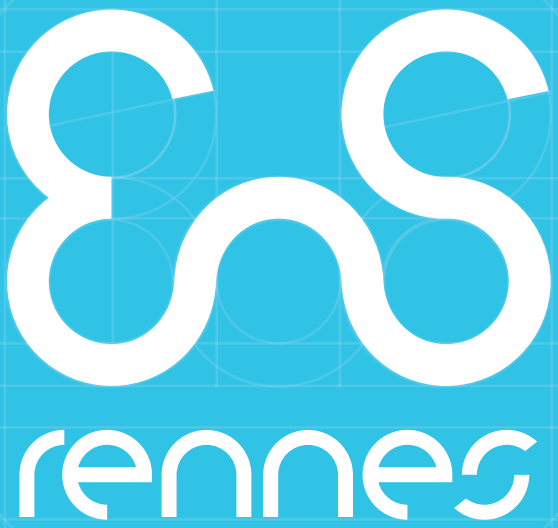
UNIVERSITE BRETAGNE LOIRE



THÈSE / ENS RENNES
Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES
Mention : Informatique
École doctorale MathSTIC

présentée par
Yannick Zakowski
Préparée à l'unité mixte de recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Verification of a concurrent garbage collector

Thèse soutenue le 19 décembre 2017
devant le jury composé de :

Marieke Huisman / *rapporteur*
Professeure, Université de Twente (Pays-Bas)

Stephan Merz / *rapporteur*
Directeur de recherche, Inria Nancy

Daniel Hirschkoff / *examinateur*
Maître de Conférences, ENS de Lyon

Francesco Zappa Nardelli / *examinateur*
Directeur de recherche, Inria Paris

Olivier Ridoux / *examinateur*
Professeur des universités, Université de Rennes 1

David Cachera / *encadrant*
Maître de conférences, ENS Rennes

David Pichardie / *encadrant*
Professeur des universités, ENS Rennes

Résumé

Les compilateurs modernes constituent des programmes complexes, réalisant de nombreuses optimisations afin d'améliorer la performance du code généré. Du fait de cette complexité, des bugs y sont régulièrement détecté, conduisant à l'introduction de nouveau comportement dans le programme compilé.

En réaction, il est aujourd'hui possible de prouver correct, dans des assistants de preuve tels que Coq, des compilateurs optimisants pour des langages tels que le C ou ML. Transporter un tel résultat pour des langages haut-niveau tels que Java est néanmoins encore hors de portée de l'état de l'art. Ceux-ci possèdent en effet deux caractéristiques essentielles: la concurrence et un environnement d'exécution particulièrement complexe.

Nous proposons dans cette thèse de réduire la distance vers la conception d'un tel compilateur vérifié en nous concentrant plus spécifiquement sur la preuve de correction d'un glaneur de cellules concurrent performant. Ce composant de l'environnement d'exécution prend soin de collecter de manière automatique la mémoire, en lieu et place du programmeur. Afin de ne pas générer un ralentissement trop élevé à l'exécution, le glaneur de cellules doit être extrêmement performant. Plus spécifiquement, l'algorithme considéré est dit «au vol»: grâce à l'usage de concurrence très fine, il ne cause jamais d'attente active au sein d'un fil utilisateur. La preuve de correction établit par conséquent que malgré l'intrication complexe des fils utilisateurs et du collecteur, ce dernier ne collecte jamais une cellule encore accessible par les premiers.

Nous présentons dans un premier temps l'algorithme considéré et sa formalisation en Coq dans une représentation intermédiaire conçue pour l'occasion. Nous introduisons le système de preuve que nous avons employé, une variante issue de la logique «Rely-Guarantee», et prouvons l'algorithme correct.

Raisonner simultanément sur l'algorithme de collection et sur l'implantation des structures de données concurrentes manipulées générerait une complexité additionnelle indésirable. Nous considérons donc durant la preuve des opérations abstraites: elles ont lieu instantanément. Pour légitimer cette simplification, nous introduisons une méthode inspirée par les travaux de Vafeiadis et permettant la preuve de raffinement de structures de données concurrentes dites «linéarisables». Nous formalisons l'approche en Coq et la dotons de solides fondations sémantiques. Cette méthode est finalement instanciée sur la principale structure de données utilisée par le glaneur de cellules.

Abstract

Modern compilers are complex programs, performing several heuristic-based optimisations. As such, and despite extensive testing, they may contain bugs leading to the introduction of new behaviours in the compiled program.

To address this issue, we are nowadays able to prove correct, in proof assistants such as Coq, optimising compilers for languages such as C or ML. To date, a similar result for high- level languages such as Java nonetheless remain out of reach. Such languages indeed possess two essential characteristics: concurrency and a particularly complex runtime.

This thesis aims at reducing the gap toward the implementation of such a verified compiler. To do so, we focus more specifically on a state-of-the-art concurrent garbage collector. This component of the runtime takes care of automatically reclaiming memory during the execution to remove this burden from the developer side. In order to keep the induced overhead as low as possible, the garbage collector needs to be extremely efficient. More specifically, the algorithm considered is said to be "on the fly": by relying on fine-grained concurrency, the user-threads are never caused to actively wait. The key property we establish is the functional correctness of this garbage collector, i.e. that a cell that a user thread may still access is never reclaimed.

We present in a first phase the algorithm considered and its formalisation in Coq by implementing it in a dedicated intermediate representation. We introduce the proof system we used to conduct the proof, a variant based on the well- established Rely-Guarantee logic, and prove the algorithm correct. Reasoning simultaneously over both the garbage collection algorithm itself and the implementation of the concurrent data-structures it uses would entail an undesired additional complexity. The proof is therefore conducted with respect to abstract operations: they take place instantaneously. To justify this simplification, we introduce in a second phase a methodology inspired by the work of Vafeiadis and dedicated to the proof of observational refinement for so-called "linearisable" concurrent data-structures. We provide the approach with solid semantic foundations, formalised in Coq. This methodology is instantiated to soundly implement the main data-structure used in our garbage collector.

VERIFICATION OF A CONCURRENT GARBAGE COLLECTOR

YANNICK ZAKOWSKI

December 2017

CONTENTS

LIST OF FIGURES

LISTINGS

1

INTRODUCTION

As years pass by, a recurrent assessment becomes ever strikingly more undeniable: software is pervasive. It spread over the decades through all aspects of our life, until reaching a new symbolic bastion: even our fridges (!) nowadays contain software. This evolution has brought numerous enhancements to our quality of life, from recreational activities to medical advances, as well as to scientific progress. But it should not be pondered without keeping in mind another reality: software systems are extremely complex objects. As a consequence, *bugs* are no less pervasive: programs occasionally exhibit unexpected behaviours. A bug can embody itself in different kinds. It can result in a functional error, for which the program outputs an incorrect result, failing to fulfil its specification. The program can also simply fail to output a result, suffering from a runtime error, i. e. a crash. Finally, a bug can also more subtly be functionally correct, but produce intermediate states that allow a malicious user to temper with the system's integrity.

Much like in any industry, the consequences of such defects in the products depend on the context of use. They might be essentially harmless, a mere annoyance, in benign contexts; they may trigger a restart of a phone application for instance. In contexts identified as *safety-critical* however, bugs can be the cause of significant costs, including human losses. This category of software includes most famously aircraft flight controllers, automated driverless subways, medical software and soon driverless cars. Both the manufacturers of these applications and the public institutions controlling the quality of such life-threatening software are well aware of the need for a high level of confidence in their execution. Any kind of bug would result in a potential accident, or a vulnerability allowing for a criminal attack.

Despite this awareness, bugs in critical systems do occur, sometimes preemptively detected by experts, most of the time following an accident. In 2008, a vulnerability allowing for denial-of-service attacks is found in pacemakers by Halperin et al. [49]: one could theoretically remotely stop someone's pacemaker. Spacecraft suffered along the years particularly costly losses due to software. In 1996, thoughtless reuse of a module from Ariane 4 in a different environment led the European rocket Ariane 5 to blow up [37]. In 2016, the Japanese satellite Hitomi is lost as a consequence of a glitch in its star tracker [144]. Gloomier, multiple fatal accidents between 2002 and 2009 have been directly linked to uncontrollable accelerations of Toyota vehicles [124].

FORMAL METHODS

Scientists have acknowledged and sought to address the challenge since the early days of computer science, leading to many approaches. The most natural reflex is to mirror other disciplines: programs are

rigorously tested. While irreplaceable, the art of testing proves itself insufficient. Indeed, the set of inputs and contexts with which a program can be executed is usually infinite, preventing testing on its own to ever rule out the existence of a bug in program. In contrast, *formal methods* are a set of scientific techniques aiming at proving, in the traditional mathematical sense, that a software system indeed fulfils its specification. The approach consists in choosing a mathematical model to abstract the system, expressing its specification in a formal language, and proving the adequacy of both components through mathematical techniques. Crucially, the process does not require the actual *execution* of the program, and is able to prove the absence of bug for *any* run of the program. Depending on the theoretical tools used to model the system and verify it, such *static* techniques can notably rest upon model-checking, abstract interpretation or program logics.

While some of the ideas from formal methods have continuously percolated towards industry, it was unclear for a long time whether these techniques could ever scale to real-life challenges. Formal software verification has nonetheless become increasingly popular, notably following major successes in safety-critical embedded software for the avionic. Astrée [14], one of the most influential abstract interpreters, was able in 2003 to prove the absence of a certain class of bugs in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C code. Caveat [12], another static analyser developed by the French nuclear agency (CEA), has been similarly used at Airbus. Nowadays, companies such as Facebook [39] or Amazon [102] have integrated formal methods as part of their production cycle.

MANAGED LANGUAGES

These different formal methods share a common pattern: they tackle the problem by considering the end product, the program itself. A complementary approach is to change the tools used to make these programs by improving the design of programming languages themselves. An emblematic case is the use of *automatic memory management* [65], nowadays featured by a majority of modern languages. A entire class of memory-management related bugs, namely double-free and use after free errors, are ruled out by construction from any program written in a language supporting this feature. High-level languages such as Java additionally rely on typing disciplines to rule out additional classes of bugs either statically – such as the use of an operator on an inadequate data-structure – or dynamically, substituting dangerous behaviours by interruptions of the execution – such as an access to an array out of its bound.

In order to enforce such safety properties, the compiler of those languages injects a runtime system, or simply runtime, into the client code. This notion is usually loosely defined as any behaviour of the execution of a program which is not directly attributable to the program itself. The term may therefore encompass a wild range of notions, from the execution model of parallel executions to runtime checks enforcing safety properties at execution time, most notably in dynamically typed languages. In the limit, the runtime may design the virtual machine upon which the bytecode is executed, such as is the case in Java for instance. In this work, we refer specifically to as runtime the pieces of code that the compiler injects in order to provide services to programs during their execution. Examples may include the injection of array bounds checks, of type checking assertions or of a garbage collector.

Contrary to static analyses, the use of a runtime naturally entails a cost over the code size, execution time or memory footprint. In order to mitigate this issue, implementations of runtime are therefore strongly optimised, resulting in subtle code, likely to be written in a low-level language for efficiency. This assessment is even strengthened when one wants to use these techniques in safety-critical contexts, where real-time constraints intervene. We are hence pulled a step back: implementations of runtime, and compilers as a whole, are extremely complex programs, hence likely to be bugged. Worse, their bugs may contaminate any program written in the compiled language. Indeed, whether the source program satisfies its specification is not the endgame, the relevant question is truly whether the compiled, executable machine code.

Compilers are therefore in this sense the most important software systems to statically verify in order to obtain a complete confidence chain from the specification of the source program to the executable effectively run.

VERIFIED COMPILATION

Modern compilers may be among the most complex entities humans have built to date. The most widely used C compiler, gcc, currently amounts for more than seven million lines of code, spread across more than seventy thousands file. Such compilers perform complex static analyses in order to optimise the compiled code, and the correctness of these analyses is extremely difficult to ensure, especially when combined together. They sometimes even openly accept incorrectness over corner cases to enable aggressive optimisations, such as with the `-Ofast` option from the gcc compiler. For a long time, the impossibility to combine safety and optimising compilers has been accepted as inescapable. Empirical studies – such as the NULLSTONE tool [100] or Eide and Regehr's study on volatiles [38] – kept finding

miscompilation issues in state-of-the-art compilers. This situation led engineers working with critical systems, such as in avionics, to simply give up on optimisation altogether.

For a long time, researchers have sought to address the situation by designing a formally verified compiler. All approaches share a common goal. First, to provide a rigorous meaning to the behaviour of the source and target languages manipulated by the compiler, namely their semantics. Second, to establish that the various transformations the compiler performs over the program preserve this semantics. In the sixties, McCarthy and Painter initiated the field by proving the correctness of a compiler for arithmetic expressions [93]. In 1973, Morris [98] refined the approach by introducing a simulation-based methodology for proving the correctness of real-sized compilers: one shows that steps of computations at the target language can always be mimicked at the source language.

MACHINE-CHECKED VERIFIED COMPILATION

Entering the eighties, we had the essential theoretical tools – semantics of languages and simulations – to prove a real-size compiler. However, manual proofs can not completely fit the bill. First, programs are complicated objects, whose semantics can interact in various ways with the compiler's transformations. A proof of correctness of a compiler therefore contains both subtle algorithmic arguments, as well as a significant amount of details. The task is therefore extremely error prone, and difficult to review. Second, manual proofs are much too likely to abstract away from concrete details. Indeed, there is always a gap between an algorithm and its implementation. The use of proof assistants, such as Coq [23] or Isabelle [61], addresses both issues. A program takes the responsibility for reviewing the proof, granting complete trust to the proof of correctness. This trust naturally relies on the confidence we have in the proof assistant itself. However, these programs are usually built in such a way that they include a small, trusted kernel whose size and complexity remains manageable. If the kernel is correct, then the bugs in the other parts of the program are harmless to the validity of the proofs. Second, these proof assistants provide a unified logical framework to write programs and their specification, as well as to prove these specifications. They additionally provide an *extraction mechanism* which generates from the formal development the fragment responsible for processing the data: the mechanism erases all specifications and proofs, and leaves the user with the resulting verified program. We therefore have the possibility to prove the code we run, and not only the algorithm we implement.

Following this insight, the first machine-assisted proof of a compiler for a high-level assembly language was performed in the nineties by Moore [96, 97]. Since then, steady progress has been made to

both verify increasingly more realistic compilers, and to scale towards high-level languages. On the first front, the C language has been the main object of study. The Verisoft project [80] tackled in 2005 the compilation of Co, a type-safe subset of C. Four years later, the CompCert C compiler by Leroy et al. [22, 71, 82] has been the first to reach industrial standards. CompCert is a fully verified, optimising compiler for a very large subset of the ISO C 99 standard, down to the assembly language for the x86, PowerPC and ARM architectures. While it only allows for carefully chosen optimisations, the produced code is at least twice as fast as the one produced by gcc with no optimisation (`gcc -O0`). The additional trust in the compiler brought by the verification has been empirically confirmed. In particular did CompCert manage to resist recent empirical attempts to find miscompilation bugs in compilers, as described by Yang et al. [147]. This unprecedented trust in an optimising compiler led Airbus to integrate its use into their development process.

Since then, the field has been extremely active. CompCert has been adapted by Ševčík et al. [130] to extend the supported C language with concurrency primitives for thread management and synchronisation. In 2010, Chlipala [17] formalised in Coq a compiler from a small, untyped functional language with mutable references and exceptions to an idealised assembly language. The proofs are designed in a highly modular and automated way, allowing for proofs to withstand extensions of the language. The Vellvm project [150] tackles, also in Coq, the proof of correctness of components of the LLVM compiler [78, 133]. On recent years, the CertiCoq [8] project aims to build a proven-correct compiler for dependently-typed, functional languages – such as Gallina, the core language of the Coq proof assistant – rising new foundational questions about compilation of dependently typed languages. Finally, the CakeML compiler [74, 132] is an optimising, verified in HOL4, compiler for the ML language. In particular, the compiler therefore includes a verified runtime containing a sequential garbage collector.

MACHINE-CHECKED VERIFIED COMPILATION FOR CONCURRENT, MANAGED LANGUAGES

Despite all these striking achievements, verified compilation is not yet mature enough to handle the compilation of a high-level language such as Java. Beyond the sheer complexity of the language, two major features raise a challenge: concurrency and runtime management. Figure 1 underlines the gap in complexity these additional feature represent. The source code of the `OpenJDK Hotspot Java Virtual Machine` [112] is represented by blocks scaled proportionally to the size of the corresponding files. We observe that the runtime, and in particular the memory and multi-threading management, constitutes

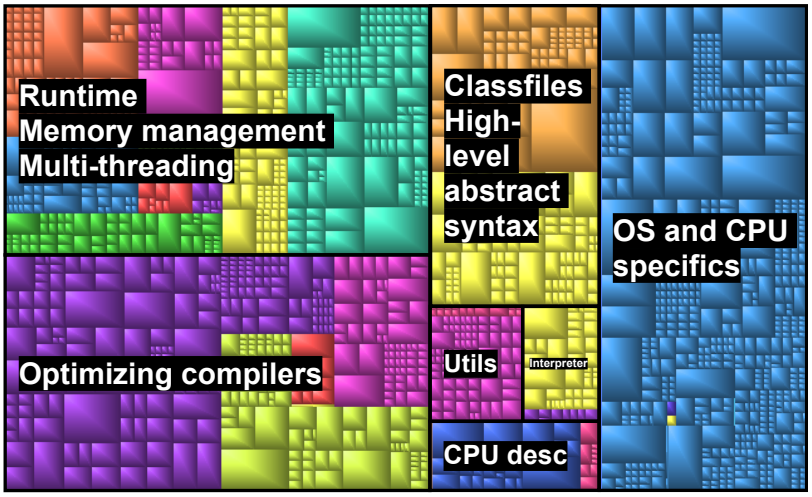


Figure 1: Tree map of Java's HotSpot source code (~900 kloc). Each file is represented by a block whose size is scaled with the length of the file, and colour determined by its nature.

roughly a quarter of the total source code. Once again, being performance critical, it contains subtle, highly concurrent pieces of code.

Handling this multi-threaded runtime is a complex task. Indeed, semantics for concurrent programs are much more complex to reason about than their sequential counterpart. First, a concurrent language is inherently non-deterministic, leading to an exponentially more complex set of reachable states. Additionally, we lose the benefit of determinism when building simulations, which CompCert takes a special care to exploit. In such a context[1], to build the desired *backward* simulation, where the source mirrors the behaviours of the target, it suffices to build a *forward* simulation, where the roles are reversed. Second, local reasoning is a lot harder to conduct in a concurrent setting. The central problem is that a property established at a program point of a thread's code could always be invalidated by another thread's interference. Developing adequate reasoning principles and program logic to reason about concurrent programs has been a major topic of research during the last thirty years [113, 123].

CompCertTSO [130] offered a first foundational contribution to the problem by managing to adapt CompCert to a realistic concurrent execution model, the so-called *Total Store Ordering* (TSO) memory model, and verify some fence-elimination optimisations. While a major step, their compiler is carefully and cleverly designed as to i) retrieve locally some notions of determinism to get away with forward simulations, ii) avoid having to perform fine-grained reasoning about concurrent programs. In order to support a high-level managed language, we however cannot rest on those luxuries. The compiler does

1 The semantics is additionally required to satisfy a notion of receptiveness. If a state can step by emitting an event corresponding to a read or write to the outside world, then it can also step for any other value similarly read or written.

not only transform the client code, but also injects concurrently running services, such as garbage collectors, monitors, or schedulers. A verified compiler for a language such as Java therefore requires to prove correct the complex, highly concurrent, runtime implementations.

CONTRIBUTIONS AND STRUCTURE OF THE DOCUMENT

In this thesis, we show how to progress one step closer towards a verified compiler for a high level, managed, concurrent programming language such as Java. In particular, we consider the emblematic challenge of an *on-the-fly* garbage collector and demonstrate how to formally verify its implementation in the Coq proof assistant. In agreement with the verified compilation tradition, this work puts a strong emphasis on proofs performed with respect to an operational semantics, and results expressed in terms of simulations.

We argue that a methodological approach to this task is mandatory. Proving such a concurrent service is indeed a complex puzzle. Extremely subtle invariants must be designed in harmony with sequential proof annotations such that no interference can perturb the validity of any of the assertions. Due to this interdependence, separation of concerns is crucial. To this end, we introduce several solutions.

