and $m'' \perp (m \uplus m_\cap)$, we obtain $t/(\sigma_{|\text{prog}} \uplus {\sigma''}_{|\text{prog}}, m \uplus m_\cap \uplus m'') \Downarrow (Q \uplus \{({\sigma''}_{|\text{prog}}, m'')\})$.

- Since $(\sigma_{|\text{prog}} \uplus {\sigma''}_{|\text{prog}}, m \uplus m_\cap \uplus m'') = (\sigma_0, \mu_0)_{|\text{prog}}$, by the consequence property of omni-big-step (theorem A.1), it suffices to show $(Q \uplus \{({\sigma''}_{|\text{prog}}, m'')\}) \subseteq \text{AcceptableStates}(\sigma_0, \mu_0, \Gamma' \circledast \Gamma'')$.

  Take $(s_r, m_r) \in (Q \uplus \{({\sigma''}_{|\text{prog}}, m'')\})$. We need to show that $(s_r, m_r) \in \text{AcceptableStates}(\sigma_0, \mu_0, \Gamma' \circledast \Gamma'')$.

- There is a decomposition $s_r = s'_r \uplus s''_r$ and $m_r = m'_r \uplus m''_r$ such that $(s'_r, m'_r) \in Q$ and $(s''_r, m''_r) \in \{({\sigma''}_{|\text{prog}}, m'')\}$. Since $\{({\sigma''}_{|\text{prog}}, m'')\}$ is a singleton, we have $s''_r = {\sigma''}_{|\text{prog}}$ and $m''_r = m''$.

- By definition of Q, there exist m' such that $m'_r = m' \uplus m_\cap$, and $(s'_r, m' \uplus \text{IntoRO}(m_\cap)) \in \text{AcceptableStates}(\sigma, \mu, \Gamma')$. By definition of AcceptableStates (definition 4.8.2), there exist σ' and μ' such that $s'_r = {\sigma'}_{|\text{prog}}$, and $m' \uplus \text{IntoRO}(m_\cap) = {\mu'}_{|\text{prog}}$, and $(\sigma', \mu') \mathrel{\overline{\in}} \Gamma'$, and $\text{OnlyRO}(\mu) = \text{OnlyRO}(\mu')$, and $\forall x \in \text{dom}(\sigma) \cap \text{dom}(\sigma'), \sigma(x) = \sigma'(x)$.

- $s_r = (s'_r \uplus s''_r) = ({\sigma'}_{|\text{prog}} \uplus {\sigma''}_{|\text{prog}}) = (\sigma' \uplus \sigma'')_{|\text{prog}}$

- We know that $\Gamma' \circledast \Gamma''$ is well-scoped and that $\sigma' : \Gamma'.\text{pure}$. Therefore, for any x free in Γ'', $x \in \text{dom}(\Gamma') \subseteq \text{dom}(\sigma')$. Similarly, we know that for any x free in Γ'', $x \in \text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$. Therefore, since $\forall x \in \text{dom}(\sigma) \cap \text{dom}(\sigma'), \sigma(x) = \sigma'(x)$, for any x free in Γ'', $\sigma(x) = \sigma'(x)$. This implies $\text{Subst}\{\sigma\}(\Gamma'') = \text{Subst}\{\sigma'\}(\Gamma'')$. We know that $(\sigma'', \mu'') \in \text{Subst}\{\sigma\}(\Gamma'')$. Thus, $(\sigma'', \mu'') \in \text{Subst}\{\sigma'\}(\Gamma'')$.

- Let us show that $\mu' \uplus \mu''$ is well-defined. Take $l \in \text{dom}(\mu') \cap \text{dom}(\mu'')$. By writing $\mu'(l) = (\alpha', v')$ and $\mu''(l) = (\alpha'', v'')$, we need to show that $v' = v''$.

  We know that ${\mu'}_{|\text{prog}} = m' \uplus \text{IntoRO}(m_\cap)$. Therefore, either $l \in \text{dom}(m')$ or $l \in \text{dom}(\text{IntoRO}(m_\cap)) = \text{dom}(m_\cap)$. Similarly, we can

show that $\mu''_{|\text{prog}} = m'' \uplus \text{IntoRO}(m_\cap)$. Therefore, either $l \in \text{dom}(m'')$ or $l \in \text{dom}(m_\cap)$.

We know that $m_r = m' \uplus m_\cap \uplus m''$, thus m', $m_\cap$ and m'' are disjoint. We can therefore deduce that necessarily $l \in \text{dom}(m_\cap)$. If we write $m_\cap(l) = (\mathcal{M}, v_\cap)$, then by definition of $\mu'_{|\text{prog}}$ we can deduce that $v' = v_\cap$, and similarly that $v'' = v_\cap$. Thus, we obtain $v' = v''$.