- We design a dedicated intermediate representation RTIR. The language is at the right level of abstraction to both be sufficiently concrete to allow subsequent compilation to executable code, while being sufficiently high-level to allow for a tractable proof of the garbage collector. To this end, abstract concurrent data-structures are embedded in the semantics of the language.

- We formalise a *Rely-Guarantee* logic tuned to ease mechanisation of proofs. Our proof system indeed offers a better separation of concerns between sequential proofs and stability obligations than traditional ones, and supports partial automation.

- We introduce an iterative development process to the construction of rely-guarantee proofs, and follow this process to prove the garbage collector.

- We formalise a rely-guarantee-based methodology for atomic refinement of linearisable data-structures. Our approach does not require the introduction of a new program logic and is equipped with strong semantic foundations compatible with the framework of verified compilation.

Leveraging these principles, our first major contribution is the formalisation and proof of correctness, in Coq, of a realistic on-the-fly garbage collector inspired from the algorithm for Java by Domani et

al. [35]. We prove that the collector never reclaims an object reachable by a client thread, ruling out dangling pointers, a result whose core description is contained in Chapter 5. This work is published in the proceedings of the 8th *International Conference on Interactive Proving* (ITP'17) [148] and its formal development is available online[2].

To support the claim that RTIR is designed at the right level of abstraction, we show how to soundly implement the abstract data-structures the language supports. This work constitutes our second major contribution, described in Chapter 6. We formalise, in Coq, a light-weight methodology to refine *linearisable* fine-grained data-structures, an instrumental step for the compilation of RTIR towards an executable language. The approach reduces the problem to proof obligations expressed in a rely-guarantee logic, and provides strong semantic foundations compatible with the framework of verified compilation. This work is published in the proceedings of the 33rd Symposium on Applied Computing, in the Software Verification and Testing track (SAC-SVT'18) [149], and its formal development is available online[3].

The remainder of this thesis is organised as follows. Chapter 2 provides general background on memory management, giving an overview of the ecosystem into which the particular garbage collection algorithm we consider fits. Chapter 3 describes the formal techniques we use and formalise. After a brief presentation of the Coq proof assistant, we describe RTIR, the programming language we use to implement the garbage collector, and conclude by presenting rely-guarantee reasoning and the specific proof system we define. Chapter 4 and Chapter 5 respectively describe the algorithm and its formal proof. Finally, Chapter 6 provides context on linearisability and exposes the meta-theorem we proved, as well as its application to refine the abstract mark buffers manipulated by RTIR, solving the core challenge towards its compilation down to a fully executable language.

NOTE

This thesis work is tied to a project to which have contributed, in alphabetical order: David Cachera, Delphine Demange, Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, Jan Vitek and Yannick Zakowski.

My personal technical contributions are:

- the design of the incremental approach, the proof of the underlying meta-theory over the rely-guarantee system and its mechanisation;
- the complete mechanised proof of the garbage collector;

2 http://www.irisa.fr/celtique/ext/cgc/
3 http://www.irisa.fr/celtique/ext/simulin/

- the formulation of the methodology for refinement of linearisable data-structures, its mechanisation and its proof of soundness.

2

BACKGROUND ON MEMORY MANAGEMENT

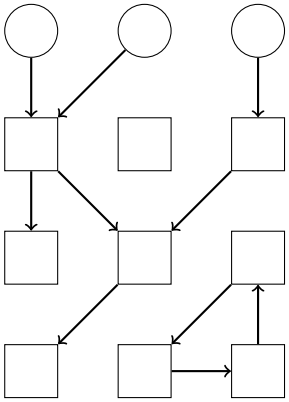


Figure 2: An abstraction of the view of the memory commonly used garbage collectors. The underlying array of bites, or single-linked list of cells, is abstracted away by accessing the memory through an allocator and a free primitive.

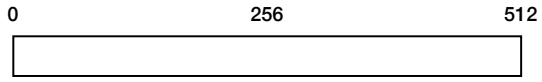
Computer science is intrinsically concerned with encoding and manipulating data. A programming language is therefore meant in particular to specify how and when those data should be *allocated* in memory. This memory being finite in practice, someone has to also inherit the burden of specifying when a chunk of memory dedicated to encode a piece of data can be safely reclaimed, and therefore reused.

This concern, known as *memory management*, is essentially orthogonal to the logic of the program one can write. Nonetheless, it turns out to be often subtle to handle. We review in this chapter some of the existing approaches used to tackle memory management, setting the stage for the specific algorithm of *garbage collection* we formalised during our thesis, whose description and formalisation are covered respectively in Chapters 4 and 5.

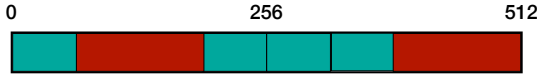
2.1 ALLOCATION

In essence, memory is a mere enormous array of bytes. The low level abstraction that is usually built upon is to interpret this array as a sequence of *cells*. A cell is characterised by its location, the address of its beginning in the underlying array, the value stored in the cell, and sometimes the size of the cell.

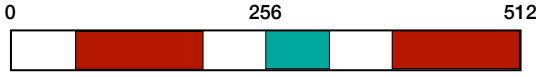
Whether we strive for manual or automatic memory management, we do not want the programmer to have to worry about which concrete location is used to store a piece of data. An *allocator* is a primitive taking up this responsibility: upon a request to allocate some data, the allocator takes care of organising concretely the memory, and return to the user an address at which he can access its data. Similarly, a *free* primitive takes care of reclaiming a chunk of memory. This mechanism allows for higher level abstractions of the memory.



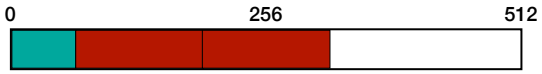
(a) Initial empty 512 bytes of memory.



(b) Ongoing run, the memory is full.



(c) After naïve reclamation, the memory is fragmented: we cannot allocate a new block of 128 bytes.



(d) The same memory, defragmented.

Figure 3: Illustration of the problem of fragmentation over a 512 bytes memory. We assume a program allocating objects of 64 and 128 bytes.

Notably, we shall see that a garbage collector for a language with objects typically interpret the memory as a graph such as the one depicted on Figure 2. Round nodes represent addresses stored in variables, the *roots* of the program, while square nodes represent the *cells*. The object held in the cell points to other cells through its fields, which are represented as the arrows in the graph. We shall come back in greater detail to this model in Chapter 4, in the specific context of RTIR, a language we introduce in Chapter 3.

Being able to abstract away from the details of the memory when reasoning about the correctness of a garbage collector's implementation is crucial. However it should be underlined that for performance considerations, we can never completely decorrelate the allocator and the collector choices of algorithms. In particular, the memory is likely to suffer from a problem known as *fragmentation*. Figure 3 depicts an imaginary run of a program equipped with a 512 bytes of memory, initially empty (Fig. 3a), and allocating objects of respectively 64 or 128 bytes. After some time, the program tries to allocate a new 128 bytes objects, but finds that the memory is full (Fig. 3b): a cycle of collection is performed. If neither the allocator nor the collector is clever about the structure of the memory, we may end up in a situation where allocated blocks of memory are scattered (Fig. 3c). Despite admitting more than 128 bytes of free memory, no contiguous such block can be found: the allocation still fails, where its defragmented equivalent would allow it (Fig. 3d). We tend to put the responsibility of avoiding fragmentation of the memory over the allocator, but an efficient strategy may depend on the behaviour of the collector.

2.2 MANUAL MEMORY MANAGEMENT

Memory management is crucially concerned with one property: the *lifetime* of a piece of memory the programmer wishes to allocate. This lifetime refers to the instant, at runtime, during which memory is first allocated, and then reclaimed. Languages supporting manual memory management, such as C or C++, usually allow for three kinds of allocation.

Under its simplest form, the lifetime of the allocated memory is the whole program: memory is said to be *statically allocated*. The programmer knows exactly at compile time the objects that may be used by the program at runtime. Under such an assumption, memory management is straightforward: we simply map, at runtime and once and for all, each object to a memory region of appropriate size. Naturally, static allocation puts such constraints on the programmer that its use is quite rare in modern programming languages. Global variables in C are nonetheless one example of statically allocated resources.

The second form of allocation is *automatic memory allocation*. The programmer does not have control over the lifetime of the data: the lifetime is bound to a syntactic program point. Typically, this happens in C in the case of a non-static variable declared inside a function body: the variable is stored on the stack, and reclaimed when the function returns.

The last form of allocation is *dynamic memory allocation*. The lifetime of the object is this time entirely handled by the programmer. An allocating primitive allows for a chunk of memory of the desired size to be allocated. Symmetrically, a freeing primitive allows for a specified piece of memory to be reclaimed. This mechanism brings two crucial conveniences to the programmer: both the lifetime and the size of the allocated object can be dynamically controlled. In particular does it allow for closures and recursive data-structures to be manipulated. While crucial, dynamic memory allocation in manually managed programming languages dooms us with two plagues. First, one could fail to free some dynamically allocated memory beyond the bounds of its use. Such a phenomenon, referred to as a *memory leak*, can lead the program to failure by running out of memory. Second, the inverse phenomenon may happen: a chunk of the memory could be reclaimed despite being still accessible to through a pointer. Such a pointer is said to be *dangling*, and dereferencing it is no less disastrous, leading to a crash in the best case, a nonsensical value in the worse.

Note that both problems are inherently difficult in the sense that checking their absence is a non-computable problem [142]. The practical difficulty to manage memory manually is up to (sometimes heated) debate. Nonetheless, a significant portion of the programming community has deemed this art as unsafe and error-prone,

and hence except for cases of utmost necessity, such as system programming. In order to retain the benefits of dynamic memory allocation while circumventing its dangers, most modern programming languages have therefore turned to automatic memory management.

2.3 GARBAGE COLLECTION

The most popular approach for automatic management is probably *garbage collection*. The programmer does not have to worry about managing the memory: a runtime is injected, usually by the compiler, whose sole purpose is to monitor the memory and automatically reclaim data once it is safe to do so. The invention of garbage collection is attributed to John McCarthy, around 1959, for the Lisp language [90].

Garbage collection has been an extremely active domain of research ever since. We provide in this section a concise overview of the field, and refer the interested reader to *The Garbage Collection Handbook* [65] for a more thorough exposition.

All approaches essentially reduce to one of four different collecting schemes: Mark and Sweep, Mark and Compact, Copying and Reference Counting. In practice, realistic programming languages often combine several of these approaches. They may typically treat two regions of the heap differently. We stay at a more schematic level in this presentation, reviewing the different approaches in their purest form. Orthogonal to the choice of an approach, garbage collection can be optimised via notably two enhancements: use of generations and use of concurrency.

2.3.1 *Fundamental collecting schemes*

While refined implementations are legions, all garbage collector algorithms can be summed up as following one of four collecting schemes. The simplest setup for their implementation naturally takes place in a mono-threaded environment: the program client is purely sequential, and the collector is injected inside this very same thread. We assume this context to expose the principles behind the different approaches, but all of those can be adapted to handle concurrency.

2.3.1.1 *Mark-and-Sweep garbage collection*

Garbage collectors usually trigger a collecting cycle when a call to the allocator detects an exhaustion of available memory. This cycle is a routine which typically inspects the memory, consequently reclaims objects detected as safe to do so, and potentially reorganise the memory.

To this end, garbage collectors of the Mark and Sweep family, whose seminal McCarthy's algorithm [90] is an example, maintain at any given moment a view of the memory as split in two sets O and F which do not intersect. The objects in F are known to be free, and hence ready to be allocated if needed, while the objects in O are believed to be *live*, i. e. reachable by the program.

Periodically, the collector stops the user program and initiates a cycle to update its knowledge of the current state of the memory, allowing him to refine its set O. To do so, the collector needs to determine which objects are live. First, the objects whose addresses are directly hold in local variables, i. e. the roots, are marked. Second, the collector scans the memory: following the graph memory abstraction, a breadth-first exploration is performed to mark all reachable objects as live. Finally, the remaining objects are reclaimed, being transferred from the set O to the set F.

The GC we formalise and prove correct in this document is a sophisticated, concurrent, implementation of an algorithm of the Mark and Sweep family. We come back to its description in detail in Chapter 4, and describe its verification in Chapter 5.

2.3.1.2 *Mark-and-Compact garbage collection*

The Mark and Sweep scheme does not attempt to reorganise the memory. While the underlying allocator may try to handle the burden itself, the heap is likely to end up fragmented when used in long running applications. Finding free chunks of memory can therefore become slow, leading up to slower allocation.

The Mark and Compact scheme, whose early introduction includes Edward's *two fingers* algorithm [127] and Jonkers's *threaded compaction* algorithm [67], is a strategy aiming at addressing this difficulty. Similarly to the Mark and Sweep approach, a collecting cycle, organised in phases, is periodically triggered. The first step remains the same: the accessible memory subgraph is marked. Subsequently, the concrete layout of the heap is reorganised, following one of three politics. The reorganisation can be simply *arbitrary*, aiming only at compaction without regards for neither the previous relative positioning of objects, nor the high level representation of the memory in terms of a graph. Compaction can also be *linearising*[1], trying to pack together related objects in the memory graph. Finally, *sliding* collectors compact the memory while maintaining the relative order of objects.

The sliding strategy is the most commonly used. While more costly than arbitrary compaction, it offers several advantages. First and foremost, locality is preserved as objects allocated by the same user thread at around the same period of time remains close to each others, which

1 The term of *linearising* is used here with a completely different meaning than the one we consider in Chapter 6. This is purely coincidental.

tends to prove experimentally crucial for performance. Sliding also allows for easy reclamation of all data allocated after a certain point in time. With all strategies, compaction, by removing fragmentation, allows for very fast allocation: a pointer can simply be maintained at the beginning of the free part of the heap.

Determining whether the benefits of compaction overweight its cost is a delicate task, strongly depending on the targeted system. An overview of existing techniques and their analysis can be found in [66].

2.3.1.3 *Copying garbage collection*

Mark and Compact collectors offer a solution to the fragmentation of the heap, but at a cost: the live part of the heap needs to be scanned several times. Copying algorithms [16, 42] address this issue by compacting the memory while requiring only one pass. They however perform this at the significant cost of dividing the amount of memory available to the program by two.

Copying collectors split the memory in two regions of equal size, typically referred to as *tospace* and *fromspace*. As long as possible, allocation is performed at the top of *tospace*, which always remains compacted. Would this allocation be impossible for lack of place, collection is triggered. Memory in *tospace* is traversed, and live objects are contiguously copied in the fresh memory *fromspace*. Finally, *tospace* is entirely reclaimed, and the roles of both spaces are swapped.

Benefits and drawbacks of such a scheme are straightforward. Extremely efficient allocation is attained without the need for extra headers in objects, and using only one pass over the memory. Obviously, the space cost is however significant. Additionally, depending on the environment, the cost of copying can be too high, and locality can be delicate to maintain during the copy.

2.3.1.4 *Reference counting*

Reference counting is a garbage collection technique[2] where the user program directly handles creation and destruction of objects as it operates over them. The basic idea, dating back from the early sixties [21], consists in assigning to every object a counter, typically stored in the object's header. This counter is incremented, respectively decremented, every time a new reference to the object is created, respectively destroyed.

Detecting live objects therefore does not need any scan of the memory: at any given moment, an object is considered live if and only if its reference counter is strictly positive. An object whose counter

2 Whether reference counting deserves the name of garbage collection is debated. The quite legitimate rationale against this nomenclature is essentially due to the absence of a collector in the algorithm.

is set to zero can therefore immediately be reclaimed. Note that this operation leads to the decrement of all the object's children counter, which can in turn trigger new reclaims.

Reference counting has several potential benefits. First, the overhead is distributed along the computation rather than concentrated over a collecting cycle. Additionally, reclaim of memory tends to take place as soon as possible, without the need for waiting for a subsequent cycle. Finally, the algorithm remains viable when part of the system, and hence part of the memory graph, is unavailable. Its disadvantages include the need for an extra-header to objects, an overhead directly stressed upon the user threads and an impossibility to reclaim unreachable cycles. These issues can however be strongly mitigated with advanced improvements to the basic algorithm.

2.3.2 *Generational garbage collection*

Tracing collectors, i. e. all schemes introduced but reference counting, perform work proportional to the number of objects they process. Long live objects are therefore needlessly processed during numerous collection cycles, which is especially costly in the case of a copying scheme.

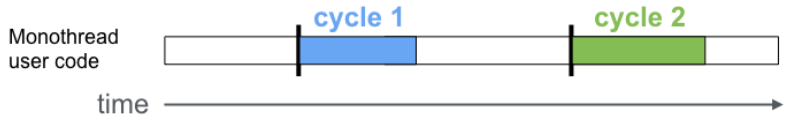
Generational collectors [88, 136] introduce a very practical optimisation in reaction to this phenomenon. Empirically, objects allocated long time ago are likely to still remain reachable for a long time. Conversely, many objects die very quickly. The idea behind generational garbage collection is therefore to organise the heap in generations: old objects, typically objects which have witnessed more than a collection cycle for instance, are tagged as such. Most collecting cycles then only trace the memory subgraph of young objects. Now and then, a special cycle scans the whole memory to find the few old dinosaurs which eventually died.

Choosing the amount of generations, the rhythm at which they are scanned and rate of decay is naturally a fine art, which once again depends heavily on the targeted environment.

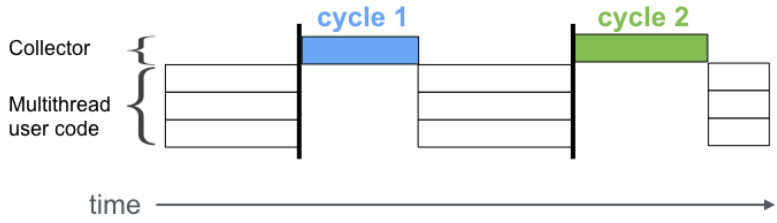
2.3.3 *Multi-threaded garbage collection*

Comparing the various approaches to garbage collection is an extremely complex task, whose conclusion is sensitive to the context of execution. Some modern languages, such as Java, actually ship in several garbage collectors and provide flags for the programmer to choose which to use depending on its use case.

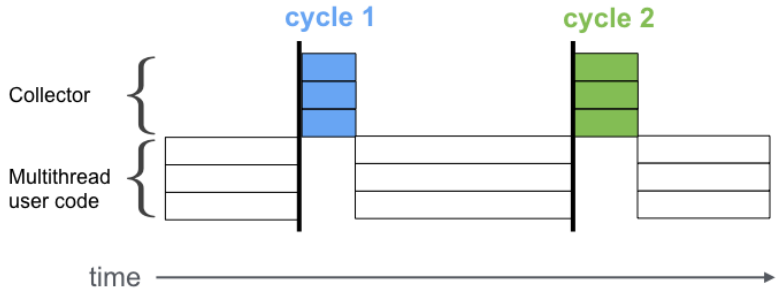
Nonetheless, the three tracing approaches share a common pitfall: they impose a significant overhead to the user program by stopping it completely while a full cycle is performed. Figure 4a illustrates



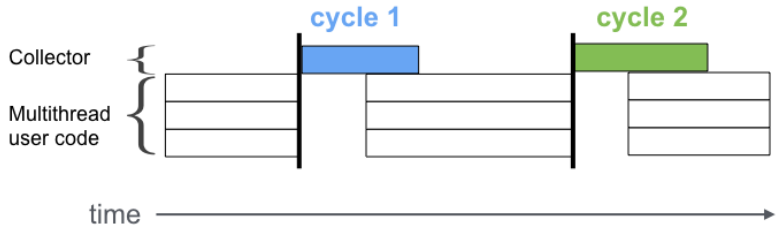
(a) Sequential garbage collection



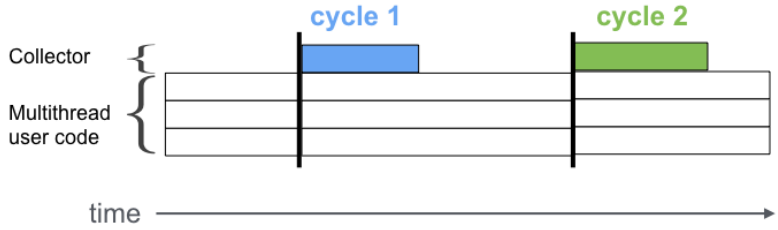
(b) Multi-threaded Stop-The-World garbage collection



(c) Parallel garbage collection



(d) Mostly concurrent garbage collection



(e) On-The-Fly garbage collection

Figure 4: Garbage collection: comparison of the user threads pause time with various approaches.

the situation over a timeline of an execution. A quite natural idea is therefore to exploit multi-threading in order to reduce this burden.

The starting point is to host the collector in a dedicated, separated thread. Coincidentally, we may now have several user threads. We easily understand that manipulating the memory while the collector tries to analyse and reclaim it might be dangerous. A first approach is therefore to keep asking to all user threads to interrupt their work during the collecting cycle. This approach, illustrated on Figure 4b, is referred to as Stop-The-World.

Naturally we immediately see that we do not benefit much from multi-threading with a simple Stop-The-World approach. One way to improve the situation is to parallelise the collection itself. Figure 4c shows three threads hosting the collector, speeding up the cycle and therefore reducing the waiting time of user threads despite maintaining a Stop-The-World policy.

An orthogonal idea to reduce the waiting time of user threads is to allow them to keep attending to their duties while the collector acts. A first, still conservative, approach continues to stop the user threads when the cycle begins. The collector can therefore easily setup things nicely, and allow the other threads to start over once it is ready, finishing its cycle as they are already back to work. This scenario is the one of Figure 4d.

The step towards the most sophisticated approach is now quite natural: so-called On-The-Fly garbage collectors, such as the one represented on Figure 4e, *never* interrupt the user threads. Naturally, this means that the user threads are able to modify the memory dynamically as the collector tries to perform its duty. In such a context, and in the remainder of this manuscript, user threads are hence referred to as *mutators*. These algorithms are infamous for being extremely hard to program without bugs, as well as to prove correct. Indeed, they require a subtle synchronisation between threads and uttermost care as to which operations over the memory are safe in order to take into account all interferences that may occur. We come back in detail to these considerations in Chapters 4 and 5, as the algorithm we formalise and verify is indeed such an On-The-Fly garbage collector. The underlying scheme is a Mark and Sweep algorithm, its relative simplicity making it particularly suitable to support a complex concurrent optimisation.

A last recent sophistication is the adaptation to *real-time systems*. Such systems must enforce strong notions of progress: essentially, at no cost should the user program ever be delayed for more than a hard real-time limit. Examples of such systems include software managing pacemakers, or the brakes of a car. Even On-The-Fly garbage collectors may fail to comply with these constrains. Meeting these requirements is therefore a challenge of its own, notably tackled by Pizlo during his PhD [119].

2.4 CONCLUSION

Garbage collection is a rich discipline, which is nowadays a crucial part of most modern languages. Their formal verification is therefore a key challenge to bring verified compilation to managed languages, especially in a concurrent setup.

We tackle this challenge in this manuscript. Most specifically, we will describe in detail in Chapter 4 an implementation of a Mark and Sweep, On-The-Fly garbage collector, one of the most challenging kind of algorithm the discipline offers. This type of algorithm is realistic for industry-oriented programming languages. In particular, the specific implementation we consider is a realistic excerpt from an algorithm for Java introduced by Domani et al. in 2000 [35], and constituting one of the four garbage collector provided by the `OpenJDK Hotspot Java Virtual Machine` [112]. The verification of the algorithm is covered in Chapter 5.

3

REASONING ABOUT CONCURRENT PROGRAMS: THE RELY-GUARANTEE APPROACH

Before giving concrete expression to the overview of garbage collection we gave in Chapter 2 through the description of the algorithm we consider in this thesis, we introduce in this chapter the technical tools upon which we build.

We first describe the language, i. e. the logic, used to formalise and verify the algorithm: we use the *Coq* proof assistant. In order to get as close as possible to reasoning about actual code, we then introduce the programming language we formalise in Coq and in which we implement the algorithm: the intermediate representation RTIR. Finally, the proof of the garbage collector itself is conducted in a particular proof system: a so-called *Rely-Guarantee* logic.

3.1 THE COQ PROOF ASSISTANT

The second half of the twentieth century witnessed the appearance of a new interesting set of tools: *proof assistants*. A proof assistant is a computer program which fulfils two complementary tasks: on one hand it interactively helps the user to build a mathematical proof of a statement, on the other it checks the provided proof and gives guarantees about its validity. For this interaction with the computer to be possible, the proof assistant ships with a formal language representing a logic in which mathematics can be formalised.

3.1.1 *Proof assistants*

Three major motivations progressively raised the need for such tools. First, mathematicians were starting to come up with proofs so complex that very few experts could review them, at the cost of major efforts. This situation led to incorrect proofs, written up by renown scientists of good faith, standing up for years before being identified as wrong. Examples of such include the *Yamabe theorem*, claimed solved in 1960 by mathematician Yamabe, whose proof has been discovered critically incorrect in 1968. It was only another 16 years later, in 1984, that the problem was finally once again considered solved.

In the Yamabe case, the traditional mathematics eventually succeeded on their own. But even more shocking is the case of the *four colour theorem*, whose statement is extremely intuitive: can a planar map always be coloured such that no two contiguous regions share the same colour by using only four different colours? A quite convincing proof was given by Alfred Kempe in 1879, but found incorrect in 1891. This was however no small mistake: it would take nearly a century to finally witness a withstanding demonstration, in 1976 by Kenneth Appel and Wolfgang Haken. This definitive proof of the four colour theorem is of interest for an additional reason: it is arguably the historic first computer-aided proof of a significant theorem. Indeed, the proof consists in managing to reduce the colouring of any

map to the colouring of *only* 1936 (later reduced to 1476) particular elementary configurations. Checking each of these cases manually would be absurdly time-consuming, error-prone and therefore meaningless. Such a repetitive task is however perfectly fit for a computer: proof assistants grant us access to new proofs of statements of interest. The Appel-Haken still left a small room for doubts: the software used being both non-trivial and developed specifically for the task, it could be bugged. Georges Gonthier dispelled those doubts in 2002 [47] by proving once again the theorem, but this time inside of the widely-used, accepted as sound, general purpose proof assistant Coq.

Finally, the development of computer science created another use case for proof assistants, encompassing both a need for trust and automation. Wishing to reason about programs, we need to conduct reasoning which includes repetitive subtasks over objects of significant size. An emblematic use case is the one of compilers, whose behemothic size and complexity render manual guarantees of correctness extremely hard to build, and to trust.

The development of this need for a mechanisation of mathematics coincides with the apparition of the necessary tools to do so. A proof assistant indeed first and foremost relies on a theory, a logic in which mathematics can be formalised. Most modern proof assistants either rely on *Higher Order Logic* – such as HOL4 [48] or Isabelle [61] – or variants of dependently-typed theories – such as Agda [2], PVS [111], Lean [79] or our contestant of interest, Coq [23] – with rare apparition of *set theory* – with Mizar [95] and Rodin [1] notably.

3.1.2 *Coq*

The developments[1] accompanying this manuscript are developed in the Coq proof assistant[2]. This manuscript is intended to be readable with very little knowledge of Coq, we only provide to the reader a broad overview of the assistant.

Coq is based on a rich dependently typed theory: programs and proofs are formalised in the same language, the *Calculus of Inductive constructions*. Formalising mathematics in a type theory builds upon the so-called *Curry-Howard correspondence*: a type can be interpreted as a property, and a term, i. e. a program, of this type is a proof of the property. The property is therefore certified as true by a type checking algorithm attesting that the candidate *proof term* admits the property as its type. Behind this seemingly anecdotal fact hides a powerful advantage. Inconsistencies in a proof assistant would render null and void any proof it hosts. Such an inconsistency could come from the underlying mathematical theory itself: we have to trust mathe-

1 http://www.irisa.fr/celtique/ext/cgc/
http://www.irisa.fr/celtique/ext/simulin/

2 More specifically the 8.4pl6 version of Coq.

maticians on this side. But as any software, the implementation itself could also be buggy. The piece of code a proof assistant rely on for its soundness is referred to as the *trusted code base*: a proof verified in a proof assistant is certified correct *granted that* this core is correct. By relying on a type checking algorithm to validate a proof, Coq is able to have a reasonably small trusted code base: it does not matter how the proof-term is constructed as long as no bug lies in the type checker. The facilities around the checker need not be trusted.

For instance, the following type expresses the commutativity of the logical conjunction, where `Prop` is the type of propositions[3].

```
∀ A B : Prop, A ∧ B → B ∧ A
```

Correspondingly, a proof of this statement is for instance the functional program which takes as argument a pair of terms of types `A` and `B`, i. e. a pair of proofs of `A` and `B`, and returns the pair whose arguments have been swapped:

```
fun (A B: Prop) (p: A * B) ⇒ (snd p, fst p)
```

Naturally, building proof terms manually soon becomes intractable. To overcome this difficulty, Coq provides a set of *tactics*. A tactic is an instruction which manipulates the current proof term at a higher level, and builds the underlying concrete proof term, the program, under the scene. For instance, the proof of the previous statement of commutativity can be interactively done as follows.

```
Lemma and_commut: ∀ A B: Prop, A ∧ B → B ∧ A.
Proof.
  intro A B p. (* We introduce our hypotheses *)
  destruct p as [HA HB]. (* We destruct the pair p *)
  split. (* The goal being a product, we prove each goal separately *)
  apply HB. (* To prove B, we apply our hypothesis HB of type B *)
  apply HA. (* To prove A, we apply our hypothesis HA of type A *)
Qed.
```

On top of the rich set of predefined tactics, Coq provides a language *Ltac* for the user to define its own tactics. This facility is instrumental for automation. In particular, we rely heavily on Ltac in our developments to lighten the proof burden of the Rely-Guarantee proofs exposed in Chapters 5 and 6. The specifics of such tactics being quite technical and tied to Coq, we shall however not linger on their definition in this manuscript, and refer the interested reader to the formal developments.

The last feature of Coq we wish to introduce are inductive definitions. A type, be it a datatype in `Type` or a property in `Prop`, can be defined by providing a set of rules to construct its terms, the so-

3 Coq separates propositions, living in `Prop`, from data, living in `Type`. While this separation would not be necessarily in general to formalise and prove properties, it most notably allows for an automatic extraction mechanism into OCaml or Haskell of the purely computational part of a development.

called *constructors*. Intuitively, the resulting type admits as the set of its inhabitants the smallest fixpoint of these constructors.

For instance, natural numbers in the standard library are defined following Peanos's definition: a natural is either zero or the successor of a natural. This translates straightforwardly into an inductive definition.

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

In addition to being a convenient way to define datatypes, an inductive definition *automatically* generates an induction principle to reason about the resulting type. For `nat`, this inductive principle coincides with the usual recursion principle of natural numbers: prove the property holds on `0` and that assuming it holds for a number, then it holds for its successor.

```
nat_ind : ∀ (P : nat → Prop) (H0: P 0)
            (IH: ∀ n : nat, P n → P (S n)) (n: nat),
          P n
```

This example gives us the opportunity to underline two peculiar stylistic habits of Coq users that may puzzle at first scientists more familiar with set theory. First, due to the tight relationship that the mathematical language we use to write down properties shares with functional programming, it is a common pattern to replace a conjunction on the left hand-side of an implication by an implication: the property `A ∧ B → C` can be written equivalently `A → B → C`. From a functional standpoint, the rationale of this process corresponds to the well known *currying* process, taking a function admitting a pair for argument to an higher order function of one argument. In Coq, expressing properties in this style is more convenient to interact with the system, and became a folkloric habit of the community. This style therefore naturally crept into this manuscript. Second, the underlying logic of Coq is sufficiently expressive to allow quantification over any type. Implication is therefore simply a particular case of universal quantification upon which the name of the variable is not referred to in the right handside of the implication. Since it notably allows the user to enforce the way the system will name the hypothesis once introduced, universal quantification in place of implication is commonly used.

This very simple example uses an inductive construct to define a datatype. Inductive types are however far more general. In particular shall they be used to define our semantics: each constructor corresponds to a way a term may reduce.

The interested reader wishing to know more about the Coq proof assistant may want to consider the *Software Foundations* book by Pierce et al. [118] which offers a gentle yet thorough introduction to the system. Alternative resources include Bertot and Casteran's *Coq'Art* [13], as well as Chlipala's *Certified Programming with Dependent Types* [18].

This section set the world we live in: all our work is formalised in Coq, and available online. Most contributions are however agnostic from the specific choice of a proof assistant. We therefore accordingly try to expose those contributions with as few Coq notations as possible in the remainder of the document. In the remainder of the chapter, we describe two other technical tools we need: a dedicated language to implement our garbage collector and a dedicated program logic to reason about this implementation.

3.2 A DEDICATED INTERMEDIATE REPRESENTATION FOR CONCURRENT RUNTIME IMPLEMENTATIONS

Implementations of runtime for a concurrent high-level language such as Java is a complex task. The algorithms at stake, and first and foremost the garbage collector, are both inherently extremely subtle, as well as concerned with numerous low level details. Formally verifying such systems in a proof assistant with the intent to embed them in a verified compiler naturally adds an additional significant layer of difficulty.

For the task to remain tractable, it therefore is crucial to identify the right level of abstraction at which the runtime should be designed and proved. A common approach is to completely abstract away from code, approximating the algorithm as a transition system. However, doing so leaves a significant gap towards verified compilation since the refinement of a transition system by an implementation with respect to an operational semantics is a far from trivial task. We therefore want to stick to operational semantics, and argue that while this rises more challenges, it leads us closer to our end goal. We hence need to design a language capable of expressing the constructs over the memory the runtime shall handle, but nonetheless amenable to formal reasoning principles and tractable proof methodologies. Three characteristics can be identified to guide this design.

First, the runtime is a piece of code meant to be injected by the compiler into the client's code. Hence although the runtime and its proof of correctness are to be defined in isolation, they need to be able to communicate with clients.

Second, runtime implementations hold an intrinsic need for introspection. A garbage collector has to be able to traverse the heap, iterate over fields of objects or inspect the content of local variables. The sought language should therefore natively support the inspection and manipulation of objects.

$$
\begin{array}{lll}
 & X, Y \in \texttt{gvar} \qquad x, y \in \texttt{lvar} & \\
 & t, m, C \in \texttt{tid} \qquad f \in \texttt{fid} \qquad rn \in \texttt{list fid} & \\
\texttt{cmd} \ni c \;:=\; \texttt{skip} & \mid \texttt{assume}\ e & \mid\ x = e \\
\mid\ x = Y & \mid\ X = e & \mid\ x = y.f \\
\mid\ x.f = e & \mid\ \texttt{atomic}\ c & \\
\mid\ x = \texttt{alloc}(rn) & \mid\ \texttt{free}\ x & \mid\ x = \texttt{isFree?}(y) \\
\mid\ c_1\ ;\ c_2 & \mid\ c_1 \oplus c_2 & \mid\ \texttt{loop}(c) \\
\mid\ x.\texttt{push}(y) & \mid\ x = y.\texttt{empty?}() & \mid\ x = y.\texttt{top}() \\
\mid\ x.\texttt{pop}() & \mid\ X = y.\texttt{copy}() & \\
\mid\ \texttt{foreachRef}\ x \in l\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od} & & \mid\ x \leftarrow_{\texttt{ref}} r \\
\mid\ \texttt{foreach}\ (x\ \texttt{in}\ l)\ \texttt{do}\ c\ \texttt{od} & & \\
\mid\ \texttt{foreachField}\ (f\ \texttt{of}\ x)\ \texttt{do}\ c\ \texttt{od} & & \\
\mid\ \texttt{foreachObject}\ x\ \texttt{do}\ c\ \texttt{od} & & \\
\mid\ \texttt{foreachRoot}\ (x\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od} & &
\end{array}
$$

Figure 5: Simplified Syntax of RtIR. Proof annotations are elided.

Third, runtime services are elaborate concurrent services. Proving them correct implies being able to express and reason about coordination between threads, as well as manipulation of concurrent data structures. In addition, the language must be shipped with a proof methodology suitable for conducting such a reasoning.

With these rationale in mind, we introduce the intermediate representation RtIR. Our verified GC, presented in Chapter 4, is coded in RtIR, and fully proved correct with respect to its operational semantics.

3.2.1 *Syntax*

Figure 5 shows the syntax of our RunTime Intermediate Representation (RtIR). It provides two kinds of variables: *global* or *shared* variables, in `gvar`, that can be accessed by all threads, and *local variables*, in `lvar`, used for thread-local computations. Side-effect-free expressions (e) are built from constants and local variables with the usual arithmetic and boolean operators. Commands include standard instructions, such as `skip`, `assume` e, local variable update $x = e$, and classic combinators: sequencing, non-deterministic choice ($c_1 \oplus c_2$), and loops. Macros are used to define higher level constructs as needed. The usual conditional ($\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$) can be defined as ($\texttt{assume}\ e; c_1$) $\oplus$ ($\texttt{assume}\ !e; c_2$), where we write $!e$ for the boolean negation of e. While loops and repeat-until loops can be encoded similarly: $\texttt{while}\ b\ \texttt{do}\ c \triangleq \texttt{loop}(\texttt{assume}\ b; c); \texttt{assume}!b$.

RtIR also provides atomic blocks (`atomic` c). In our GC, we use atomic blocks only to add ghost-code – code only used for the proof,

not taking part in the computation – and to model linearizable data structures. While this is naturally not possible in general, we take special care in our developments to only use atomic constructs which can be refined into low-level, fine-grained implementations using techniques like [63], or the one we develop in Chapter 6. In the refinement methodology for linearisable data-structure exposed in Chapter 6, we also make use of an atomic block to define a *Compare-And-Swap* (CAS) instruction as a macro. A CAS is an instruction, natively supported by processors such as Intel's X86, which allows to perform atomically, in a single computational step, both a read and write to the shared memory. More specifically, the instruction $\mathrm{CAS}(X, e_o, e_n)$ reads the value in X, compares it to the value of the expression e_o, and if both values are equal, write into X the value of the expression e_n. The notion of atomicity of an instruction being dependent of of targeted environment of execution, using atomic blocks to define macros allows the language to be flexible and reused in different contexts.

Instruction `alloc(rn)` allocates a new object in the heap by extracting a fresh reference from the freelist – a pool of unused references – and initialising all of its fields in the record name `rn` to their default value. Conversely, `free` puts a reference back into the freelist. Instruction `isFree?` looks up the freelist to test whether a reference is in it. We use these memory management primitives to implement the garbage collector in Chapter 4.

In RTIR, basic instructions related to shared-memory accesses are fine-grained, *i.e.* they perform exactly one global operation (either read or write). These include loads and stores to global variables and field loads and updates. Apart from these basic memory accesses, RTIR provides abstract concurrent queues which implement the *mark buffers* of [35], accessible through standard operations `y = x.top()`, `x.pop()`, `x.push(y)`, `x = y.empty?()`. The use of these buffers, necessary for the implementation of the GC, will be made clear in Chapter 4. While we could implement these data structures directly in RTIR, we argue that to carry out the proof of the GC, it is better to reason about them at a higher level, and hence to assume that they behave atomically. Implementing these data structures in a correct and linearizable [56] fashion is an orthogonal problem. Chapter 6 is dedicated to this concern. Mark buffers also provide an operation `X = y.copy()`, to perform a deep copy of a whole buffer at once: we only use this operation in ghost code.

A salient ingredient of RTIR is its native support for *iterators*, allowing to easily express many bookkeeping tasks of the GC. The iterator `foreachRef x ∈ l when P(x) do c od` and its accompanying instruction $x \leftarrow_{\mathtt{ref}} r$ do not belong to the language *per se*: they are only used in the semantics. We therefore delay their description to the presentation of the semantics. The iterator `foreach (x in l) do c od` iterates `c` through all elements `x` of the static list `l`. The binder `x` is represented

here in the syntax for sake of clarity, but is actually a metavariable bound at the Coq level: command `c` has type `A → cmd` where `A` is the type of the elements over which we iterate. We come back to this technicality when describing the semantics. This iterator is used in our implementation of a GC in order to iterate over all threads. Some more sophisticated bookkeeping tasks include the visiting of all the fields of a given object, the marking of each of the roots – references bound to local variables – of mutators, or the visiting of every object in the heap (performed during the *sweeping* phase). In those cases, the lists of elements to be iterated upon is not known statically, so we provide dedicated iterators. The iterator `foreachField (f of x) do c od` iterates `c` on all the fields `f` of the object stored in `x`. Command `foreachRoot (r of t) do c od` iterates over the roots of mutator thread `t`, while `foreachObject x do c od` iterates over all objects. We stress the fact that iterators have a fine-grained behaviour: the body command `c` executes in a small-step fashion.

3.2.2 *Operational semantics*

We now turn to the description of the execution state and semantics of RTIR. The soundness of our logic, and hence the correctness of the garbage collector we prove, is phrased directly in terms of this operational semantics.