- We know that $\sigma' \uplus \sigma''$ and $\mu' \uplus \mu''$ are well-defined, and that $(\sigma', \mu') \in \Gamma'$ and $(\sigma'', \mu'') \in \text{Subst}\{\sigma'\}(\Gamma'')$. Thus by theorem C.8, $(\sigma' \uplus \sigma'', \mu' \uplus \mu'') \in (\Gamma' \circledast \Gamma'')$.
- Let us show that $\mu' \uplus \mu''$ is bounded. Let us take $l \in \text{dom}(\mu' \uplus \mu'')$. By writing $(\mu' \uplus \mu'')(l) = (\alpha, v)$, we need to show that $\alpha \leq 1$. Let us distinguish three cases:
  - Suppose $l \in \text{dom}(\mu')$ and $l \notin \text{dom}(\mu'')$, then $(\mu' \uplus \mu'')(l) = \mu'(l)$. Since we know by $(\sigma', \mu') \overline{\in} \Gamma'$ that μ' is bounded, then $\alpha \leq 1$.
  - Suppose $l \in \text{dom}(\mu'')$ and $l \notin \text{dom}(\mu')$, then $(\mu' \uplus \mu'')(l) = \mu''(l)$. By definition of $(\sigma_0, \mu_0) \overline{\in} \Gamma \circledast \Gamma''$, we know that $\mu \uplus \mu''$ is bounded. Therefore, μ'' is also bounded and thus $\alpha \leq 1$.
  - Suppose $l \in \text{dom}(\mu') \cap \text{dom}(\mu'')$. If we write $\mu'(l) = (\alpha', v)$ and $\mu''(l) = (\alpha'', v)$, then by definition of $\mu' \uplus \mu''$, we know that $\alpha = \alpha' + \alpha''$.

    We know that $\mu'_{|\text{prog}} = m' \uplus \text{IntoRO}(m_\cap)$, and by the same reasoning as before, we can say that $l \in \text{dom}(\text{IntoRO}(m_\cap))$. Thus, by definition of $\mu'_{|\text{prog}}$, we obtain $\alpha' < 1$, and therefore $l \in \text{dom}(\text{OnlyRO}(\mu'))$.

    As $\text{OnlyRO}(\mu) = \text{OnlyRO}(\mu')$, we have $\mu(l) = \mu'(l) = (\alpha', v)$. We know that $\mu \uplus \mu''$ is bounded. Therefore, we can conclude $\alpha' + \alpha'' \leq 1$.

  In all cases $\alpha \leq 1$, therefore $\mu' \uplus \mu''$ is bounded and thus, $(\sigma' \uplus \sigma'', \mu' \uplus \mu'') \overline{\in} (\Gamma' \circledast \Gamma'')$.
- $\text{OnlyRO}(\mu \uplus \mu'') = \text{OnlyRO}(\mu' \uplus \mu'')$ directly follows from $\text{OnlyRO}(\mu') = \text{OnlyRO}(\mu)$.
- $\forall x \in \text{dom}(\sigma \uplus \sigma'') \cap \text{dom}(\sigma' \uplus \sigma''), (\sigma \uplus \sigma'')(x) = (\sigma' \uplus \sigma'')(x)$ directly follows from $\forall x \in \text{dom}(\sigma) \cap \text{dom}(\sigma'), \sigma(x) = \sigma'(x)$.
- We conclude by instantiating the definition 4.8.2 of AcceptableStates($\sigma_0, \mu_0, \Gamma' \circledast \Gamma''$) with $s_r = (\sigma' \uplus \sigma'')_{|\text{prog}}$, and $m_r = (\mu' \uplus \mu'')_{|\text{prog}}$, and $(\sigma' \uplus \sigma'', \mu' \uplus \mu'') \overline{\in} \Gamma' \circledast \Gamma''$, and $\text{OnlyRO}(\mu \uplus \mu'') = \text{OnlyRO}(\mu' \uplus \mu'')$, and $\forall x \in \text{dom}(\sigma \uplus \sigma'') \cap \text{dom}(\sigma' \uplus \sigma''), (\sigma \uplus \sigma'')(x) = (\sigma' \uplus \sigma'')(x)$.

□

E Soundness of the algorithmic rule for typechecking for loops

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

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

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

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

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

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

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

Theorem E.1: Specialization of a triple

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

This is straightforward from the implication $(\sigma_0, \mu) \mathrel{\overline{\in}} (\Gamma_0 \circledast \text{Specialize}_{\Gamma_0}\{\sigma\}(\Gamma)) \implies (\sigma_0 \uplus \sigma, \mu) \mathrel{\overline{\in}} \Gamma$.

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

Theorem E.2: Specialized context entails itself

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

This theorem directly follows from definition C.9 by choosing $\sigma_2 = \sigma \uplus \sigma_1$.

Now let us prove our main theorem:

Theorem E.3: For is derivable from ForSeq and ForPar

The algorithmic typing rule For can be derived from the standard separation logic rules ForSeq and ForPar.

Proof. We need to make a case analysis on whether the loop is parallel or not which corresponds to the component π:

- If $\pi = \textbf{seq}$ (the loop is sequential), our For rule can be derived from the rule ForSeq by placing in Γ_{inv} all the exclusive per-iteration resources produced by iterations strictly before j, all the exclusive per-iterations resources consumed by iterations after and including j, and all the

resources shared across iterations. More formally, we use the instantiations:

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

From our rule For, we know that $\{[\Gamma_0.\text{pure},\ i : \text{int},\ i \in R] \circledast \Gamma^{\text{in}}_{\text{pre}}\}\ t\ \{\Gamma^{\text{in}}_{\text{post}}\}$. By theorem E.1 applied with σ^{inv}, we therefore know that $\{[\Gamma_0.\text{pure},\ i : \text{int},\ i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}})\}\ t\ \{\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}})\}$. By adding a frame $\Gamma^{\text{in}}_{\text{frame}}$, defined as:

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

we obtain $\{[\Gamma_0.\text{pure}, i : \text{int}, i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) \circledast \Gamma^{\text{in}}_{\text{frame}}\}\ t\ \{\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}}) \circledast \Gamma^{\text{in}}_{\text{frame}}\}$. Then, by the consequence rule, we can deduce the required hypothesis $\{[E_0, i : \text{int}, i \in R] \circledast \Gamma_{\text{inv}}(i)\}\ t\ \{\Gamma_{\text{inv}}(i + R.\text{step})\}$ for applying ForSeq by proving two entailments:

- $[E_0,\ i : \text{int},\ i \in R] \circledast \Gamma_{\text{inv}}(i) \implies [\Gamma_0.\text{pure},\ i : \text{int},\ i \in R] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) \circledast \Gamma^{\text{in}}_{\text{frame}}$

  We know that $\Gamma^{\text{in}}_{\text{pre}} = [\chi.\text{vars}] \circledast \chi.\text{excl.pre} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv}$. By definition of Specialize and σ^{inv}, we have $\text{Specialize}_{\Gamma_0}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{pre}}) = \text{Subst}\{\sigma^{\text{inv}}\}(\chi.\text{excl.pre.linear} \circledast \frac{1}{R.\text{len}}\chi.\text{shrd.reads} \circledast \chi.\text{shrd.inv})$.

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

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

- $\text{Subst}\{\sigma^{\text{inv}}\}(\Gamma^{\text{in}}_{\text{post}}) \circledast \Gamma^{\text{in}}_{\text{frame}} \Rightarrow \Gamma_{\text{inv}}(i + R.\text{step})$

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

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

the following equality:

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

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

- $\Gamma_0 \Rightarrow [E_0] \circledast \Gamma_{\mathsf{inv}}(R.\mathsf{start}) \circledast F$

  We know by the subtraction $\Gamma_0 \ominus \Gamma^{\mathsf{out}}_{\mathsf{pre}}$ that $\Gamma_0 \Rightarrow [E_{\mathsf{frac}}] \circledast \mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\Gamma^{\mathsf{out}}_{\mathsf{pre}}) \circledast F$. As $\Gamma^{\mathsf{out}}_{\mathsf{pre}} = [\chi.\mathsf{vars}] \circledast (\circledast_{i\in R}\ \chi.\mathsf{excl.pre}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})$, by definition of Specialize and σ^{inv} and since we know σ^{out} covers all the pure bindings of $\Gamma^{\mathsf{out}}_{\mathsf{pre}}$, we have: $\mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\}(\Gamma^{\mathsf{out}}_{\mathsf{pre}}) = \mathsf{Subst}\{\sigma^{\mathsf{inv}}\}((\bigstar_{i\in R}\ \chi.\mathsf{excl.pre.linear}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\restriction\cdot\mathsf{dom}(\chi.\mathsf{shrd.inv.pure})\}(\mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})))$.

  By theorem E.2, $\mathsf{Specialize}_{\Gamma_0}\{\sigma^{\mathsf{out}}\restriction\cdot\mathsf{dom}(\chi.\mathsf{shrd.inv.pure})\}(\mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})) \Rightarrow \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})$. Therefore, $\Gamma_0 \Rightarrow [\Gamma_0.\mathsf{pure}] \circledast \mathsf{Subst}\{\sigma^{\mathsf{inv}}\}((\bigstar_{i\in R}\ \chi.\mathsf{excl.pre.linear}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{start}\}(\chi.\mathsf{shrd.inv})) \circledast F$.

  To conclude, we can introduce $\chi.\mathsf{excl.post}$ and find $\Gamma_{\mathsf{inv}}(R.\mathsf{start})$ using the equality $\varnothing = \circledast_{i\in\mathbf{range}(R.\mathsf{start},R.\mathsf{start},R.\mathsf{step})}\ \chi.\mathsf{excl.post}$.

- $\Gamma_{\mathsf{inv}}(R.\mathsf{end}) \circledast F \Rightarrow \Gamma_r$

  We know that $\Gamma_r = \mathsf{CloseFracs}([\Gamma_0.\mathsf{pure},\ E_{\mathsf{frac}}] \circledast F \circledast \mathsf{Subst}\{\sigma^{\mathsf{out}}\}((\circledast_{i\in R}\ \chi.\mathsf{excl.post}) \circledast \chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := R.\mathsf{end}\}(\chi.\mathsf{shrd.inv})))$. By scoping rules of the loop contracts, bindings of $\chi.\mathsf{shrd.inv}$ cannot occur in other elements of χ. Therefore, in this context σ^{out} and σ^{inv} produce the same substitution.

  To conclude, we use the fact that the range $\mathbf{range}(R.\mathsf{stop},\ R.\mathsf{stop},\ R.\mathsf{step})$ of the iterated separating conjunction over $\chi.\mathsf{excl.pre.linear}$ from $\Gamma_{\mathsf{inv}}(R.\mathsf{stop})$ is empty and notice that for any context Γ, $\Gamma \Leftrightarrow \mathsf{CloseFracs}(\Gamma)$.