3.2.2.1 *Typing information*

The semantics of RTIR is enriched with typing information. Basic types in `typ` include `TNum` for numeric constants, `TRef` for references to regular objects, and `TRefSet` for non-null references to abstract mark-buffers: `typ` $\triangleq$ { `TNum`, `TRef`, `TRefSet` }.

Local variables, global variables, and field identifiers are declared to have exactly one of these types. Additionally, local variables are flagged as belonging to the GC, i. e. used in the injected code, or to the client, i. e. any other variable. To do so, they are defined in Coq as *records* with three fields. The types are retrieved through the corresponding accessor, respectively `lvar_typ`, `gvar_typ` and `fid_typ`, while the accessor `user` indicates if the variable belongs to the client.

```
Record lvar := { lvar_name: varId   ; lvar_typ: typ ; user: bool }.
Record gvar := { gvar_name: varId   ; gvar_typ: typ }.
Record fid  := { fid_name : fieldId ; fid_typ : typ }.
```

RTIR manipulates two kinds of values: numeric values in the Coq type `Z` and references in `ref`. Types are mapped to values with the function `value` of type `typ → Type`.

```
Definition value (t:typ):Type :=
  match t with
  | TNum ⇒ Z
```

```
  | TRef  TRefSet => ref end.
```

3.2.2.2 *Execution states*

Local (resp. global) environments map local (resp. global) variables to values of their declared type. To do so, we make use of the powerful type system of Coq: these maps are dependently types functions.

```
Definition lenv := ∀ x:lvar, value (lvar_typ x).
Definition genv := ∀ X:gvar, value (gvar_typ X).
```

As you can see, the nature, i. e. the type, of the value returned when applying the function `lenv` to an argument depends on the value of this argument. Dependent types are well known to be trodden lightly with, for they may be delicate to manipulate. However, they allow to express powerful invariants directly in the type of objects. Here, by virtue of typing, the `lenv` and `genv` maps *cannot* return a value whose nature mismatches the type of the looked up variable: a `TNum` variable can only be bound in `lenv` to integers, and so on. Using those dependent types frees us from proving the well-formedness property, and perhaps most importantly spares us from manually invoking this property when needed.

A thread-local state is then defined by a local environment and a command to execute. A global state includes a global environment `ge` and a heap `hp` – a partial map from references to `objects`. We consider two distinct kinds of objects: regular objects, mapping fields to values, and abstract mark-buffers. Global states also include two components essential to the implementation of a GC: `roots` and a `freelist`. The freelist is indeed a shared data structure, while roots are considered to be thread-local – mutators are responsible for handling their own roots with thread-local counters. Here, we model roots as part of the global state only to ease proof annotations – our final theorem is an invariant of the program global state.

```
Definition thread_state := (cmd * lenv).
Record gstate := {  ge: genv;
                    hp: ref → option object;
                    freelist: ref → bool;
                    roots: tid → ref → nat }.
```

Finally, execution states include the states of all threads and a global state.

```
Definition state := ((tid → option thread_state) * gstate).
```

3.2.2.3 *Semantics of atomic instructions*

RTIR is equipped with two kinds of operational semantics: a *big-step* semantics, and a *small-step* interleaving semantics. The big-step semantics, purely sequential, has two essential uses. First, it defines the

$$\frac{}{((\rho,\sigma) \mid \texttt{skip}) \to_t^1 (\rho,\sigma)} \qquad \frac{[\![b]\!]\rho = true}{((\rho,\sigma) \mid \texttt{assume } b) \to_t^1 (\rho,\sigma)}$$

$$\frac{x.(user) = false \quad [\![e]\!]\rho = v_e}{((\rho,\sigma) \mid x = e) \to_t^1 (\rho[x \leftarrow v]_e,\sigma)} \qquad \frac{x.(user) = true \quad [\![e]\!]\rho = v_e \quad \rho(x) = v_x}{((\rho,\sigma) \mid x = e) \to_t^1 (\rho[x \leftarrow v_e], rts[x^-, v_x^+]v_e)}$$

$$\frac{x.(user) = false \quad ge(Y) = v}{((\rho,\sigma) \mid x = Y) \to_t^1 (\rho[x \leftarrow v],\sigma)} \qquad \frac{x.(user) = true \quad ge(Y) = v_n \quad \rho(x) = v_o}{((\rho,\sigma) \mid x = Y) \to_t^1 (\rho[x \leftarrow v_n], \sigma\{rts[x^-, v_o^+]v_n\})}$$

$$\frac{[\![e]\!]\rho = v}{((\rho,\sigma) \mid X = e) \to_t^1 (\rho, \sigma\{ge[X \leftarrow v]\})} \qquad \frac{[\![e]\!]\rho = v \quad \rho(x) = r_x \quad hp(r_x) \doteq ob}{((\rho,\sigma) \mid x.f = e) \to_t^1 (\rho, \sigma\{hp[r_x \leftarrow [ob[f \leftarrow v]]]\})}$$

$$\frac{hp(\rho(y)) \doteq ob \quad ob(f) \doteq v_n \quad \rho(x) = v_o}{((\rho,\sigma) \mid x = y.f) \to_t^1 (\rho[x \leftarrow v_n], \sigma\{rts[v_o^-, v_n^+]\})}$$

$$\frac{r \in \mathcal{F} \quad hp' = hp[r \leftarrow init_{obj}(rn)] \quad \mathcal{F}' = \mathcal{F} \setminus \{r\}}{((\rho,(ge,hp,\mathcal{F},rts)) \mid x = \texttt{alloc}(rn)) \to_t^1 (\rho[x \leftarrow r], (ge,hp',\mathcal{F}',rts))}$$

$$\frac{}{((\rho,(ge,hp,\mathcal{F},rts)) \mid \texttt{free } x) \to_t^1 (\rho, (ge, hp[r \leftarrow None], \mathcal{F} \cup \{r\}, rts))}$$

$$\frac{\rho(y) = r \quad b = (r \in \mathcal{F})}{((\rho,\sigma) \mid x = \texttt{isFree?}(y)) \to_t^1 (\rho[x \leftarrow b],\sigma)}$$

$$\frac{\rho(x) = r_x \quad \rho(y) = r_y \quad hp(r_x) \doteq s}{((\rho,\sigma) \mid x.\texttt{push}(y)) \to_t^1 (\rho, \sigma\{hp[r_x \leftarrow r_y :: s]\})} \qquad \frac{\rho(x) = r_x \quad hp(r_x) = Some(v :: s)}{((\rho,\sigma) \mid x.\texttt{pop}()) \to_t^1 (\rho, \sigma\{hp[r_x \leftarrow s]\})}$$

$$\frac{\rho(y) = r_y \quad hp(r_y) = Some(v :: s)}{((\rho,\sigma) \mid x = y.\texttt{top}()) \to_t^1 (\rho[x \leftarrow v],\sigma)} \qquad \frac{\rho(y) = r_y \quad hp(r_y) \doteq s \quad b = (s = nil)}{((\rho,\sigma) \mid x = y.\texttt{empty?}()) \to_t^1 (\rho[x \leftarrow b],\sigma)}$$

$$\frac{ge(X) = r_x \quad \rho(y) = r_y \quad hp(r_y) \doteq s}{((\rho,\sigma) \mid X = y.\texttt{copy}()) \to_t^1 (\rho, \sigma\{hp[r_x \leftarrow s]\})}$$

$$\frac{hp(r) \doteq ob}{((\rho,(ge,hp,\mathcal{F},rts)) \mid x \leftarrow_{ref} r) \to_t^1 (\rho[x \leftarrow r], (ge,hp,\mathcal{F},rts))}$$

Figure 6: Semantics of atomic instructions.

semantic validity of Hoare-like tuples for basic instructions in our proof system (see Section 3.3.2). Second, it is used by the small-step semantics to define the meaning of commands in atomic blocks. The small-step semantics on the other hand is interleaving, and used to prove our final soundness results.

Both semantics share the reduction rules for elementary instructions, those which are inherently atomic[4]. Figure 6 gives the corresponding rules for the reduction relation of a thread t: given a pair of a local state and a global state, a command produces a new such pair. We do not have a new resulting command since we define here only the semantics of commands which fully execute in one single step.

```
Inductive (· | ·) →¹ₜ · : (lenv*gstate) → cmd → (lenv*gstate) → Prop.
```

We introduce a few notations to lighten the presentation of the rules. We use meta-variables $\rho \in$ `lenv` for local maps and $\sigma \in$ `gstate` for global states, and respectively ge, hp, $\mathcal{F}$ and rts for the components of a global state. The map resulting from the update of m with a new binding of v to x is written $m[x \leftarrow v]$. Updates of the `roots` mapping are handled particularly. Operation $\texttt{roots}[t, x, v_o, v_n]$ checks the type of variable x: if it is `TRef`, then it decrements, for thread t, the counter of reference v_o and increment the one of reference v_n; otherwise it does nothing. To keep rules compact, we also use ensemblist notations to manipulate the freelist. When a map is partial, such as is the case with the heap, we write $hp(x) \doteq v$ as a shorthand to $h(x) = Some\ v$. We write $r.(f)$ to access the field f of the Coq record r, and $r\{f[x \leftarrow v]\}$ to denote the record r in which the field f is updated. Finally, we assume a trivial semantics $[\![\cdot]\!]$ for effect-less expressions.

The semantics is mostly standard, but for the management of roots. We briefly comment the rules, following Figure 6 from top to bottom. Both `skip` and `assume` behave as the identity, but `assume` is naturally blocking until its condition is satisfied. Interestingly, local computations $x = e$ admit two different rules, depending on whether x is a client variable or not. Indeed, if the variable belongs to the client, we need to additionally manage the roots. If the type of x is `TRef`, the counter of the previously hold reference is decremented, while the new one is incremented. Note that we need here the semantics to depend on the thread's identifier. The case of loads of global variables $x = Y$ is similar. Global stores $X = e$ behave as expected. Loads and stores to the heap are similar to their global counterparts, except the semantics is blocking if we try to access references outside of the domain of the heap, or fields outside of the domain of the object.

Allocation $x = \texttt{alloc}(rn)$ picks non-deterministically a reference r from the freelist $\mathcal{F}$. A fresh object is bound to r in the heap: the domain of the object is the set of fields provided in rn, and each

4 Since inductive constructs with return type in `Prop` define relations and not functions, we do not need to explicitly define in Coq the subset of instructions which are atomic.

field is bound to either `0` or the `Null` reference depending on its type. Finally, `r` is removed from the freelist. Freeing a reference `r` simply removes it both from the domain of the heap and the freelist. Testing the freedom of a reference is a simple membership test to the freelist.

Next come the abstract operations over the buffers. Note once again that they are atomic, a fact instrumental to keep the proof of the garbage collector tractable, and whose formal rationalisation is covered in Chapter 6. As a result, the instruction are very simple. The `push` and `pop` operations update the buffer in the heap correspondingly, the semantics of `pop` being blocking if the buffer is empty. The `top` and `empty?` operations raise no surprise. Finally, the `copy` instruction performs the deep-copy of the buffer and stores it in the heap. Naturally, contrary to the other four, this operation cannot be refined in such an atomic way: it is only used to implement ghost code used to reason, most specifically in Section 5.5.3. The interested reader may have noticed that the semantics of buffers is a bit more complex in the formal development. Indeed, although we present them here for clarity as a simple list, enforcing a *First-In-First-Out* discipline, they are in reality less specified so that they can be refined by various implementations.

Lastly, the $\leftarrow_{\texttt{ref}}$ instruction is simply used to bind a reference to a variable at runtime. We shall explicit its use in the description of the semantics of the iterators, in the following section.

3.2.2.4 *Big-step semantics*

We build a sequential big-step semantics on top of the semantics of elementary commands. It has the same signature as the previous one, but now reduces any term from RTIR.

```
Inductive (· | ·) ⇓ ·: (lenv*gstate) → comm → (lenv*gstate) → Prop.
```

The rules are presented on Figure 7. A reduction of an atomic instruction also reduces in the big-step semantics. The control flow is completely traditional, the non-deterministic loop being expressed recursively in terms of a non-deterministic choice with `skip`. The semantics of an atomic block is simply the semantics of its innards.

Iterators are more interesting. First, `foreach (x in S) do c od` iterates a command `c` through all the elements of a list of elements of type `A`. Its intuitive semantics goes as expected for a traditional loop. However, as alluded to when describing the syntax, we make use of so-called *higher-order abstract syntax* [117] when defining iterators: `x` is not a program variable, but rather bound at the Coq level by parameterising `c` by an element of type `A`, giving it the type `A → cmd`. Intuitively, we would rather write the iterator as `foreach l do (fun x ⇒ ... x ...)`, and, the iterator would execute the sequence `(c a1) ;; ... ;; (c an)`, where each `ai` corresponds to one element in the list `l`. This way of relying on a higher-order command has several

$$\frac{(s_1 \mid c) \to_t^1 s_2}{(s_1 \mid c) \Downarrow s_2} \qquad \frac{(s_1 \mid c_1) \Downarrow s_2 \quad (s_2 \mid c_2) \Downarrow s_3}{(s_1 \mid c_1; c_2) \Downarrow s_3}$$

$$\frac{(s_1 \mid c_1) \Downarrow s_2}{(s_1 \mid c_1 \oplus c_2) \Downarrow s_2} \qquad \frac{(s_1 \mid c_2) \Downarrow s_2}{(s_1 \mid c_1 \oplus c_2) \Downarrow s_2}$$

$$\frac{(s_1 \mid \texttt{skip} \oplus (c; \texttt{loop}(c))) \Downarrow s_2}{(s_1 \mid \texttt{loop}(c)) \Downarrow s_2} \qquad \frac{(s_1 \mid c) \Downarrow s_2}{(s_1 \mid \texttt{atomic}\ c) \Downarrow s_2}$$

$$\frac{}{(s \mid \texttt{foreach}\ (x\ \texttt{in}\ \texttt{nil})\ \texttt{do}\ c\ \texttt{od}) \Downarrow s}$$

$$\frac{(s_1 \mid (c(a); \texttt{foreach}\ (x\ \texttt{in}\ S)\ \texttt{do}\ c\ \texttt{od})) \Downarrow s_2}{(s_1 \mid \texttt{foreach}\ (x\ \texttt{in}\ (a :: S))\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}$$

$$\frac{\texttt{ref\_fields}(ob) = S \quad (s_1 \mid \texttt{foreach}\ (f\ \texttt{in}\ S)\ \texttt{do}\ c\ \texttt{od})) \Downarrow s_2}{(s_1 \mid \texttt{foreachField}\ (f\ \texttt{of}\ ob)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}$$

$$\frac{}{(s \mid \texttt{foreachRef}\ x \in \texttt{nil}\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s}$$

$$\frac{P(a) = \mathit{false} \quad (s_1 \mid \texttt{foreachRef}\ x \in S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}{(s_1 \mid \texttt{foreachRef}\ x \in a :: S\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}$$

$$\frac{P(a) = \mathit{true} \quad (s_1 \mid (x \leftarrow_{\mathrm{ref}} a; c(a); \texttt{foreachRef}\ x \in S\ \texttt{when}\ )(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}{(s_1 \mid \texttt{foreachRef}\ x \in (a :: S)\ \texttt{when}\ P(x)\ \texttt{do}\ c\ \texttt{od}) \Downarrow s_2}$$

$$\frac{\begin{array}{c}\mathit{AllObjects}(\mathcal{O}) \\ (\sigma_1 \mid \texttt{foreachRef}\ o \in \mathcal{O}\ \texttt{when}\ \texttt{in\_heap}(o)\ \texttt{do}\ c\ \texttt{od}) \Downarrow \sigma_2\end{array}}{(\sigma_1 \mid \texttt{foreachObject}\ o\ \texttt{do}\ c\ \texttt{od}) \Downarrow \sigma_2}$$

$$\frac{\begin{array}{c}\mathit{AllRoots}_t(\mathcal{R}) \\ (\sigma_1 \mid \texttt{foreachRef}\ r \in \mathcal{R}\ \texttt{when}\ \texttt{True}(r)\ \texttt{do}\ c\ \texttt{od}) \Downarrow \sigma_2\end{array}}{(\sigma_1 \mid \texttt{foreachRoot}\ (r\ \texttt{of}\ t)\ \texttt{do}\ c\ \texttt{od}) \Downarrow \sigma_2}$$

Figure 7: Big-step semantics.

benefits. First, it avoids over-constraining the type A of the elements of the list. Indeed, if the binding were implicitly done via a regular local variable, this would restrict type A to the type of semantic values of our language, so that the local update of x can effectively be done at the start of each iteration of the loop. Second, it allows us to iterate over introspective values, like thread identifiers, while keeping the semantics simple: we do not need to introduce values of such types in the language. We use in our development this iterator to iterate over all thread identifiers, as well as to define a dedicated iterator over the fields of an object, `foreachField (f of ob) do c od`. The value of the set over which we iterate is computed at runtime through the predicate `ref_fields(ob)` which returns the list of fields of object `ob` holding a value of type `TRef`. The iterator will just execute the command `(c f)` for all the fields of type reference of the object stored in `ob`.

With more sophisticated examples of bookkeeping tasks, such as visiting all fields of an object or all roots of a thread, the list of elements to be iterated upon is not known statically. To provide dedicated iterators for these cases, we use a generic iterator over a list of reference, `foreachRef` $x \in S$ `when` $P(x)$ `do c od`. It iterates over a fix list of references, but also take as a parameter a predicate P ruling out dynamically references: if P is not satisfied, the reference is directly discarded. In contrast to `foreach` $(\cdot$ `in` $\cdot)$ `do` $\cdot$ `od`, this iterator binds concretely at each iteration the value iterated upon to a local variable through the dedicated instruction $\leftarrow_{\texttt{ref}}$. We then define on top of it the iteration over all objects `foreachObject do od`: predicate $\mathit{AllObjects}$ characterises the list of all objects allocated in the heap initially; during each iteration, predicate `in_heap` rules out objects which would have been freed in the mean time. Finally, the command `foreachRoot (r of t) do c od` iterates over all roots of a thread `t`, set which is once again determined at runtime by the predicate $\mathit{AllRoots}_t$.

3.2.2.5 *Small-step semantics*

Equipped with our big-step semantics, we can now define the interleaving, small-step semantics. First, we define the thread-local relation.

```
Inductive (· | ·) → (· | ·): (lenv*gstate) → comm →
                             (lenv*gstate) → comm → Prop.
```

We omit the details of the rules: they simply mimic the big-step semantics in a small-step style. The only detail worth noting is the reduction step for the atomic block: we make use of the big-step semantics.

$$\frac{(\sigma_1 \mid c) \Downarrow \sigma_2}{(\sigma_1 \mid \texttt{atomic c}) \to (\sigma_2 \mid \texttt{skip})}$$

Since no interleaving can happen inside of such a block, we execute it all at once. Finally, the interleaving semantics $\rightsquigarrow$ can be defined as a relation over `state`:

$$\frac{\begin{array}{c}\mathtt{tss}(t) = (c_1, \rho_1) \\ ((\rho_1, \sigma_1) \mid c_1) \to ((\rho_2, \sigma_2) \mid c_2)\end{array}}{(\mathtt{tss}, \sigma_1) \rightsquigarrow (\mathtt{tss}[t \leftarrow (c_2, \rho_2)], \sigma_2)}$$

3.2.2.6 *Well-typedness invariants*

A number of invariants are guaranteed by typing: (i) each variable in the local or global environment contains a value of the appropriate type, (ii) any reference of type `TRef` is either null, in the domain of the heap, or in the freelist, and (iii) each abstract mark-buffer is accessible from a unique global variable, indexed by a thread identifier. This mechanism enforces separation of mark-buffers by typing.

3.3 REASONING ABOUT CONCURRENT PROGRAMS

In 1969, Tony Hoare introduced his eponymous logic to reason about sequential programs [58]. While the approach kept spanning research projects for decades to scale to complex constructs, such as higher order recursion or stores, the fundamental intuition was already present in this seminal paper. This intuition is embodied in the aspect of the judgement the logic introduces.

Indeed, the semantic interpretation of a triplet $\models \{P\}\ c\ \{Q\}$ expresses that if the execution of the program c from any initial state satisfying the predicate P terminates in a state σ, then σ is guaranteed to satisfy the predicate Q. Note that only partial correctness is ensured: c might not terminate. Alternate systems handling total correctness exist, but we do not consider them here.

On top of the semantic interpretation of a triplet, one defines a *proof system*, or *deductive system*, whose judgements are denoted as $\vdash \{P\}\ c\ \{Q\}$. While the interpretation of the triplets are purely semantic based, a deductive system is meant to be governed by the syntax of the program. Proofs can therefore be performed modularly over the program. To ensure that these deductive proofs are meaningful, one proves the *soundness* of the system – that the deductive judgement entails the corresponding semantic judgement, i. e.:

$$\forall P,\ Q,\ c,\ \vdash \{P\}\ c\ \{Q\} \implies \models \{P\}\ c\ \{Q\}.$$

The original system is also complete: the implication is an equivalence.[5]

5 The relevant notion of completeness is a bit subtle in that the system is only complete "in the sense of Cook", i. e. *relative* to its underlying assertion language. We refer the interested reader to Glynn Winskel's wonderful book [143] for a detailed account.

We describe in this section how Hoare's idea has been adapted to handle concurrent programs, and describe our formalisation of a particular deductive system.

3.3.1 *The Owicki-Gries approach*

Reasoning about concurrent programs with shared memory brings up a major question: how can we soundly combine (sequential) proofs of different threads? Indeed one cannot reason locally without any concern for the environment: the steps another thread may take could invalidate the reasoning. This problem, referred to as *interference*, has probably been explicitly formulated for the first time by Ashcroft and Manna [10]. Their approach to soundly combine concurrent systems consists in reducing two interfering processes into an equivalent single one.

Owicki and Gries [109, 110] were the first authors to embed the difficulty into a program logic in the spirit of Hoare. They introduced the following rule for parallel composition (specialised here to the particular case of two threads for clarity):

$$\frac{\{P_1\}\ c_1\ \{Q_1\} \text{ and } \{P_2\}\ c_2\ \{Q_2\} \text{ are interference-free}}{\{P_1 \wedge P_2\}\ c_1 \| c_2\ \{Q_1 \wedge Q_2\}}$$

Their rule requires to prove correct both threads independently, and additionally to check that proofs are so-called *interference-free*. Intuitively, this condition is a sufficient condition for the operational semantics of a thread to not break the sequential proof of another one. Naturally, one such sufficient condition would be for threads to not share any variables, but this would be too restrictive for any practical use. Their condition is therefore more complex.

Essentially, they state the following. Consider *any assignment* $x := e$ in c_1, and take its associated precondition pre_1 in the sequential proof $\{P_1\}\ c_1\ \{Q_1\}$. Then this assignment must not disturb any part of the sequential proof of c_2. To ensure this fact, if you consider, for *any statement* contained in c_2, the associated precondition pre_2 extracted from the proof $\{P_2\}\ c_2\ \{Q_2\}$, then the following stability property holds: $\{\mathrm{pre}_1 \wedge \mathrm{pre}_2\}\ x := e\ \{\mathrm{pre}_2\}$. Symmetrically, the same must naturally hold by inverting the indexes.

While the approach is sound, it is quite impractical. First, the parallel composition rule is not compositional in that the proof of a thread depends on the code of the other threads it shall be composed with. The methodology therefore can only applied to closed system, for which all components are known when performing the proof. Second, the number of proof obligations grows exponentially in the number of threads involved.

3.3.2 *Rely-Guarantee reasoning*

To improve the situation, Jones introduced in 1981 the *Rely-Guarantee* (RG) logic [64]. The intuition is very similar to the Owicki-Gries approach, but offers thread compositionality.

The key idea is to prove each thread t in complete isolation, by making it write down a contract expressed as relations over the shared memory. On the one hand, t makes a promise: its execution will only modify the shared memory in ways which are described in its *guarantee* G_t. In return, t assumes something about the kind of environment it tolerates: for the proof to remain valid, t should only be subjected to interferences over the shared states contained in its *rely* R_t. A judgement has therefore now the following form: $R_t, G_t \models \{P\}\ c\ \{Q\}$.

In order to compose threads, one therefore only needs to check the compatibility of the contracts: the rely R_t should contain the guarantee of all other threads. The idea is synthesised by the composition rule, once again expressed here with only two threads for simplicity. We use set notations for operators over relations.

$$\frac{\begin{array}{cc} G_1 \subseteq R_2 & G_2 \subseteq R_1 \\ R_1, G_1 \models \{P_1\}\ c_1\ \{Q_1\} & R_2, G_2 \models \{P_2\}\ c_2\ \{Q_2\} \end{array}}{R_1 \cap R_2, G_1 \cup G_2 \models \{P_1 \wedge P_2\}\ c_1 \| c_2\ \{Q_1 \wedge Q_2\}}$$

If both contracts are compatible, the proofs of both threads may be combined. The resulting guarantee is the union of both thread's guarantees: indeed, their parallel composition can generate any effect either component could have caused. Reciprocally, the only interference they now can tolerate are interferences they both accept: the resulting rely is the intersection of both relies.

Naturally, the notion of stability of assertions remains crucial: establishing a predicate at a program point still makes no sense if an interference may invalidate it. However, while the Owicki-Gries approach need to establish such stability when composing threads, with respect to the concrete code, a RG logic has a convenient abstraction of all possible interferences a thread may endure: its rely. The relevant notion of stability of a proof annotation in a thread t is therefore the stability of this annotation under the rely relation of t. Naturally, global invariants have to be proved stable under all relies.

Definition 1 (Stability under interference)**.** *Given a domain of states* S*, a predicate* $P \subseteq S$ *is said to be stable under a relation* $R \in (S \times S)$ *if*

$$\forall \sigma\ \sigma' \in S,\ (\sigma \in P \wedge (\sigma, \sigma') \in R) \rightarrow \sigma' \in P$$

Since stability is no longer explicitly dependent on the code of other threads, stability proofs can be deferred out of the parallel composition rule. Traditionally, such as in Coleman and Jones [19], any proof rule implicitly carries a side condition stating the stability of all predicates involved. This requirement, convenient to prove

meta-theorems over the logic such as its completeness, is however impractical and overly conservative. Prensa [120] lightened the burden by checking stability only in the atomic block rule, enclosing all elementary instructions considered as atomic inside an atomic block. Vafeiadis [137] inquired several alternative strategies. While the question of stability may seem rather mundane at first glance, it turns out to be the bottleneck for a practical use of the logic, and especially so in a mechanised context. We hence introduce our own variation of a mechanised rely-guarantee logic for RTIR, carefully designed to lighten the development process of a significant RG proof, such as the proof of a GC we describe in Chapter 5.

3.3.3 *A refined Rely-Guarantee proof system for* RTIR

On top of RTIR, we formalise a RG program logic. We however design the logic with two major goals in sight: partial automation and separation of concerns.

3.3.3.1 *High-level design choices of proof rules*

In our approach, we first annotate a program, as is usually done on paper, and then prove the annotated program using syntax-directed proof rules. To this end, the syntax of commands is extended to include *annotations*. This sequential proof, and all proof rules, do not deal with stability concerns. Combined with the use of syntax-directed proof rules, this design allow for partial automation of sequential proofs. From a very pragmatic perspective, it is also relevant to remember that mechanised, sequential proofs of programs are far from easy to perform. Being able to focus on them in isolation is therefore a huge productive benefit.

Naturally, we do not claim that a RG proof should be conducted without concern for interferences. All predicates should indeed be stated with the right rely in mind. However, the proof system aims at decoupling sequential and concurrent reasoning. Its first layer is therefore a Hoare-like system, with no use of relies or guarantees. A second layer handles interferences: proof obligations about relies, guarantees and stability checks of annotations, as illustrated in Section 3.3.3.7.

Finally, to avoid polluting programs with routine annotations, typically the case for global invariants, the first layer of the system *assumes* that such invariants hold, and the second layer requires to separately prove their invariance as a stability check. Once again, this separates proof concerns. We shall see in Chapter 5 that it is instrumental in designing a tractable workflow: thanks to this separation, complex invariants can be progressively refined as understanding of the algorithm goes.

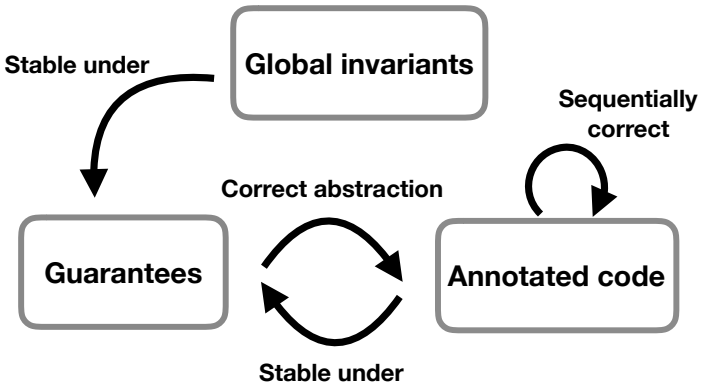


Figure 8: Structure of a RG proof in our proof system. The dependence between the components are made explicit, and proof obligations are isolated.

Before covering the presentation of the system itself, we sum up the objective by describing the resulting general structure of a rely-guarantee proof. Thanks to the separation of concerns we enforce, a RG proof in our system can be schematically represented as three interdependent components requiring proof obligations. A wrap-up of the situation is depicted on Figure 8.

First, the code of each thread is modelled through its guarantees, and global invariants shall be proved stable under any of those guarantees. The code of each thread is annotated, and those annotations shall (i) be proved to hold sequentially and (ii) be stable under the other threads guarantees, assuming the invariant[6]. Finally, for each thread, we need to prove that its guarantee is indeed an abstraction of its operational semantics.

This structure of the development is to put in contrast with traditional rely-guarantee systems. As described in Section 3.3.2, stability assertions are usually entangled with sequential reasoning by baking them directly into the proof rules. We in constrast move this concern away from the rule, as is described in Section 3.3.3.5.

3.3.3.2 *Annotations*

Two approaches can be taken when formalising a language of predicates in a proof assistant such as Coq. In a *deep embedding*, one defines the *syntax* of the language, and provide a semantics to this syntax by defining the interpretation of the terms. This approach gives great control over the resulting language, in particular in the sense that introspection is extremely easy. However, by defining a clean new world into which the predicates live, all reasoning facilities provided by Coq are lost. In contrast, a *shallow embedding* aims at leveraging the expressivity of Coq's logic. To do so, the language of assertions

6 In other words, the rely condition is implicitly taken as the union of the guarantee conditions of the other threads.

we define is given no syntax: a predicate in our language is directly encoded as a Coq predicate. Wildmoser and Nipkow investigated in depth the benefits of both approaches in 2004 [141].

We use this second solution, the one of a shallow embedding into Coq, for our language of assertions. The type of an annotation is either `pred` $\triangleq$ `gstate→lenv→Prop` for a *predicate* over the whole state, or `gpred` $\triangleq$ `gstate→Prop` when they express *global predicate*, over the global state only. Typically, the global invariant of the GC is of type `gpred`. We also define the usual logical connectives on `pred` and `gpred` with the expected meaning. Conjunction is written `A/\\B` and implication is written `A⟶B`. Annotations of type `gpred` are automatically cast into `pred` when needed.

The syntax presented in Section 3.2 is extended to take precondition annotations into account. While elementary commands that do not make use of the global state do not need to be extended, basic commands accessing memory (*e.g.* field loads and updates, global loads and stores, and mark-buffer operations) have to take an extra argument of type `pred`, representing the pre-condition of the command. This is also the case for loops, annotated with a loop-invariant, and atomic blocks, whose body may affect the global state. The semantics of RTIR completely ignores annotations which are only relevant for proofs.

In the sequel, we use the notation `P@c` to designate a command `c` whose annotation argument is set to `P`.

3.3.3.3 *Sequential Layer*

First, we define the validity of a sequential Hoare tuple, with respect to the big-step operational semantics of commands. The predicate, written `I ⊨ t: ⟨P⟩ c ⟨Q⟩`, is straightforward to define:

```
Definition bhoare (t:tid) (I:gpred) (P Q:pred) (c:cmd) :=
 ∀ gs1 le1 gs2 le2
    (PRE : P gs1 le1)    (SEM : opsem t (gs1,le1) c (gs2,le2))
    (INV1: I gs1)        (INV2: I gs2),
  Q gs2 le2.
```

This semantic judgement asserts that, for thread `t`, if command `c` runs in a state satisfying precondition `P`, and if the execution terminates, the final state must satisfy post-condition `Q` under the assumption that the global predicate `I` is an invariant. Establishing that `I` is indeed invariant is done separately.

This semantic tuple being defined, we turn to the proof system. Logic judgements for commands in this first layer are of the Hoare style form `I ⊢ t: ⟨P⟩ c ⟨Q⟩`. To ease mechanical automation of reasoning, we combine two styles when defining the proof system. For basic commands which do not require annotations and simple command compositions (sequence, non-deterministic choice and loops),

proof rules follow the traditional weakest-precondition style. A simple systematic inversion of the rules therefore permits to easily break a proof of a program into elementary proof obligations by propagating the annotations. This style can be seen in the following rules:

$$\frac{}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{skip}\ \langle \texttt{P} \rangle}$$

$$\frac{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ c_1\ \langle \texttt{R} \rangle \qquad \texttt{I} \vdash \texttt{t:}\ \langle \texttt{R} \rangle\ c_2\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ c_1 ; c_2\ \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P1} \rangle\ c_1\ \langle \texttt{Q} \rangle \qquad \texttt{I} \vdash \texttt{t:}\ \langle \texttt{P2} \rangle\ c_2\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P1} \between \texttt{P2} \rangle\ c_1 \oplus c_2\ \langle \texttt{Q} \rangle}$$

On the other hand, commands that require annotations directly embed the semantic judgement $\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{c}\ \langle \texttt{Q} \rangle$ as a proof obligation. For instance:

$$\frac{}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{fun gs le} \Rightarrow [\![\texttt{b}]\!]\texttt{le=true} \to \texttt{P gs le} \rangle\ \texttt{assume b}\ \langle \texttt{P} \rangle}$$

$$\frac{\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{c}\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@atomic c}\ \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@X} = e\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@X} = e\ \langle \texttt{Q} \rangle}$$

$$\frac{\texttt{P1} \to \texttt{P2} \qquad \texttt{I} \vdash \texttt{t:}\ \langle \texttt{P2} \rangle\ \texttt{c}\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P1} \rangle\ \texttt{c}\ \langle \texttt{Q} \rangle} \qquad \frac{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P'} \rangle\ \texttt{c}\ \langle \texttt{P} \rangle \qquad \texttt{P} \between \texttt{I} \longrightarrow \texttt{Q} \qquad \texttt{P} \between \texttt{I} \longrightarrow \texttt{P'}}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@loop(c)}\ \langle \texttt{Q} \rangle}$$

This design has similar motivations and is closely related to the choice of a shallow embedding for our logic. First, by relying directly on the semantics, we can benefit from Coq facilities to inverse inductive constructs to automatically unfold the semantics of basic programs. Second, since annotations can be any Coq predicate, a feature we exploit heavily to express easily complex graph reachability predicates, handling base cases in a syntactic way would rise difficult issues such as the definition of a substitution into our predicates.

By combining both styles, we benefit both from a syntax directed propagation of annotations and a flexible semantics-oriented resolution of proof obligations.

3.3.3.4 *Interference Layer*

This layer takes into account threads interference with a given command, handling the validity of guarantees and the stability of program annotations *w.r.t.* the context. This can be seen in the definition of a valid RG tuple:

```
Record RGt (t:tid) (R:rg) (G:list rg) (I:gpred) (P Q:pred) (c:cmd) := {
  RGt_hoare:     I ⊢ t: ⟨P⟩ c ⟨Q⟩
; RGt_stable:    stable I P R ∧ stable I Q R ∧ AllStable I c R
; RGt_guarantee: AllRespectGuarantee t I c G }.
```

Here, the type $\texttt{rg} \triangleq \texttt{gstate} \to \texttt{gstate} \to \texttt{Prop}$ defines relies and guarantees as binary relations between global states. In our development,

we build them from annotated commands. For a command P@c, the associated rg is defined by running the (big-step) operational semantics of c from a pre-state satisfying P to a post-state. More precisely, we refer to the interference an atomic command guarded by an annotation may cause to the environment as its *action*.

```
Definition action  (rg:pred*comm) : gstate → gstate → Prop :=
  let '(P,c) := rg in
  fun g1 g2 ⇒ ∃ l1 l2,
              P g1 l1 ∧ opsem rt t (g1,l1) c (g2,l2).
```

Predicate stable defines the stability of a predicate in pred *w.r.t.* a rely R, given some invariant I:

```
Definition stable (I:gpred) (H:pred) (R:rg) : Prop :=
  ∀ gs1 gs2 l,
    (I gs1 ∧ H gs1 l ∧ R gs1 gs2 ∧ I gs2) → H gs2 l.
```

Predicate AllStable then builds the conjunction of the stability conditions for all assertions syntactically appearing therein. It is formally defined as a fixpoint constructing a Coq predicate. We write P@c as a short hand to any basic command annotated with predicate P and omit the case of iterators.

```
Fixpoint AllStable (I:gpred) (c:cmd) (R:rg) (S: Prop) : Prop :=
  match c with
  | P@c ⇒ stable I P R
  | P@loop(c) ⇒ stable I P R ∧ AllStable I c R S
  | c1;c2 | c1 ⊕ c2 ⇒ AllStable I c1 R (AllStable I c2 R S)
  end.
```

The validity of the guarantee of a command (embodied in the predicate AllRespectGuarantee) follows the same principle, this time accumulating proof obligations that all elementary effects of the command are reflected by an elementary guarantee in list G. We use the notion of action to express what it means for an annotated command to respect a guarantee, expressed as a list of possible interferences.

```
Definition RespectGuarantee (I:gpred) (P:pred) (c: cmd) (G: list rg) :
    Prop :=
  ∃ g,
    In g G ∧ action (P/⋀castP I,c) ⊆ clos_refl_trans g.
```

Where clos_refl_trans g denotes the reflexive, transitive closure of g and castP I a simple casting of the gpred I to a pred by leaving the local environment unconstrained. Finally, AllRespectGuarantee collects all elementary obligations to RespectGuarantee in a command in a similar way to AllStable.

3.3.3.5 *Program RG specification*

The RG specification of a program `p` is defined as a record collecting guarantees `G` and pre- and post-conditions `P` and `Q` for all threads. Formally, we have the following definition:

```
Record RGt_prog (G:tid → rg) (I:gpred) (P Q:tid → pred) (p:program) :=
{ RGp_t:∀ t, RGt t (Rely G t) (G t) I (P t) (Q t) (cmd t p)
; RGp_I:∀ t, stable TTrue I (G t) }.
```

The predicate `RGp_t` requires that the command of each thread is proved valid. It is worth noting here that only guarantees need to be considered: for each thread, its rely is extracted from the other thread's guarantees by (`Rely G t`), which significantly reduces the redundancies of specifications. Second, `RGp_I` requires that `I` is indeed an invariant. This is encoded as a stability condition under the guarantees of each of the threads, and under the trivial invariant `TTrue` $\triangleq$ `(fun _ _ ⇒ True)`. In a nutshell, since all threads' code satisfy their guarantees, this is sufficient to prove that the global invariant `I` is preserved after any number of steps of the program.

3.3.3.6 *Reasoning about Iterators*

As expected, the case of iterators is more involved. We illustrate their treatment on `foreach`. Recall that `foreach` iterates on a list of data of type `A`, morally representing a loop. Hence, its proof involves a loop invariant, parameterised by the sublist of elements that have been already visited. Predicates annotating foreach loops are thus indexed by a list of visited elements. Moreover, the loop body often includes annotations mentioning visited elements, so we also index it by a list of visited elements and a current element. Summing up, the syntax of `foreach` extended with annotations is `P@foreach (x in l) do c od` where annotation `P` has type `list A → pred`, and `c` has type `list A → A → cmd`. The associated proof rule is:

$$\frac{\begin{array}{l}\texttt{∀ a seen, prefix (seen++[a]) l →} \\ \qquad \texttt{I ⊢ t: ⟨P seen⟩ (c seen a) ⟨P (seen++[a])⟩} \\ \texttt{P l /\ I ⟶ Q}\end{array}}{\texttt{I ⊢ t: ⟨P nil⟩ P@foreach (x in l) do c od ⟨Q⟩}}$$

The first premise amounts to proving a valid tuple whose pre- and post-conditions are adjusted to the list of already visited elements `seen`. The second premise requires the pre-condition `P` applied to the whole list of elements to entail the post-condition of the iterator itself.