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

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

We directly have $\Gamma^{\mathsf{in}}_{\mathsf{pre}} = \Gamma_{\mathsf{pre}}(i)$. From the hypothesis $(\sigma^{\mathsf{in}}_{\mathsf{post}}, \varnothing) = \Gamma^{\mathsf{in}}_{\mathsf{post}} \boxminus \chi.\mathsf{excl.post} \circledast \frac{1}{R.\mathsf{len}}\chi.\mathsf{shrd.reads} \circledast \mathsf{Subst}\{i := i + R.\mathsf{step}\}(\chi.\mathsf{shrd.inv})$, we obtain $\Gamma^{\mathsf{in}}_{\mathsf{post}} \Rightarrow \Gamma_{\mathsf{post}}(i)$. From the hypothesis $\{[\Gamma_0.\mathsf{pure},\ i : \mathsf{int},\ i \in R] \circledast \Gamma^{\mathsf{in}}_{\mathsf{pre}}\}\ t\ \{\Gamma^{\mathsf{in}}_{\mathsf{post}}\}$, we can therefore deduce $\{[E_0,\ i : \mathsf{int},\ i \in R] \circledast \Gamma_{\mathsf{pre}}(i)\}\ t\ \{\Gamma_{\mathsf{post}}(i)\}$ and apply the rule ForPar.

Now, we know by the rule ForPar that $\{[E_0] \circledast \circledast_{i\in R} \Gamma_{\text{pre}}(i)\}$ $\mathbf{for}^{\mathbf{par}}\ (i \in R)\ t\ \{\circledast_{i\in R} \Gamma_{\text{post}}(i)\}$. By theorem E.1, we have $\{[E_0] \circledast \circledast_{i\in R} \text{Specialize}_{\Gamma_0}\{\sigma^{\text{out}}\}(\Gamma_{\text{pre}}(i))\}$ $\mathbf{for}^{\mathbf{par}}\ (i \in R)\ t$ $\{\text{Subst}\{\sigma^{\text{out}}\}(\circledast_{i\in R} \Gamma_{\text{post}}(i))\}$.

To conclude, we frame the linear resource set F (that comes from the rule For itself) and prove the following entailments:

- $\Gamma_0 \Rightarrow [E_0] \circledast \text{Specialize}_{\Gamma_0}\{\sigma^{\text{out}}\}(\circledast_{i\in R} \Gamma_{\text{pre}}(i)) \circledast F$

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

- $\text{Subst}\{\sigma^{\text{out}}\}(\circledast_{i\in R} \Gamma_{\text{post}}(i)) \circledast F \Rightarrow \Gamma_r$

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

□

F Details of triple minimization

In the description of the triple minimization operator in section 5.4, we did not explain how it is computed. This section gives the missing implementation details.

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

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

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

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

Algorithmically, Minimize can be defined by iterating over its first argument.

Start with $E^{\text{fracs}} = \emptyset$, $\hat{F} = \emptyset$, $\hat{F}' = \Gamma'.\text{linear}$ and $F^{\text{framed}} = \emptyset$.

For each binding $y : H$ in Γ, lookup y in Δ:

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

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

G Example typechecking of subexpressions

This section presents an example application of the Subexpr from section 5.5 and repeated below:

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

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

Figure 2 shows the typechecking steps. The figure includes 4 columns, describing the steps associated with each of the 4 subterms. The rows explain how the metavariables from the rule Subexpr are instantiated. In particular, observe how the two subexpressions $\text{get}(p)$ both have read-only access to

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

Figure 2: Example of an application of the SUBEXPR rule on an expression $\hat{\mathcal{E}}[q, \text{get\_incr(c)}, \text{get}(p), \text{get}(p)]$, in a context $\langle p, q, c : \text{ptr(int)} \mid y_p : p \rightsquigarrow \text{Cell}, y_q : q \rightsquigarrow \text{Cell}, y_c : c \rightsquigarrow \text{Cell} \rangle$.

the same resource $H = p \rightsquigarrow \text{Cell}$. As the details in the figure show, the first get(p), according to its minimized precondition, only needs a fraction of H. This fraction is carved out, obtaining a subfraction αH and leaving $(1-\alpha)H$ for the second get(p).

H Details of loop minimization

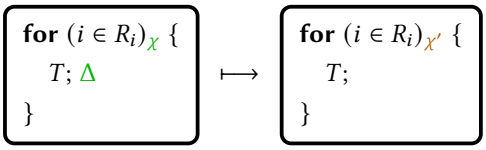
Figure 3 describes the loop minimization transformation. Essentially, it uses the Minimize operator to minimize the exclusive part of the loop contract, and it tries to reduce the footprint of the shared part of the loop contract by taking arbitrary subfractions and using shared reads whenever possible.

For that we overload the operator IntoRO to operate on linear contexts. This overloading is defined recursively as follows:

$$\text{IntoRO}(F) = \begin{cases} \langle | \rangle & \text{if } F = \varnothing \\ \langle \alpha : \text{frac} \mid y : \alpha H \rangle \circledast \text{IntoRO}(F') & \text{if } F = (y : H) \star F' \end{cases}$$

For pure variables, it simply removes those that are not used and adds the new arbitrary fractions generated during the previous steps.

One technical difficulty: postcondition of a loop contract uses names for linear resources, and these names must be matched to corresponding resources at the end of the body of the loop. In fact our typechecker had to prove an entailment there. We can remember the map ς from linear resource names at the end of the loop body to linear resources names in the postcondition, and use it in the loop minimization transformation.



with:

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

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

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

Figure 3: The basic transformation `Loop.minimize`. The Minimize operation is that defined for triples in section 5.4. $\text{usedVars}(X)$ is the set of all variables used in X at least once.

Bibliography

[Ala+24] Sami Alabed, Daniel Belov, Bart Chrzaszcz, Juliana Franco, Dominik Grewe, Dougal Maclaurin, James Molloy, Tom Natan, Tamara Norman, Xiaoyue Pan, Adam Paszke, Norman A. Rink, Michael Schaarschmidt, Timur Sitdikov, Agnieszka Swietlik, Dimitrios Vytiniotis, and Joel Wee. *PartIR: Composing SPMD Partitioning Strategies for Machine Learning*. 2024. URL: https://arxiv.org/abs/2401.11202 (cited on page 44).

[AG22] Étienne Alepins and Jens Gustedt. *Unsequenced functions*. ISO/IEC JCT1/SC22/WG14 document N2956. Apr. 2022. URL: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2956.htm (cited on page 105).

[Ama+20] Vasco Amaral, Beatriz Norberto, Miguel Goulão, Marco Aldinucci, Siegfried Benkner, Andrea Bracciali, Paulo Carreira, Edgars Celms, Luís Correia, Clemens Grelck, Helen Karatza, Christoph Kessler, Peter Kilpatrick, Hugo Martiniano, Ilias Mavridis, Sabri Pllana, Ana Respício, José Simão, Luís Veiga, and Ari Visa. "Programming languages for data-Intensive HPC applications: A systematic mapping study". In: *Parallel Computing* 91 (2020), p. 102584. DOI: https://doi.org/10.1016/j.parco.2019.102584 (cited on page 9).

[Arm+11] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150 (cited on page 17).

[Bag+19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code". In: *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. Ed. by Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley. IEEE, 2019, pp. 193–205. DOI: 10.1109/CGO.2019.8661197 (cited on page 128).

[Bag+16] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. "Opening Polyhedral Compiler's Black Box". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2016 (cited on pages 45, 128).

[BI19] Paul Barham and Michael Isard. "Machine Learning Systems are Stuck in a Rut". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 177–183. DOI: 10.1145/3317550.3321441 (cited on pages 9, 44).

[Bar+18] Y. Barsamian, A. Charguéraud, S. A. Hirstoaga, and M. Mehrenberger. "Efficient Strict-Binning Particle-in-Cell Algorithm for Multi-Core SIMD Processors". In: *24th International Conference on Parallel and Distributed Computing (Euro-Par)*. Vol. 11014. Lecture Notes in Computer Science. Springer, Cham, 2018, pp. 749–763. DOI: 10.1007/978-3-319-96983-1_53 (cited on pages 31, 32).

[Bar18] Yann Barsamian. “Pic-Vert: A Particle-in-Cell Implementation for Multi-Core Architectures”. PhD thesis. Université de Strasbourg, 2018 (cited on page 9).

[Bar+09] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. “Certificate Translation for Optimizing Compilers”. In: *ACM Trans. Program. Lang. Syst.* 31.5 (2009). DOI: 10.1145/1538917.1538919 (cited on page 19).

[Bau+20] Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. *WP plug-in manual*. 2020 (cited on page 16).

[Ben04] Nick Benton. “Simple relational correctness proofs for static analyses and program transformations”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’04. Venice, Italy: Association for Computing Machinery, 2004, pp. 14–25. DOI: 10.1145/964001.964003 (cited on page 18).

[BC23] Guillaume Bertholon and Arthur Charguéraud. “An AST for Representing Programs with Invariants and Proofs”. In: *34èmes Journées Francophones des Langages Applicatifs (JFLA 2023)*. 2023, pp. 43–58 (cited on pages 17, 21).

[BC25a] Guillaume Bertholon and Arthur Charguéraud. “Bidirectional Translation between a C-like Language and an Imperative λ-calculus”. In: *36èmes Journées Francophones des Langages Applicatifs (JFLA 2025)*. 2025 (cited on page 21).

[BC25b] Guillaume Bertholon and Arthur Charguéraud. “Bidirectional Translation between a C-like Language and an Imperative Lambda-calculus”. In: *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*. Roiffé, France, Jan. 2025 (cited on page 64).

[Ber+24a] Guillaume Bertholon, Arthur Charguéraud, Kœhler, Begatim Bytyqi, and Damien Rouhling. “OptiTrust: Producing Trustworthy High-Performance Code via Source-to-Source Transformations”. Draft paper. 2024 (cited on page 22).

[Ber+24b] Guillaume Bertholon, Arthur Charguéraud, Thomas Kœhler, Begatim Bytyqi, and Damien Rouhling. “Interactive source-to-source optimizations validated using static resource analysis”. In: *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. 2024, pp. 26–34 (cited on page 21).

[BC20] João Bispo and João MP Cardoso. “Clava: C/C++ source-to-source compilation using LARA”. In: *SoftwareX* 12 (2020), p. 100565 (cited on page 45).

[BP13] Jasmin Christian Blanchette and Lawrence C Paulson. “Hammering Away”. In: *A User’s Guide to Sledgehammer for Isabelle/HOL. url: http://isabelle. in. tum. de/dist/Isabelle2013-2/doc/sledgehammer. pdf* (2013) (cited on page 17).

[Blo+17] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. “The VerCors Tool Set: Verification of Parallel and Concurrent Software”. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110 (cited on page 45).