3.3.3.7 *Soundness of the logic*

Soundness states that invariant `I` holds in every state reachable from a well-formed initial state – which must satisfy `I` by construction – through the small-step semantics presented in Section 3.2. Formally:

```
Hypothesis init_wf : ∀ tsi gsi, init_state p (tsi,gsi) →
    RGt_prog G I P p Q                          (* program RG spec *)
  ∧ (∀ t c le, tsi(t) = Some(c, le) → P t gsi le) (* pre-cond hold *)
  ∧ I gsi.                                      (* I holds initially *)
Theorem soundness : ∀ ts gs, reachable init_state p (ts,gs) → I gs.
```

The proof of this theorem relies on an auxiliary proof system, proved equivalent to the one presented earlier. The auxiliary system reuses the same basic components, but proof rules now require to prove everything in place: the invariant, the pre- and post-conditions, the stability of annotations, and the validity of guarantees. For instance, compare the rule for instruction $X = e$ in the previous system with the proof rule of the auxiliary system.

$$\frac{\texttt{I} \vDash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \rangle}{\texttt{I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \rangle}$$

$$\frac{\begin{array}{c}\texttt{TTrue} \vDash \texttt{t:}\ \langle \texttt{P} \mathbin{/\!\!\backslash} \texttt{I} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \mathbin{/\!\!\backslash} \texttt{I} \rangle \\ \texttt{stable TTrue (P /\ I) G} \quad \texttt{stable TTrue (Q /\ I) G} \\ \texttt{RespectGuarantee t I G (P@}X = e\texttt{)}\end{array}}{\texttt{R, G, I} \vdash \texttt{t:}\ \langle \texttt{P} \rangle\ \texttt{P@}X = e\ \langle \texttt{Q} \rangle}$$

This auxiliary system is very close to the classic RG logic [64, 138]. Its verbosity makes it easier to reason about the soundness proof.

The soundness proof itself consists in a subject-reduction lemma in the style of Wright and Felleisen [145]. We establish that, starting from an execution state satisfying `RGt_prog`, if a thread `t` takes a small-step, then `t`'s resulting code admits a new precondition such that the resulting new execution state still satisfies `RGt_prog`. Naturally, we also reestablish that the global invariant `I` holds. Invariance of `I` follows from the fact that, in each rule of the auxiliary system, the invariant is part of the pre- and post-conditions, which are stable against any step of the rely and the guarantee of the stepping thread.

3.4 RELATED WORK

The search for adequate reasoning principles for concurrent systems has been a major research area for decades. In 2012, a monograph by de Roever et al. [123] inventories and unifies more than 100 original publications on concurrency verification. Amidst this zoo, we chose for our work the well-established rely-guarantee logic. We judged and found confirmation that by tweaking its proof system to fit our needs, it strikes an adequate balance between ease of mechanisation and sufficient expressiveness to conduct the proofs we have at hand.

Nipkow and Nieto [105] formalised in Isabelle/HOL the Owicki-Gries approach. Following this work, Nieto [104] introduced in 2003 a formalisation of the rely-guarantee method. This work focuses on

the meta-theory, establishing soundness and completeness of the system, but only applies the logic to toy examples. To our knowledge, this is the only other approach for mechanising pure rely-guarantee reasoning in a proof assistant. Recently, Amani et al. [7] formalised in Isabelle/HOL a framework to formally reason on concurrent C programs based on the Owicki-Gries methodology.

However, an alternate major trend in concurrent reasoning has grown in popularity since the beginning of the millennium. Reynolds, Ishtiaq and O'Hearn introduced *separation logic* [62, 121], a twist on Hoare logic initially designed to ease sequential reasoning of mutable data structures. The gist of the approach is to reverse the interpretation of assertions. Traditionally, sharing is the default: the precondition preceding a function call constraints the accepted states, but it leaves the function the possibility to act upon any resource. Separation logic takes the reverse approach by enforcing assertions to characterise the memory much more precisely: if a resource is not mentioned in the specification of a function, then the function *cannot* modify this resource. This mechanism is enforced by introducing in the syntax of the logic dedicated operators. Predicate $x \mapsto v$ is exclusively satisfied by the heap which not only holds value v at location x, but whose domain is reduced to the singleton $\{x\}$. Assertions over bigger heaps are constructed through the separating conjunction: $P_1 * P_2$ holds over heaps which can be be split in two disjoint sub-heaps h_1 and h_2 satisfying respectively P_1 and P_2.

Brookes and O'Hearn independently observed that this new interpretation of assertions is particularly suitable to concurrent reasoning, introducing a concurrent separation logic [15, 106]. Assertions in separation logic can be interpreted as ownership of a resource, ensuring that a variable left unmentioned in the specification of a function can be safely manipulated by the environment. This results in a strong compositionality embodied by the much celebrated *frame rule*: a separation proof can have its post and pre-conditions extended with a frame through the separating conjunction.

$$\frac{\vdash \{P\}\, c\, \{Q\}}{\vdash \{P * R\}\, c\, \{Q * R\}}$$

The *Program Logics for Certified Compilers* monograph by Appel et al. [9] notably describes a formalisation in Coq of separation logic.

However, while separation logic allows for more concise arguments and better modularity than rely-guarantee approaches, it *cannot* be applied to our problem at hand. By enforcing separation of resources, the logic is unable to express the kind of fine-grained synchronisation the runtime implementations of compilers perform. The logic can only apply a protocol following strict use of critical sections, while we need concurrent accesses to shared data-structures.

The last decade has witnessed major efforts to recover the ability to express general synchronisation protocols while retaining the ben-

efits from separation logic. The two independent precursors to this reconciliation of rely-guarantee and concurrent separation logic are Vafeiadis and Parkinson's *RGSep* [138] and Feng et al.'s *SAGL* [41]. Overseeing technical details, both solutions are essentially equivalent. The heap is split in two parts: a shared one and a private one. The private part is handled in separation logic style, they are strictly separated between threads. The shared part is handled in a rely-guarantee style. These first lightweight approaches still did not manage to extend separation logic's compositionality over shared resources. Successive incremental improvements have therefore been designed over the years. Feng introduced with *LRG* [40] in 2009 a notion of *separation of actions*, essentially granting interferences with their own separating conjunction. As resources, protocols can hence be locally reasoned about. Another major contribution comes from *concurrent abstract predicates*, introduced in 2010 by Dinsdale-Young et al. [30]. Client programs using libraries can be certified with respect to abstract specifications of those libraries. Any concrete implementation of the abstract predicate can then be soundly used.

While this trend has been for a long time seemingly too unstable and evolving for mechanisation, recent impressive formalisation efforts suggest that it arrived to maturity. Liang et al. [84] proved in Coq the meta-theory for an extension of *LRG* equipped with an additional deductive system to prove program transformations. Nanevski et al. [101, 129] formalised in Coq a very expressive logic, *FCSL*, able to conduct subtle fine-grained reasoning. Resource protocols are encoded through so-called *concurroids*, transition systems describing the effects of a piece of code at a higher level than a traditional interference. Finally, *Iris* [73] is a similar higher-order logic synthesising most progress made during the last decade. It most notably has been recently used to formalise Rust's meta-theory as part of the RustBelt project [69].

Investigation on how the use of one of those modern logics would be beneficial to lighten the proof effort we needed in our development is a much desired further line of work.

3.5 CONCLUSION

Mechanised verification of programs in a proof assistant such as Coq makes for a subtle craft. Jumping head first into a task such as verifying a realistic concurrent garbage collector is bound to quickly become intractable. We therefore introduced in this chapter the technical tools we need to proceed with methodology.

First, we argued that in order to remain close to the goal of an embedding of the garbage collector into a verified compiler while allowing for a tractable proof effort, an intermediate representation should be carefully designed. To answer this concern, we introduced RTIR,

which notably supports natively introspection and iteration over objects, fields and roots. Additionally, abstract concurrent buffers can be atomically manipulated in the language.

Second, the logic in which to prove our implementation should be both sufficiently expressive to conduct the proof, while tractable enough to scale with a mechanised proof of significant complexity. We chose to this end a rely-guarantee approach, and tweaked the usual approach to this logic in order to separate proof concerns as much as possible, while easing automation.

We are now ready to jump to our main topic: we describe in Chapter 4 the garbage collector algorithm we considered, and explain its formal verification in Chapter 5.

4

DESCRIPTION OF AN ON-THE-FLY GARBAGE COLLECTOR

We covered in Chapter 2 a bird-view of the various existing garbage collection techniques. We emphasised that On-The-Fly garbage collection makes for a striking challenge in concurrent verification, leading to our interest.

The particular line of work we considered to formalise takes its roots in 1978, in a paper by Dijkstra et al. [29]. In this work, they design a concurrent Mark and Sweep algorithm in which synchronisation is reduced to a minimum so that the mutators never wait.

Fifteen years later, as part of Doligez's PhD [32–34], Doligez, Leroy and Gonthier have investigated the porting of Dijkstra's algorithm to the ML language. Among numerous optimisations and subtle implementation considerations to adapt to ML's memory model, they most notably discovered that a bug in the original algorithm had been left unnoticed during all those years, and they fixed it. Stressing more than ever the subtlety of these concurrent algorithms, the bug only arises when at least two mutators interact in an intricate way with the collector. The resulting algorithm is referred to as the DLG algorithm.

In 2000, Domani et al. [35] adapted in turn the DLG algorithm, this time with the intend to port it to Java. This final algorithm is the one upon which we build our work. While we do not model the sordid details related to Java they had to tackle, the On-The-Fly garbage collector we consider is a subset of Domani's algorithm, tackling the *full concurrency complexity*. We describe in this chapter our implementation of the GC in RTIR, the intermediate representation introduced in Chapter 3, as well as its associated correctness theorem. The verification of the algorithm is covered in Chapter 5.

4.1 SOUNDNESS OF A GARBAGE COLLECTOR

Before diving into the algorithm itself, we specify what we expect a GC to satisfy, and therefore which theorem we aim to prove.

4.1.1 *Informal statement*

Fundamentally, we want to show that the collector never reclaims memory that is later accessed by a mutator. This statement is however a bit too precise: detecting that a reference will not be accessed anymore by a mutator is undecidable in general. The GC therefore satisfies a stronger property stating that the collector never reclaims memory that is still reachable from a mutator. The latter property, which we make precise further down, naturally entails the former.

Invariant 1 (Correctness invariant (informal))**.**
If a reference can be reached through a sequence of loads and updates by a mutator, then this reference is indeed allocated in the heap.

In practice, we expect a GC not only to be harmless, but also to actually perform efficiently its collecting job. Proving such notions of liveness, asserting for instance that an unreachable reference will eventually be collected, is outside the scope of this work.

4.1.2 *Formal statement*

In order to formally prove the correctness of the GC we consider, we program the collector and the mutators in RTIR, and prove that their parallel composition preserves the correctness as an invariant on global execution states. The proof is conducted in the mechanised rely-guarantee logic described in Chapter 3, and its soundness theorem used to obtain a correctness result over the operational semantics of the code. The formalisation of this correctness statement already raises an interesting question of design. We define in this section an abstract client, so-called *Most General Client*, and phrase our correctness theorem with respect to this entity.

4.1.2.1 *Injection of the GC seen as a program transformation*

The mutators collaborate with the collector by participating in the bookkeeping required for the collection to be correct. In practice, such bookkeeping code is injected into the client code by the compiler. Sticking to the viewpoint of verified compilation, this injection can be seen as a program transformation.

Considering a program client $\mathtt{c} \in$ RTIR going through the compiling process, the injection consists in performing two main transformations over $\mathtt{c}$. First, each atomic memory operation from RTIR– i. e. `alloc`, `update`, `load` and `mov` – are substituted for their instrumented, non-necessarily atomic counterparts. This elementary transformation, denoted by T, naturally depends on the algorithm implemented.

Note that beyond the direct instrumentation of the operations required for the GC to be correct, two additional mechanisms may have to be injected. First, for obvious performance concerns, the collector should probably not collect continuously, but rather periodically. A mechanism triggering a new collection cycle is therefore needed. This mechanism is likely to be injected in the instrumentation of `alloc`. Second, as will be described in detail in Section 4.2.2, mutators may have to run a collaboration routine periodically. How often these mechanism are triggered are fine-tuning considerations, crucial to the efficiency of the GC but irrelevant to its correctness. We therefore abstract away from these considerations by considering that a new cycle can be initiated by the collector at any time, and that any mutator may run the collaboration routine at any point.

The transformation T is lifted by structural induction to transform all mutators of client $\mathtt{c}$ into a program $\mathcal{T}(\mathtt{c})$, and applying the transformation piece-wise for parallel composition of mutators. The second

step consists in composing $\mathcal{T}(c)$ in parallel with the code of the collector GC.

Proving a GC correct therefore can be seen as proving, for any client $c \in$ RTIR, that the injected code $\texttt{GC} \| \mathcal{T}(c)$ satisfies the correctness invariant. In order to avoid both manipulating explicitly an arbitrary client, and committing to a specific policy to trigger collaboration from mutators, we consider an equivalent approach based on a *Most General Client* (MGC).

4.1.2.2 *Most General Client*

Rather than explicitly reasoning over an arbitrary program, we use a most general client to prove the functional correctness of the GC. The MGC is an abstraction of an unknown client through a transition system representing a collector thread composed with an arbitrary number of mutators – with identifiers in Mut – each running the injected pieces of code resulting from the transformation T in a non-deterministic loop. In the following, we refer to these elementary pieces of code as follows:

$$\texttt{T(alloc(rn))} \triangleq \texttt{Alloc(rn)} \qquad \texttt{T(x.f=e)} \triangleq \texttt{Update(x.f,e)}$$
$$\texttt{T(x=e.f)} \triangleq \texttt{Load(x,e.f)} \qquad \texttt{T(x=e)} \triangleq \texttt{ChangeRoots(x,e)}.$$

We delay the precise definition and explanation of these operations to Section 4.2. As explained previously, mutators may run at any time a collaboration routine `Cooperate()`. The MGC can therefore be defined as:

Definition 2 (MGC)**.**

$$\begin{aligned} \texttt{mutator} \triangleq\ & \mathsf{loop}\big(\ \texttt{Update}(\texttt{x.f}, \texttt{v}) \\ & \quad \oplus \texttt{Load}(\texttt{x}, \texttt{e.f}) \oplus \texttt{Alloc}(\texttt{rn}) \\ & \quad \oplus \texttt{Cooperate}() \oplus \texttt{ChangeRoots}()\big) \\ \texttt{mgc} \triangleq\ & \texttt{mutator} \,\|\, \ldots \,\|\, \texttt{mutator} \end{aligned}$$

Open variables in the definition of the MGC should be understood as universally quantified. We present here the MGC entirely in terms of pseudo-code for the sake of clarity, but the actual Coq definition is an operational characterisation of this thread system appending the code in a non-deterministic way rather than an actual loop. Note that for convenience, we restricted in the formal development and therefore the previous definition the `Update` operation to variables as opposed to expressions. However since expressions are purely local, this is always equivalent to storing first the result of the evaluation of e into a fresh local variable and calling the `Update` operation on this variable, a behaviour that our MGC indeed enables.

We can now rephrase our goal: the MGC composed with the collector, $\texttt{GC} \| \texttt{MGC}$ should satisfy the correctness invariant. As the MGC encom-

passes by its non-deterministic choices anything an injected program could do, both approaches are therefore equivalent[1]:

$$(\forall\ c,\ (GC \parallel \mathcal{T}(c))\ \text{correct}) \text{ if and only if } ((GC \parallel MGC)\text{correct}).$$

4.1.2.3 *Formal notion of correctness*

We can now formalise the notion of correctness of a garbage collector sketched in Invariant 1. We begin by defining the reachability of a reference by a mutator.

We first define a notion of path between two references. Given a heap `hp`, a reference `r1` holding a cell `c` in `hp` is said to *point to* any reference `r2` contained in `c`. Two references are said *reachable* if they are related by the reflexive and transitive closure, denoted `star`, of the `pointsto` relation for a fixed heap.

```
Definition pointsto gs r1 r2 := ∃ f c,
    gs.(hp) r1 = Some c ∧ c f = r2.
Definition reachable gs r1 r2 := star (pointsto gs.(hp)) r1 r2.
```

In order to define what it means for a reference to be reachable from a mutator, we need to make the entry points through which a mutator can access the memory explicit. Since the language prohibits to forge a pointer, the only way for a mutator to have access to a reference is through one of its local variables. We qualify these entry points as roots[2].

Definition 3 (Roots of a mutator)**.**
Given a local environment ρ*, a root of a mutator* m *is a reference* r *bound in* ρ *to a local variable of* m*.*

As will be explained in Section 4.2.2, identifying the roots of all mutators is crucial to the GC. Remember that this bookkeeping is reflected in our RTIR in the `roots` field of a global state.

Given a global state `gs`, a reference `r` is therefore said to be *reachable from* a mutator `m` if `r` is reachable from a root of `m`.

```
Definition reachable_from gs m r2 :=
    ∃ r1, gs.(roots) m r1 > 0 ∧ reachable gs r1 r2
```

Finally, remember that the `freelist` field of a global state contains the non-allocated part of the memory. We qualify a reference belonging to the freelist as `Blue`.

1 The formal development does not prove this statement.

2 Global variables also contain roots. These global variables being however always accessible to the collector, they add no difficulty to the correctness of the GC. For this reason, RTIR uses global variables only for the injected code and assumes that the client itself does not manipulate global variables.

```
Definition Blue (gs:gstate) (r:ref) : Prop :=
    In r gs.(freelist).
```

With these ingredients, we can finally give the formal definition of Invariant 1:

Invariant 2 (Correctness invariant)**.**

```
Definition I_correct: gpred := fun gs ⇒
    ∀ m r, In m Mut ∧ reachable_from gs m r → ¬ Blue gs r.
```

The initial states we consider are meant to be obtained by a startup phase of the runtime. This phase carefully initialises the intrinsic features of the runtime, such as the freelist, such that all invariants hold. In the formal development, we abstract away from the construction of this state and manipulate an axiomatic initial state. The main theorem we prove relies on the predicate `reachable_mgc`. This predicate asserts that a global state can be reached from a predefined implicit initial state through a series of steps performed by the `MGC` in parallel with the collector. Its formal definition relying on the exact definition of the `MGC`, we omit it here. The theorem hence is:

```
Theorem gc_sound: ∀ gs, reachable_mgc gs → I_correct gs.
```

4.2 DESCRIPTION OF THE ALGORITHM

The GC we consider is of the Mark and Sweep family. The collector periodically performs a cycle during which the heap is traversed, marking cells that are suspected to be *alive*, i. e.reachable from mutators' roots. Once the marking procedure finishes, the sweeping procedure reclaims cells detected as unreachable by putting them back in the freelist.

Thanks to the underlying allocator, the algorithm has an abstract view of the memory depicted in Figure 9. The memory is seen as a graph containing two main kinds of nodes: references held in mutators' variables, the roots, are represented as circles and memory cells are represented as squares. Arrows between cells embody the *points to* relation introduced in the previous section, while a root simply points to the cell it holds in the heap.

In order to reason about the memory, we use the tricolour abstraction popularised by Dijkstra [29]. During a collection cycle, all cells are initially `White`: those cells have not yet been visited, they are potentially dead. Cells identified as potentially live are first marked `Grey` when visited. Once all successors of a `Grey` cell have themselves been explored, the cell is finally marked `Black`. Additionally, as introduced in the definition of Invariant 2, the `Blue` colour is used to designate cells which belong to the freelist. Two notions of colours must not be confused. A field `colour` is reserved to hold two potential values – denoted `WHITE` and `BLACK` –, and implemented as numerical constants.

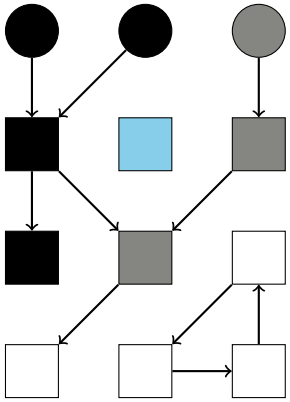


Figure 9: Abstract view of the memory as manipulated by the garbage collector.