[BM11] Sylvie Boldo and Guillaume Melquiond. “Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq”. In: *2011 IEEE 20th Symposium on Computer Arithmetic*. 2011, pp. 243–252. DOI: 10.1109/ARITH.2011.40 (cited on page 130).

[Boy03] John Boyland. “Checking Interference with Fractional Permissions”. In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. Ed. by Radhia Cousot. Vol. 2694. Lecture Notes in Computer Science. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on page 71).

[BK+00] Gary Bradski, Adrian Kaehler, et al. “OpenCV”. In: *Dr. Dobb’s journal of software tools* 3.2 (2000). https://opencv.org/ (cited on page 26).

[BDG19] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. “Using Standard Typing Algorithms Incrementally”. In: *NASA Formal Methods*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Cham: Springer International Publishing, 2019, pp. 106–122 (cited on page 132).

[Cal+19] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok. Yang. *Go Huge or Go Home: POPL’19 Most Influential Paper Retrospective.* 2019. URL: https://blog.sigplan.org/2020/03/03/go-huge-or-go-home-popl19-most-influentialpaper-retrospective/ (cited on page 128).

[Cao+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. “VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs”. In: *J. Autom. Reason.* 61.1-4 (2018), pp. 367–422. DOI: 10.1007/S10817-018-9457-5 (cited on page 62).

[Cha10] Arthur Charguéraud. “Program Verification Through Characteristic Formulae”. In: *International Conference on Functional Programming (ICFP)*. Sept. 2010, pp. 321–332 (cited on page 16).

[Cha20a] Arthur Charguéraud. “Separation Logic for Sequential Programs (Functional Pearl)”. In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: 10.1145/3408998 (cited on page 70).

[Cha20b] Arthur Charguéraud. “Separation logic for sequential programs (functional pearl)”. In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: 10.1145/3408998 (cited on page 23).

[Cha+22] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. “Omnisemantics: Smooth Handling of Nondeterminism”. To appear in ACM Transactions on Programming Languages and Systems (TOPLAS). Sept. 2022 (cited on page 135).

[Cha+23] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. “Omnisemantics: Smooth Handling of Nondeterminism”. In: *ACM Trans. Program. Lang. Syst.* 45.1 (Mar. 2023). DOI: 10.1145/3579834 (cited on pages 87, 140).

[Cha+15] Gaurav Chaurasia, Jonathan Ragan-Kelley, Sylvain Paris, George Drettakis, and Frédo Durand. “Compiling high performance recursive filters”. In: *Proceedings of the 7th Conference on High-Performance Graphics, HPG 2015, Los Angeles, California, USA, August 7-9, 2015*. Ed. by Michael C. Doggett, Steven E. Molnar, Kayvon Fatahalian, Jacob Munkberg, Elmar Eisemann, Petrik Clarberg, and Stephen N. Spencer. ACM, 2015, pp. 85–94. DOI: 10.1145/2790060.2790063 (cited on page 27).

[Che+21] Lorenzo Chelini, Martin Kong, Tobias Grosser, and Henk Corporaal. "LoopOpt: Declarative Transformations Made Easy". In: *Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '21. Eindhoven, Netherlands: Association for Computing Machinery, 2021, pp. 11–16. DOI: 10.1145/3493229.3493301 (cited on page 45).

[Che+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *OSDI*. USENIX Association, 2018 (cited on pages 9, 35, 44).

[Dal21] Bill Dally. *The Future of Computing: Domain-Specific Architecture*. Keynote at Chesapeake Large Scale Analytics Conference (CLSAC). Oct. 2021. URL: https://www.clsac.org/uploads/5/0/6/3/50633811/2021-clsac-dally.pdf (cited on page 7).

[DMS22] Thibault Dardinier, Peter Müller, and Alexander J. Summers. "Fractional resources in unbounded separation logic". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563326 (cited on pages 72, 87, 126).

[DLM11] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. "Certifying the Floating-Point Implementation of an Elementary Function Using Gappa". In: *IEEE Transactions on Computers* 60.2 (2011), pp. 242–253. DOI: 10.1109/TC.2010.128 (cited on page 130).

[Els24] Martin Elsman. "Double-Ended Bit-Stealing for Algebraic Data Types". In: *Proc. ACM Program. Lang.* 8.ICFP (Aug. 2024). DOI: 10.1145/3674628 (cited on page 124).

[EM19] Gidon Ernst and Toby Murray. "SecCSL: Security Concurrent Separation Logic". In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 208–230 (cited on page 127).

[Eva+22] Thomas M Evans, Andrew Siegel, Erik W Draeger, Jack Deslippe, Marianne M Francois, Timothy C Germann, William E Hart, and Daniel F Martin. "A survey of software implementations used by application codes in the Exascale Computing Project". In: *The International Journal of High Performance Computing Applications* 36.1 (2022), pp. 5–12. DOI: 10.1177/10943420211028940 (cited on page 9).

[Fea92] Paul Feautrier. "Some efficient solutions to the affine scheduling problem: one dimensional time". In: *Intl. Journal of Parallel Programming* 21.5 (Oct. 1992), pp. 313–348 (cited on page 45).

[Fil03] Jean-Christophe Filliâtre. *Why: a multi-language multi-prover verification tool*. Research Report 1366. LRI, Université Paris Sud, Mar. 2003 (cited on page 74).

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—Where Programs Meet Provers". In: *European Symposium on Programming (ESOP)*. Vol. 7792. Lecture Notes in Computer Science. Springer, Mar. 2013, pp. 125–128 (cited on pages 16, 124).

[Fla+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. "Extended Static Checking for Java". In: *Programming Language Design and Implementation (PLDI)*. 2002, pp. 234–245 (cited on page 74).

[Gäh+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. "Simuliris: a separation logic framework for verifying concurrent program optimizations". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498689 (cited on pages 18, 119).

[Góm+20] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. "Do you have space for dessert? a verified space cost semantics for CakeML programs". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428272 (cited on page 127).

[Gué19] Armaël Guéneau. "Mechanized Verification of the Correctness and Asymptotic Complexity of Programs". PhD thesis. Université de Paris, Dec. 2019 (cited on page 127).

[Hag+20a] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. "Fireiron: a data-movement-aware scheduling language for GPUs". In: *Proceedings of the ACM International Conf. on Parallel Architectures and Compilation Techniques*. 2020, pp. 71–82 (cited on page 44).

[Hag+20b] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies". In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: 10.1145/3408974 (cited on pages 9, 44).

[Ika+22] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. "Exocompilation for productive programming of hardware accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718. DOI: 10.1145/3519939.3523446 (cited on pages 9, 44, 125).

[Ika+25] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. "Exo 2: Growing a Scheduling Language". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 426–444. DOI: 10.1145/3669940.3707218 (cited on page 44).

[Ika+21] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. "Guided Optimization for Image Processing Pipelines". In: *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2021, pp. 1–5. DOI: 10.1109/VL/HCC51201.2021.9576341 (cited on pages 9, 44).

[JP08] Bart Jacobs and Frank Piessens. *The VeriFast Program Verifier*. Tech. rep. CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008 (cited on page 126).

[JM18] Matthieu Journault and Antoine Miné. "Inferring functional properties of matrix manipulating programs by abstract interpretation". In: *Form. Methods Syst. Des.* 53.2 (2018), pp. 221–258. DOI: 10.1007/s10703-017-0311-x (cited on pages 10, 128).

[Jun+18a] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on page 71).

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *Journal of Functional Programming* 28 (2018), e20 (cited on page 16).

[Kan+24] Yamato Kanetaka, Hiroyasu Takagi, Yoshihiro Maeda, and Norishige Fukushima. "SlidingConv: Domain-Specific Description of Sliding Discrete Cosine Transform Convolution for Halide". In: *IEEE Access* 12 (2024), pp. 7563–7583. DOI: 10.1109/ACCESS.2023.3345660 (cited on page 27).

[KK22] Vasilios Kelefouras and Georgios Keramidas. "Design and Implementation of 2D Convolution on x86/x64 Processors". In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 3800–3815. DOI: 10.1109/TPDS.2022.3171471 (cited on page 9).

[Kre15] Robbert Krebbers. "The C standard formalized in Coq". PhD thesis. Radboud University Nijmegen, 2015 (cited on page 60).

[Kum+14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *POPL*. San Diego, California, USA: Association for Computing Machinery, 2014. DOI: 10.1145/2535838.2535841 (cited on page 18).

[Kun09] César Kunz. "Proof preservation and program compilation". PhD thesis. École Nationale Supérieure des Mines de Paris, 2009 (cited on page 19).

[Ler06] Xavier Leroy. "Coinductive Big-Step Operational Semantics". In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 54–68 (cited on page 123).

[Ler09] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814 (cited on page 18).

[Ler+12] Xavier Leroy, Andrew W Appel, Sandrine Blazy, and Gordon Stewart. "The CompCert Memory Model, Version 2". In: (2012) (cited on page 120).

[Liu+24] Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. "A Verified Compiler for a Functional Tensor Language". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656390 (cited on page 18).

[Liu+22] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. "Verified Tensor-Program Optimization via High-Level Scheduling Rewrites". In: 6.POPL (2022). DOI: 10.1145/3498717 (cited on pages 45, 47).

[Mis+23] Edward Misback, Caleb C. Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. "Odyssey: An Interactive Workbench for Expert-Driven Floating-Point Expression Rewriting". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST '23. San Francisco, CA, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3586183.3606819 (cited on page 130).

[Moi24] Alexandre Moine. "Formal Verification of Heap Space Bounds under Garbage Collection". PhD thesis. Université Paris Cité, 2024 (cited on page 127).

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2016. DOI: 10.1007/978-3-662-49122-5_2 (cited on page 16).

[MSS17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2017, pp. 104–125 (cited on page 127).

[Nec98] George Ciprian Necula. "Compiling with proofs". PhD thesis. Carnegie Mellon University, 1998 (cited on page 19).

[OHe19] Peter W. O'Hearn. "Separation logic". In: *Commun. ACM* 62.2 (2019), pp. 86–95. DOI: 10.1145/3211968 (cited on page 23).

[Phi+14] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. "Software Verification with VeriFast: Industrial Case Studies". In: *Sci. Comput. Program.* 82 (Mar. 2014), pp. 77–97. DOI: 10.1016/j.scico.2013.01.006 (cited on page 16).

[Pin+20] Pedro Pinto, Joao Bispo, Joao Cardoso, Jorge Gomes Barbosa, Davide Gadioli, Gianluca Palermo, Jan Martinovic, Martin Golasowski, Katerina Slaninova, Radim Cmar, et al. "Pegasus: Performance Engineering for Software Applications Targeting HPC Systems". In: *IEEE Transactions on Software Engineering* (2020) (cited on page 45).