However, and although we depict graphically the colour of the cell for representation purpose, the `White`, `Black` and `Grey` colours are more precisely an attribute of the reference which holds the cell in the heap. The `White` and `Black` colours are directly implemented in terms of the `colour` field of the cell pointed to.

```
Definition White gs r := ∃ c, gs.(hp) r = c ∧ c.colour = WHITE.
Definition Black gs r := ∃ c, gs.(hp) r = c ∧ c.colour = BLACK.
```

On the contrary, `Grey` is a logical property relying on the shared buffers that RTIR natively supports. Each thread `t` is granted a buffer `buffer[t]` and a reference is said to be `Grey` if its colour is WHITE and it is currently stored in at least one shared buffer.

```
Definition Grey gs r :=
  ∃ c t, gs.(hp) r = c ∧ c.colour = WHITE ∧ In r buffer[t].
```

In the sequel, when the context is non-ambiguous, we identify the colour of a cell with the colour of the reference holding the cell in the heap. Similarly, we represent in diagrams the local variable holding the root in place of the reference itself, and refer abusively to this variable as a root.

4.2.1 *Behaviour in a* Stop-The-World *context*

We first describe the execution of a collection cycle for a purely sequential Mark and Sweep algorithm, *a.k.a. Stop-The-World*. When a cycle is triggered, all mutators stop their execution to collaborate with the collector. This cycle is composed of three logical stages, resulting in a timeline depicted on Figure 10.

Figure 11 represents graphically a collection cycle. Initially, the memory is cleaned up by setting the colour of all cells to `White` (Fig. 11a) during the *`CLEARING`* stage. The `TRACING` stage then embodies two actions. First, the mutators publish their roots: they iterate

over the references stored in their local variables and mark them as `Grey` (Fig. 11b). The roots are naturally reachable, hence shall not be collected, and constitute the starting points to explore the accessibility graph. The collector then begins the traversal of the memory: a breadth first exploration of the graph is performed, taking each root as an entry point (Fig. 11c). During this traversal, a cell is marked as `Black` once all its successors have been marked `Grey`. At the end of this process, no `Grey` cells are left. All remaining `White` cells can be safely collected, therefore becoming `Blue`: the `SWEEPING` stage is dedicated to this reclamation (Fig. 11d).

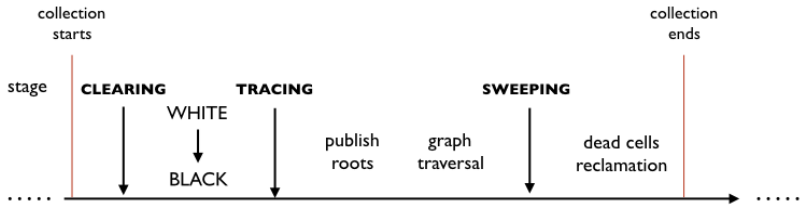


Figure 10: Timeline of a sequential execution of a collition cycle for a Mark and Sweep garbage collector.

Since all mutators stop their activity during the collection's process, the correctness of such a collector living in a sequential world is quite simple to establish: it reduces to proving correct a breadth first search in a graph. Intuitively, the key invariant to ensure is that during the trace procedure, any remaining `White` cell reachable by a mutator is guarded by a `Grey` cell.

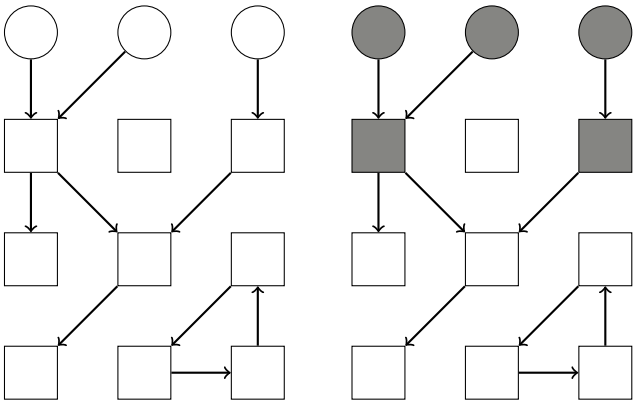
Invariant 3 (First path invariant (preliminary definition))**.**

```
Definition I_black_or_guarded: gpred := fun gs ⇒
  ∀ m r, stage[C] = TRACING ∧ reachable_from gs m r →
    Black gs r ∨ (∃ r', Grey gs r' ∧ reachable gs r r').
```

By establishing at the end of the procedure that no `Grey` cells are left, we obtain that no cell reachable by a mutator remains `White`. However, setting a cell to `Black` during the procedure could break Invariant 3 by removing the guard of a `White` cell. This issue is prevented by making sure we mark a cell `Black` only after having visited all its successors. A second invariant is therefore required to express this property. The `path` predicate is defined similarly to the `reachable` predicate, while additionally collecting the encountered references in a list. We establish that any path from a `Black` cell to a `White` cell contains a `Grey` cell.

Invariant 4 (Second path invariant (preliminary definition))**.**

```
Definition I_black_to_white: gpred := fun gs ⇒
∀ r1 p r2, stage[C] = TRACING ∧ path gs r1 p r2 ∧
```



(a) Initial memory.

(b) Publication of the roots.

(c) Ongoing tracing by the collector.

(d) Remaining white cells have been safely collected.

Figure 11: Cycle of a sequential execution for a Mark and Sweep garbage collector.

```
Black gs r1 ∧ White gs r2 →
∃ r, In r p ∧ Grey gs p.
```

In the absence of any interference, those invariants do not require any particular care for the trace procedure to preserve them. While simple, such a Stop-The-World algorithm however naturally exhibits unacceptable overheads, hence the need for concurrent alternatives such as the one we consider and describe hereafter.

4.2.2 *The* On-The-Fly *garbage collector*

We now explain the proper concurrent GC we formalised and proved correct, and introduce more specifically its RTIR code. As opposed to the Stop-The-World discipline, an *On-The-Fly* collector never interrupts the mutators.

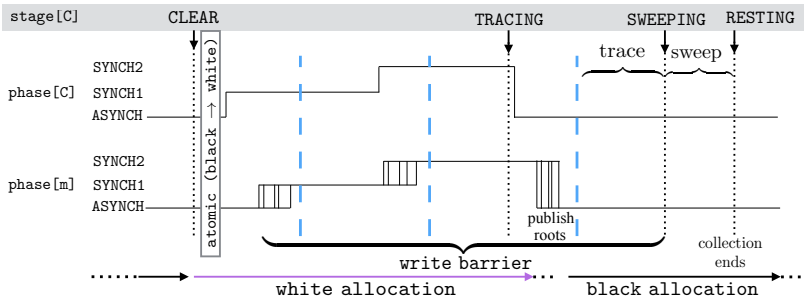


Figure 12: Timeline of an On-The-Fly collection cycle. All mutators are coalesced into the bottom line, and the collector is shown in the top line. Dotted lines represent the GC start of a new stage, and dashed lines represent the end of a phase change (handshake).

4.2.2.1 *Structure of a collection cycle*

Sticking to the Mark and Sweep scheme, the collection cycle is still divided into the three stages described in Section 4.2.1. The collector is however now hosted by a dedicated thread. Listing 1 introduces the main loop of a cycle. The stages are no longer only reasoning abstractions, but explicitly stored in a global variable `stage` and manipulated by the program.

Delaying explanations about the `handshake` procedures, the remainder of the structure is unsurprising: the collector sets successively `stage` to `CLEARING`, `TRACING` and `SWEEPING`, and calls their eponymous routines. Before dwelling into them in detail, we switch our attention towards the mutators.

As announced, the core objective is to reduce as much as possible the overhead weighing over the mutators. To this end, the mutators therefore keep running concurrently with the collector during its cycle, and should never wait. The difficulty arises naturally: mutators may modify the memory graph at any time, and thus might in particular invalidate the colouring invariants (Inv. 3 and Inv. 4). To prevent this, collaboration from the mutators is needed. Depending on the current situation of the collection cycle, they may need to change the colour of their allocation, go through *write barriers* before performing an update, whose purpose is detailed in Section 4.2.2.4, or publish themselves their roots. We therefore also need a synchronisation protocol between the collector and the mutators.

This synchronisation relies on two devices. First, the global variable `stage` allows mutators to cooperate with the collector at a coarse level. Naturally, since the collector may have kept running concurrently, this mechanism is of little value by itself. At a finer level, a handshake mechanism allows each thread to track and synchronise their status. To this end, each thread `t` is equipped with a global variable `phase[t]` whose value ranges over `ASYNCH`, `SYNCH1` and `SYNCH2`, encoded using integers (see Listing 3).

```
1  while (true) do
2    stage[C] = CLEARING
3    clear()
4    handshake()  // SYNCH1
5    handshake()  // SYNCH2
6    stage[C] = TRACING
7    handshake()  // ASYNCH
8    trace()
9    stage[C] = SWEEPING
10   sweep()
11 od
```

Listing 1: Collector

```
1  phaseC = phase[C]
2  phase[C] = phaseC + 1 mod 3
3  foreach (m in Mut) do
4    repeat phasem = phase[m]
5    until phasem == phaseC
6  od
```

Listing 2: Handshake

Figure 13: The collection cycle.

A diagrammatic representation of a cycle is shown on Figure 12, gathering all mentioned elements. We cover them in more detail in the remainder of this section.

4.2.2.2 *Mark buffers*

The `Grey` colour identifies cells visited by the collector which have a direct successor not yet visited. Accordingly, Dijkstra et al. noted in their seminal paper on On-The-Fly garbage collection [29] the need for the tracing of the memory to continue until no `Grey` cells are left. Detecting efficiently the absence of `Grey` cells has been the subject of several improvements from Dijkstra's original scans to the use of a double-ended queue [75], a scan-pointer, dirty flag and collector mark stack [32, 33] and finally the multiple mark buffers we consider [35].

In order to keep track of the remaining `Grey` cells without having to scan the memory, the algorithm makes use of shared buffers. Each mutator `m` owns a `buffer[m]` to which it can push, without synchronisation, references which as a consequence will be marked as `Grey`. The collector has its own `buffer[C]` to gather the cells awaiting its visit.

As explained in Chapter 3, RTIR natively supports such shared buffers. The `buffer[t]` variables therefore contain instances of these abstract data-structures and their operations – `push`, `pop`, `top`, `empty` – are atomic. We explain and prove this abstraction legitimate in Chapter 6.

Following the definition in Section 4.2 of the `Grey` colour, a reference is `Grey` if it points to a cell whose `colour` field is `WHITE`, and belongs to at least one buffer. The `MarkGrey` operation (Listing 6), allowing a mutator for marking a cell as `Grey`, is therefore straightforward: if the cell's colour is `WHITE`, its address is pushed onto the mutator's buffer.

```
1  // tid m : cooperate ::=
2  phasem = phase[m]
3  phaseC = phase[C]
4  if phasem != phaseC then
5    if phaseC == ASYNCH then
6      bufferm = buffer[m]
7      foreachRoot (r of m) do
8        markGrey(bufferm, r)
9      od
10   phase[m] = phaseC
```

Listing 3: Cooperate

```
1  //tid m: x := Alloc(typ)
2  atomic
3    x' = alloc(typ)
4    phasem = phase[m]
5    stageC = stage[C]
6    if (phasem != ASYNCH || stageC == CLEARING)
7    then x'.colour = WHITE
8    else x'.colour = BLACK
9    x = x'
```

Listing 4: Allocation

```
1  // tid m : update(x,f,v) ::=
2  phasem = phase[m]
3  stageC = stage[C]
4  if (phasem != ASYNCH
5      || stageC == TRACING) then
6    old = x.f
7    bufferm = buffer[m]
8    markGrey(bufferm,old)
9    markGrey(bufferm,v)
10 x.f = v
```

Listing 5: Write Barrier

```
1  // markGrey(buffer,x) ::=
2  if x != NULL then
3    xcol = x.colour
4    if xcol != BLACK then
5      buffer.push(x)
```

Listing 6: MarkGrey

Figure 14: The mutators' operations instrumented by the GC.

4.2.2.3 *Publication of the roots*

A key element of the cycle explained in Section 4.2.1 was the publication of the roots of the memory graph, from which the traversal is performed. Now that the collector is hosted inside its own thread, it cannot perform this publication by itself since it has no access to the mutators' local variables. This work is therefore put in the hands of the mutators themselves, which are expected to periodically check that the collector is waiting to start a new cycle and publish their current roots.

As explained in Section 4.2.2.1, the `stage` global variable can be used as a weak means of synchronisation between the collector and the mutators. However, the `stage` variable does not provide a reciprocal mechanism for the mutators to indicate to the collector, nor to each other, that they indeed performed this publication. This synchronisation is thus performed through the additional `phase[t]` variables.

Roots publication is handled thanks to this mechanism: before calling the `trace` procedure, the collector initiates a `handshake` (Listing 2). It then changes its own `phase` to the `ASYNCH` value (line 7) in order to signal to mutators it is ready to scan the memory, and is waiting for them to publish their roots. In the meantime, the collector enters a waiting loop until all the mutators are synchronised. Remember that our golden rule is to never stop a mutator, but that it is fine for the collector to wait for them.

On the mutators' side, the `cooperate` procedure (Listing 3), periodically run, performs the synchronisation. If a mutator detects that it

is not in phase with the collector, it adjusts its own phase. If additionally the mutator remarks that the collector's phase is set to `ASYNCH`, the mutator also publishes its roots by marking them as `Grey`.

Naturally, to avoid publishing several times their roots, a second `handshake` is needed by mutators to synchronise with the collector switching back to `SYNCH`, on line 4. We delay to Section 4.2.2.6 the justification for the presence of a third `handshake`.

4.2.2.4 *Write Barriers*

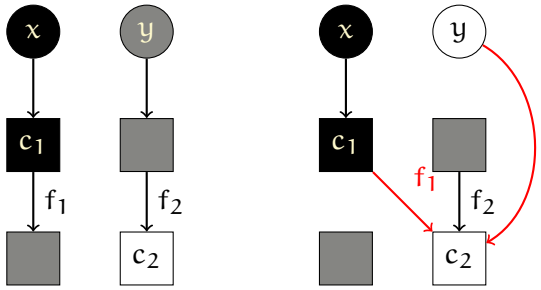


Figure 15: Dangerous concurrent `update` during `TRACING`.

Any invariant over the memory graph may naturally be broken by a rash update. Figure 15 depicts a scenario highlighting the issue. The collector is in the middle of the trace procedure, having already marked a cell c_1 as `Black`. A mutator may concurrently perform two operations. First, a `load`$(y, y.f_2)$ gives local access to a `White` cell c_2 to the mutator. Second, an `update`$(x.f_1, y)$ leads the reference holding the `Black` cell c_1 to point to the one holding the `White` cell c_2: Invariant 4 is broken.

In order to prevent this dire situation, mutators pass in a so-called *write barrier* before performing an `update`[3], whose full code is detailed in Listing 5. Every time an `update` is performed, instead of simply updating the value of the field with the atomic `x.f = v` operation, the mutator first evaluates which `stage` the collection cycle currently is in. If it detects the `update` might indeed occur concurrently with the trace procedure, it marks as `Grey` the cell to which the reference will now point to (line 9). The situation described in Figure 15 is therefore avoided and Invariant 4 is ensured.

Surprisingly, the write barrier, on line 8, also marks as `Grey` the cell which used to be pointed at before the `update`. This can however not be avoided, as is illustrated in Figure 16. The cornerstone is once again that although the mutators publish their roots, they may perform afterwards a `load` creating a new `White` root. Figure 16 exhibits the operation `load`$(y, x.f)$ giving local access to the `White` cell `c`.

3 The reader may argue that we blamed the `update` operation for a heist it performed jointly with a `load`. Indeed, alternative algorithms choose to use load barriers. However, a typical program performing more `loads` than `updates`, this solution tends to be more costly.

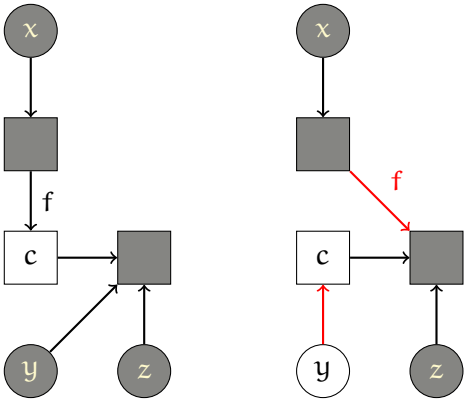


Figure 16: Updates need to also mark the cell previously pointed at.

This is not a problem in general, since the `load` operation could not be performed if c was not reachable from another entry point of the graph, x in this example. However this now vital path could be cut by a subsequent `update`: Figure 16 performs the operation `update`(x.f, z), removing all accesses to c but for local variable y.

By ensuring during an `update` that the old value is also marked as `Grey`, we guarantee that a new starting point is created to explore this potentially cut branch of the memory graph. By doing so, we therefore preserve Invariant 3. Note that Domani et al. [35] avoid marking `old` in some cases. We drop this optimisation.

4.2.2.5 *Allocation*

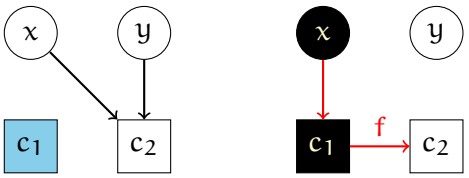


Figure 17: Dangerous `Black` allocation.

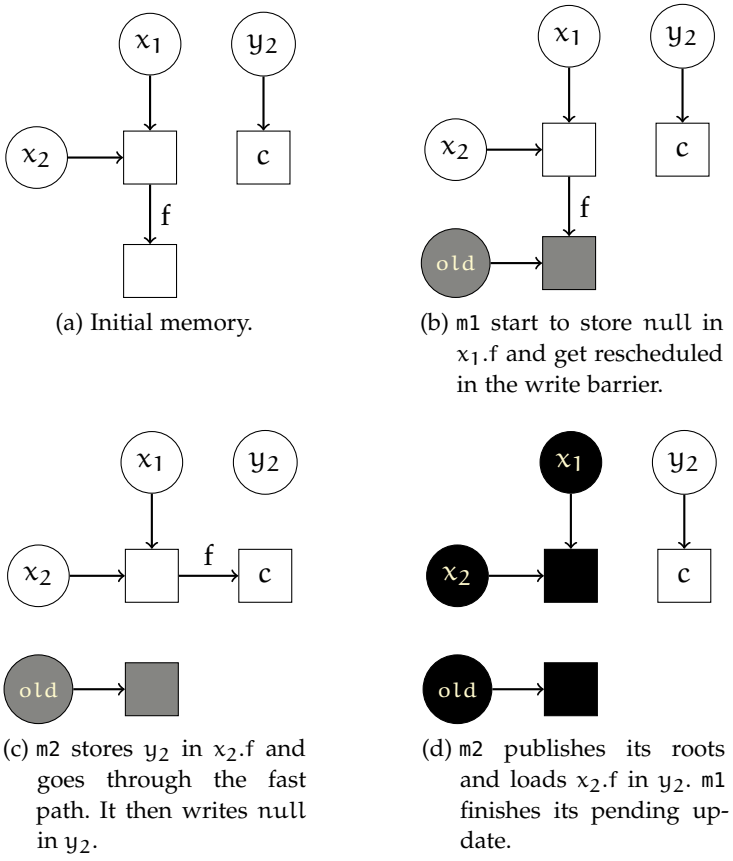
A second preoccupation when considering an On-The-Fly garbage collector is the colour of the allocation. Obviously, one does not want to allocate a `White` cell after the roots have been published: the cell would never be visited, and would immediately be reclaimed.

However, it turns out we cannot simply allocate using the colour `Black` systematically neither! Figure 17 shows a simple situation with a cell c_2 addressed by a root, and a non-allocated cell c_1. We consider a sequence of three operations.

Suppose first that c_1 is allocated `Black` during the `CLEARING` stage, and stored in x. The operation `update`(x.f, y) would then lead c_1 to point to c_2. We already broke Invariant 4. Even worse, the local link to c_2, through y, can be broken via a `load` to `null`. We end up in a situation where c_2 remains reachable, but is not protected anymore

by neither a `Grey` cell nor a `White` root which could be published later. The colour of allocation must therefore change along the cycle, as shown in Listing 4 and summed up in Figure 12.

4.2.2.6 *The need for a third handshake*



(a) Initial memory.

(b) `m1` start to store `null` in x_1.f and get rescheduled in the write barrier.

(c) `m2` stores y_2 in x_2.f and goes through the fast path. It then writes `null` in y_2.

(d) `m2` publishes its roots and loads x_2.f in y_2. `m1` finishes its pending update.

Figure 18: In presence of several mutators, a third handshake required.