[Rag23] Jonathan Ragan-Kelley. "Technical Perspective: Reconsidering the Design of User-Schedulable Languages". In: *Commun. ACM* 66.3 (2023), p. 88. DOI: 10.1145/3580370 (cited on page 44).

[Rag24] Jonathan Ragan-Kelley. *The Future of Fast Code: Giving Hardware What It Wants*. Keynote at Programming Language Design and Implementation (PLDI). June 2024. URL: https://www.youtube.com/live/66oKqvwoIv0 (cited on page 8).

[Rag+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Conference on Programming Language Design and Implementation*. 2013. DOI: 10.1145/2491956.2462176 (cited on pages 9, 44).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 12, 71).

[Sak+22] Ömer Sakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. "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*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13244. Lecture Notes in Computer Science. Springer, 2022, pp. 332–352. DOI: 10.1007/978-3-030-99527-0_18 (cited on pages 19, 45, 125).

[Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: automating the foundational verification of C code with refined ownership types". In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036 (cited on pages 16, 128).

[SB14] Julian Shun and Guy E. Blelloch. "Phase-concurrent hash tables for determinism". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '14. Prague, Czech Republic: Association for Computing Machinery, 2014, pp. 96–107. DOI: 10.1145/2612669.2612687 (cited on page 127).

[Sil+19] Cristina Silvano et al. "The ANTAREX domain specific language for high performance computing". In: *Microprocessors and Microsystems* 68 (2019), pp. 58–73. DOI: https://doi.org/10.1016/j.micpro.2019.05.005 (cited on page 45).

[SK24] Thomas Somers and Robbert Krebbers. "Verified Lock-Free Session Channels with Linking". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689732 (cited on page 127).

[Spi+24] Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. "Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq". In: *Proc. ACM Program. Lang.* 8.PLDI (2024). DOI: 10.1145/3656413 (cited on page 128).

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. "From program verification to program synthesis". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 313–326. DOI: 10.1145/1706299.1706337 (cited on page 19).

[SCS24] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. "Interactive Abstract Interpretation with Demanded Summarization". In: *ACM Trans. Program. Lang. Syst.* 46.1 (Mar. 2024). DOI: 10.1145/3648441 (cited on page 133).

[TL08] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: a case study on instruction scheduling optimizations". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444 (cited on page 18).

[Vac+03] Manish Vachharajani, Neil Vachharajani, David I. August, and Spyridon Triantafyllis. “Compiler Optimization-Space Exploration”. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Los Alamitos, CA, USA: IEEE Computer Society, 2003, p. 204. DOI: 10.1109/CGO.2003.1191546 (cited on page 8).

[Xia+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. “Interaction trees: representing recursive and impure programs in Coq”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371119 (cited on page 124).

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Conference on Programming Language Design and Implementation*. San Jose, California, USA: Association for Computing Machinery, 2011. DOI: 10.1145/1993498.1993532 (cited on page 18).

[ZHB18] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. “Visual Program Manipulation in the Polyhedral Model”. In: *ACM Trans. Archit. Code Optim.* 15.1 (2018). DOI: 10.1145/3177961 (cited on page 45).

Université de Strasbourg

Guillaume Bertholon

École doctorale Mathématiques, sciences de l'information et de l'ingénieur | ED 269
Université de Strasbourg

Interactive compilation via trustworthy source-to-source transformations

Compilation interactive par des transformations source-à-source dignes de confiance

Résumé

Les programmeurs sont confrontés à deux facteurs limitants : leur capacité à produire du code sans bogues et la puissance de calcul de leur matériel. L'optimisation manuelle du code peut repousser les limites du matériel, mais elle est chronophage, car elle augmente la taille du code et le risque de bogues.
Cette thèse vise à réduire le travail nécessaire pour produire un code optimisé et exempt de bogues. Pour cela, nous avons développé le compilateur interactif OptiTrust qui applique des transformations source-à-source guidées par l'utilisateur.
Pour assurer qu'aucune transformation ne crée de bogue, OptiTrust exploite des annotations en logique de séparation initialement fournies par l'utilisateur puis mises à jour automatiquement à chaque étape. Ces annotations peuvent encoder soit une preuve de correction fonctionnelle préservée par les transformations, soit la structure de la mémoire accessible qui permet de vérifier que les transformations préservent la sémantique du code.

Mots-clés : Code haute performance, Preuve de programme, Optimisation source-à-source, Compilation, Logique de séparation

Abstract

Programmers are usually facing two limiting factors: their ability to produce code without bugs, and the computing power of their hardware. User-driven code optimization can push the hardware limits further, but it can be time-consuming as it usually increases the code complexity and makes bugs more likely.
This thesis aims at reducing the amount of work needed to produce code that is at the same time optimized and exempt from bugs. To achieve this goal, we developed an interactive compiler called OptiTrust, which applies user-guided source-to-source transformations.
To guarantee that each transformation does not create a bug, OptiTrust leverages separation logic resource annotations initially written by the user and updated automatically at each step. These annotations can encode either a functional correctness proof, which is preserved through transformations, or the shape of the accessible mutable memory, which can be used to check that transformations preserve code semantics.

Keywords: High performance code, Software verification, Source-to-source optimization, Compilation, Separation logic