The last mystery we promised to unveil before considering the collector's routines is the presence of a third `handshake` on Listing 1, and accordingly of two distinct phases `SYNCH1` and `SYNCH2` rather than simply one `SYNCH` phase. As a matter of fact, the original algorithm by Doligez and Leroy [33] only used two phases, a synchronisation policy that Doligez and Gonthier later discovered incorrect [32]. Indeed, the bug is quite tricky as it only occurs when considering the presence of several mutators.

Consider the situation depicted on Figure 18 where we only assume two `handshakes`. On Figure 18a, we are at the beginning of a cycle and three characters are at play: the collector is looping in its first `handshake`, waiting in the `SYNCH` phase for all the mutators to join him; mutator `m1` already collaborated, its phase is also `SYNCH`; mutator `m2` however still is in `ASYNCH`. We subscript local variables by the index of the mutator's owning them. Mutator `m1` is about to perform

a single operation: `update`$(x_1.f, null)$. Mutator `m2` on the other hand has the following code scheduled.

`update`$(x_2.f, y_2)$; $y_2 = null$; `cooperate()`; `cooperate()`; $y_2 = x_2.f$

Mutator `m1` initiates the operation `update`$(x_1.f, null)$. Being `SYNCH`, the write barrier triggers, the current value of `x.f` is loaded in `old` and marked grey (line 6 on Listing 5). This way, when the update itself will be performed, the loosing branch it could create will be safely protected, which we have seen in Section 4.2.2.3 is mandatory. The update remains pending until `m1` is rescheduled, we are in the situation shown on Figure 18b.

Now `m2` gets scheduled and performs its own store, `update`$(x_2.f, y_2)$, but goes through the fast path since it still is in `ASYNCH`. Once this is done, it loads null into y_2 and finally performs its own first `cooperate`, setting its phase to `SYNCH`. The collector can therefore end its loop, get into the `TRACING` stage, set its phase to `ASYNCH` and initiate the second handshake. The current state of the heap is represented on Figure 18c.

Mutator `m2` is now ready to jump straight into the trap. First, it `cooperates` a second time, therefore publishing its roots. Since it erased y_2 beforehand, `c` will not be marked grey by this publication. However right after, it executes `load`$(y_2, x_2.f)$ to get back its local store to `c`. Mutator `m1` is finally rescheduled and able to finish its pending update. This has for effect to render `c` unreachable from mutator `m1`. It now cooperates and publish its roots, but those roots do not protect `c`: we are in Figure 18d's situation. The collector can now trace the memory and reclaim `c`, despite being reachable from mutator `m2`.

Adding an intermediate synchronisation phase, and therefore a second handshake, is sufficient to solve this issue[4]. Intuitively, a handshake flushes any pending update. By adding an intermediate phase `ASYNCH 1`, we therefore guarantee that everyone gets through write barriers before anyone can publish its roots.

We therefore obtain the full synchronisation protocol depicted on Figure 12. It is important to understand that although during this section we hand-waved quite liberally about the synchronisation protocols for didactic purpose, they are extremely subtle to ensure and prove. Indeed, imagine a mutator loads the value of the `stage` variable to decide whether it should go through a write barrier. The concurrency being fine-grained, the collector may change its `stage` right after this load. More generally, synchronisation knowledge about other threads is always approximate, leading to complex reasoning.

4 Doligez and Gonthier identified two alternatives to solve this issues, but both imposed an additional overhead to the update operation.

```
1  // trace() ::=
2  buffC = buffer[C]
3  all_empty = false
4  while (!all_empty) do
5    all_empty = true
6    foreach (m in Mut) do
7      buffm = buffer[m]
8      is_empty = buffm.isEmpty()
9      while (!is_empty) do
10       all_empty = false
11       x = buffm.top()
12       col = x.colour
13       if (col == WHITE)
14       then
15           buffC.push(x)
16           buffm.pop()
17       else
18           buffm.pop()
19       is_empty = buffm.isEmpty()
20     od
21   od
22   is_empty = buffC.isEmpty()
23   while !is_empty do
24     all_empty = false
25     ob = buffC.top()
26     col = ob.colour
27     if (col == WHITE) then
28       foreachField (f of ob) do
29         obf = ob.f
30         if obf != NULL
31           colf = obf.colour
32           then if colf == WHITE
33             buffC.push(obf)
34       od
35       ob.colour = BLACK
36     buffC.pop()
37     is_empty = buffC.isEmpty()
38   od
39 od

41 // sweep() ::=
42 foreachObject o do
43   isfree = isFree?(o)
44   if !isfree then
45     col = o.colour
46     if col == WHITE then
47       free(o)
48 od

50 // clear() ::=
51   atomic
52     foreachObject o do
53       isfree = isFree?(o)
54       if !isfree then
55         o.colour = WHITE
56     od
```

Listing 7: Clear, Trace and Sweep (Collector)

4.2.2.7 *The clear procedure*

We now turn our attention towards the three routines called by the collector, depicted on Listing 7.

During the `CLEARING` stage, the collector has to set all cells to the `White` colour. Traversing the whole memory is naturally costly. Domani et al. [35] have therefore introduced a radical optimisation: knowing that no `White` cells are left at the end of a cycle, since those cells have been reclaimed, they simply switch a bit inverting the encoding of the `White` and `Black` colours. This way, the traversal of the memory is replaced by a single atomic bitflip. We chose to model this optimisation as an atomic block (line 51) setting the colour of all currently allocated cells to `White`.

4.2.2.8 *The trace procedure*

The trace procedure, spelled out from line 2 to line 39 on Listing 7, performs the marking of all accessible cells. The traversal remains

similar in spirit to a breadth first traversal, but gets more complicated to account for the dynamic evolution of the graph.

The procedure iterates a main loop, at line 4, until the `all_empty` boolean is found true. During each iteration of this loop, two successive internal loops are performed.

First, the collector scans successively each mutator's buffer, starting from line 6. As long as the buffers are found to be empty, `all_empty` remains true and the collector does not modify anything. However, if it finds any non-empty buffer, the collector sets `all_empty` to false for the remainder of the current external iteration. In such a case, the procedure will necessarily go through a new iteration of the loop. Indeed, any non-empty buffer found means that an accessible branch of the memory graph may remain to be explored. The collector therefore acknowledges this burden of work to be treated by pouring the mutator's buffer into its own buffer through the loop on line 9. Since a cell may be shared between several mutators, a mutator's buffer might contain a cell which has already been handled, and hence is already `Black`. In this case, the reference is simply removed from the buffer, without transferring it into the collector's buffer.

Once all buffers have been scanned, the collector checks whether it has found any new work to process, on line 22. If not, the iteration ends. Otherwise, we enter in the second internal loop until the collector's buffer has been emptied once again. We consider the cell `c` at the top of the collector's buffer: if `c` is `Black`, we can dismiss it; otherwise, it is the starting point of an unexplored path. In this case, any non `Black` direct successor of `c` is pushed up into the collector's buffer to be dealt with later. Once all successors have been considered, `c` is itself marked `Black`. At the end of this loop, the collector has handled all the work the mutators had handed to it. However, similarly to a modern danaïde, the mutators may have filled back their buffers in the mean time. The collector therefore initiates a new iteration, until all buckets are witnessed empty during a single iteration.

4.2.2.9 *The sweep procedure*

Once the marking is over, the sweep procedure, on Figure 7, is straightforward: any remaining allocated `White` cell is reclaimed. This operation is conveniently expressed in RtIR with the built-in iterator over objects.

4.3 CONCLUSION

We described in this chapter the On-The-Fly garbage collector we formalise and verify. The algorithm is complex: numerous concurrent accesses to the memory are involved, and a subtle synchronisation protocol is needed to ensure that the memory retains its integrity. In

addition, the correctness of the tracing of the memory is highly non-trivial.

In the following chapter, we describe how the arguments scattered in this presentation are formalised in Coq to prove the correctness of the implementation in RTIR of the garbage collector. The proof is conducted in the rely-guarantee proof system we introduced in Chapter 3.

5

VERIFICATION OF THE GARBAGE COLLECTOR

We turn to the formal proof in Coq of the GC described in Chapter 4, using the proof system introduced in Chapter 3.

Remember that a proof in RG is composed of three main intertwined components:

- a set of invariants proved stable under all guarantees;
- a set of guarantees proved to be satisfied by the annotated code;
- an annotated code, both proved to be sequentially correct and containing only annotations stable under all guarantees.

While the proof system has already been carefully designed to enable separation of proof concerns between sequential reasoning and stability checks, mechanising such a sizeable proof still raises additional methodological concerns related both to the intrinsic complexity of the algorithm, as well as to the scalability of the approach.

A first observation is that stating upfront the right set of invariants, guarantees, and assertions is unrealistic for such a proof. In practice, one rather adjusts and refines their definitions through the development, and in particular seeks for solid intermediate results. To tackle this problem, we group invariants related to different aspects, *e.g.* the phase protocol or the colouring procedure. Those groups are ordered such that a given group of invariants is provable assuming only the underlying groups. To reflect this architecture in our development, and to avoid constant refactoring of proof scripts, we designed an incremental workflow.

A second observation concerns the stability of the global invariant of the GC, embedded in proof obligation `RGp_I` introduced in Section 3.3.3.5. Foreseeing the whole development, this proof obligation involves 18 invariants, which must be proved stable under 17 guarantees, thus requiring 306 stability proof obligations. On top of this, proof obligation `RGt_stable` adds more than 60 annotated lines of code, each bearing several predicates, that must be proved stable under significant subsets of the 17 guarantees. This becomes quickly intractable without a disciplined methodology and automation.

We expose in this chapter the methodology we developed to cope with these issues and provide an overview of the formal proof of the GC.

5.1 GHOST VARIABLES

The code we verify is precisely the one we introduced in Chapter 4. One detail has nonetheless been hidden during the description of the algorithm: the use of so-called ghost variables.

The intuition behind the notion of ghost variable is to deeply embed in the language logical variables used for reasoning. While they

facilitate the specification and verification of the program, its semantics should be independent of their presence. A typical example is the use of a boolean ghost variable to capture program points: we set the variable to true when entering a program block, and to false when exiting the block. Referring to this variable in the specification allows for a convenient way to capture whether the execution is currently inside the block.

In order to ensure that ghost variables introduced do not alter the semantics of the program, we follow for their use a strong politic: they should only be written to, and never read. Proving that functional correctness still holds over the program once the ghost code has been cleaned up is therefore an easy, administrative task.[1]

Two ghost variables are introduced to identify portions of code. The first one, `ghost_hs`, holds the value `1` while the collector is inside an handshake, and `0` otherwise. To this end, the code on Listing 2 actually begins with the instruction `ghost_hs = 1` and ends with `ghost_hs = 0`. The second one, `ghost_flip`, is used to identify one single line of code from the collector: the instant in between the setting of the `stage` to `CLEARING` and the bitflip, at lines 2 and 3 in Listing 1. To this end, line 2 becomes `atomic <stage[C] = CLEARING; ghost_flip = 0>` and the assignment `ghost_flip = 1` is introduced in between lines 3 and 4.

Additionally, each mutator `m` is equipped with a ghost variable `ghost_buff[m]`. These variables are used in the trace operation to perform a snapshot of all mutator's buffers. The snapshot is performed at line 5 on Figure 7, at the beginning of each loop iteration, with the following atomic code:

```
atomic // ghost code
  foreach (m in Mut) do
    ghost_buffer[m].copy(buffer[m])
  od.
```

This snapshot is crucial to prove the correctness of the trace procedure, as we detail in Section 5.5.3.

5.2 MODELLING THE CODE AS GUARANTEES

Before getting into proving the invariants, we first need to introduce the guarantees. Instructions which do not impact the shared memory cannot compromise the stability of an assertion. The guarantees are therefore only required to over-approximate the effectful instructions.

While the logic itself allows the guarantees to be expressed as any relation over states as long as they over-approximate the semantics of the program, we argue that expressing them directly through their

1 This cleaning operation has however not been formalised in our work. Inspiration could be drawn from the work of Hofmann and Pavlova [60] who have proposed a formal model in Coq of ghost variables allowing for their automatic elimination.

PhaseG	:= phase[m] = pc	phase[m]
Update_fieldG	:= x.f = v	path
Mut_pushG	:= buffer.push(x)	White
AllocG	:= (...)	path
RootsG	:= *axiomatised*	Reachable_from

(a) Effects issued by a mutator m.

ClearG	:= < stage[C] = CLEARING; ghost_flip = 0 >	stage[C], flipped
BitflipG	:= *axiomatised*	Black, flipped
StageG p	:= stage[C] = p	stage[C]
Hs_enterG	:= ghost_hs = 1	hs
HsG	:= phase[C] = phaseC + 1 mod 3	phase[C]
Hs_exitG	:= ghost_hs = 0	hs
FreeG r	:= free(r)	colours, path
Ghost_CopyG	:= *axiomatised*	ghost_buffer
Col_pushG	:= buffC.push(x)	White
Col_popG	:= bufferC.pop()	Grey
Col_pushG	:= bufferC.push(x)	White
Upd_colourG	:= x'.colour = BLACK	Grey, White

(b) Effects issued by the collector.

Figure 19: The injected code abstracted as guarantees. Annotations on the right are indications as to which properties of interest may be compromised by the operational effect of the guarantee.

operational semantics whenever possible is beneficial to the mechanisation. Indeed, as introduced in Chapter 3, Coq heavily relies on inductive definitions: a type can be defined by its constructors and the set of its inhabitants is the smallest fixpoint of these constructors.

Inductive types can be used both to define datatypes, such as the syntax of our programming language, as well as arbitrarily complex predicates. In particular, we use them to define our semantics: each constructor corresponds to a way a construction may reduce. Consider for instance the big step relation which, for a given thread identifier and command, relates an initial state to the resulting state after executing the command. The signature of this relation reads as follows:

```
Inductive opsem : tid → (gstate*lenv) → comm → (gstate*lenv) → Prop
```

In addition to the induction principle already mentionned in Chapter 3, inductive definitions automatically generate a mechanism named *inversion*. Since the only way to build a value of an inductive type is through its constructors, one can inspect its structure to deduce infor-

mation about it. For instance, the constructor of `opsem` corresponding to the sequence is a straightforward translation of the usual rule.

```
| opCSeq : ∀ t s1 s2 s3 c1 c2,
           opsem t s1 c1 s2 →
           opsem t s2 c2 s3 →
           opsem t s1 (CSeq c1 c2) s3
```

If we know that `Cseq c1 c2` reduces a state `s1` into a state `s3`, inverting this hypothesis introduces a new state `s2` and two hypotheses `opsem t s1 c1 s2` and `opsem t s2 c2 s3`. Iterating this process of inversion yields a very convenient way to automatically compute the relation between two states related by `opsem`, granted the program under consideration does not contain loops. We make the most of this trick by using the notion of action introduced in Chapter 3. When possible, interferences are directly expressed in terms of a command in RTIR, whose context of execution is constrained by a precondition. By defining them in terms of the operational semantics, interferences can hence be systematically inverted. We then refine the context in which they may occur along the development by strengthening incrementally the precondition.

Figure 19 sums up the operational components of the guarantees over-approximating the effect of the injected code. Since we express guarantees directly in terms of the operational semantics, most of them straightforwardly correspond to the atomic operations from the GC affecting the shared memory. We nonetheless benefit from some factorisation: `StageG` encompasses two occurrences of update to the `stage` variable during the main loop, the third one requiring extra care due to the ghost variable `ghost_flip`; `Col_pushG` subsumes both popping operations performed by the collector during the `TRACING` procedure. This kind of minor trick will require more care in refining the precondition of the actions, but allows for some factorisation in arguments, especially stability ones.

One guarantee is particular: `RootsG` does not map to any actual code but rather directly to the `MGC`. This effect models a `load` performed by the client. Indeed, the GC instruments most of the effectful operations that could be performed: updates, allocations and freeing the memory. The loads are however left untouched, hence were not introduced as instrumented code, but must nonetheless not be forgotten in the model.

As we explained, we try to stick to the operational semantics so that its systematic inversion may automatically compute the relation between states. This however cannot work when the code contains loops, as is the case in the guarantees `Ghost_CopyG`, modelling the snapshot of the buffers in the trace procedure, and in the guarantee `BitflipG`, modelling the bitflip. In those cases, annotated as *axiomatised* in Figure 19, we axiomatise their effect as a Coq inductive, and

prove once and for all the correctness of this axiomatisation. This breaks the uniformity of automation, but is restrained to sufficiently few cases to remain manageable.

Foreseeing the arguments we will develop during this section, annotations in red hint at which properties may be compromised by the corresponding guarantees. Those properties are: colours of cells, all of which are in particular denied by the `free` operations since they assume liveness of the cell; reachability of a cell from the roots of a mutator; existence of a path between two cells; values of global variables.

5.3 THE SYNCHRONISATION PROTOCOL

The GC algorithm being fine-grained, the synchronisation between threads is entirely based on the global variables `phase` and `stage`. The involved colouring invariants which lie at the crux of the proof hence rely on the correctness of this synchronisation protocol. A first major step in the development is therefore to identify a provable subset of invariants entailing the main synchronisation invariant which states that a mutator is never ahead of the collector, and at most one step behind. Intuitively, this invariant holds since all threads are always perfectly synchronised, except while a handshake is in progress. Handshakes being initiated by the collector, the collector is indeed one step ahead in the mean time. As usual, this invariant cannot be formally proved by itself but needs to be associated with several others, detailed on Figure 20.

Two minor invariants, `phase_typ` and `ghost_hs_typ`, simply constrain the potential values of the `phase` variables and the `ghost_hs` ghost variable. While those very mundane facts quite clearly hold, they immediately highlight one of the challenges of mechanisation: they both need to be proved stable under all seventeen guarantees. While this is intuitively trivial in all cases but for the few which modify those variables, we need to engineer a way to reflect this triviality in the formal development. We will come back to this issue shortly. More interesting, the invariant `handshake_out` locally strengthens `handshake_correct` using the ghost variables: when the collector is not inside the handshake procedure, all threads are perfectly synchronised. Finally, two invariants relate the stage of the collector to the phases of the threads: everyone is `ASYNCH` during the `SWEEPING` stage, and witnessing a mutator lagging behind in `SYNCH 2` is sufficient to affirm that we are inside the `TRACING` stage.

We want to prove that the conjunction of these five predicates, referred to as `synch_protocol`, is invariant. The natural way to proceed is to first consider the stability of `synch_protocol` under `Gs`, the union of all guarantees of both the mutators and the collector. Having initialised the actions defining the guarantees with an uncon-

```
Definition handshake_correct: gpred :=
  fun gs ⇒ ∀ t, In t M → (phase[C] gs = phase[t] gs ∨
                          phase[C] gs = phase[t] gs ⊕ 1).

Definition phase_typ: gpred :=
  fun gs ⇒ ∀ t, In t M → phase[t] gs = SYNCH1 ∨
                         phase[t] gs = SYNCH2 ∨
                         phase[t] gs = ASYNCH.

Definition ghost_hs_typ: gpred :=
  fun gs ⇒ ghost_hs gs = 0 ∨ ghost_hs gs = 1.

Definition handshake_out: gpred :=
  fun gs ⇒ ∀ t, In t M → phase[C] gs = phase[t] gs.

Definition sweeping_asynch : gpred :=
  fun gs ⇒ stage[C] gs = SWEEPING →
            ∀ t, In t (C::M) → phase[t] gs = ASYNCH.

Definition asynch_tracing : gpred :=
  fun gs ⇒ ∀ t t',
            In t M → In t' (C::M) →
            phase[t] gs  = SYNCH2 →
            phase[t'] gs = ASYNCH →
            stage[C] gs  = TRACING.
```

Figure 20: The set of invariants characterising the synchronisation protocol.

strained precondition, this cannot hold straightforwardly: guarantee `HsG` would break the invariant through the increment made to the phase of a collector which could already be in advance. We therefore carefully refine the guarantees for the stability to hold, while keeping in mind that we need to be conservative enough to manage afterwards to sequentially derive this precondition by annotating the code. Note however that while all code annotations need to be stable, this concern does not hold over the preconditions of guarantees.

Looking back at Figure 19, six guarantees are liable to impact the validity of `synch_protocol` by acting upon either the `stage`, the `phase` of a thread or the value of `ghost_hs`. Their preconditions are therefore reinforced to prevent this. During a mutator's collaboration, the new value assigned to its phase is always the current collector's one: indeed, the collector is necessarily stuck in the handshake's waiting loop when this program point is reached, and therefore cannot have updated its phase concurrently. `HsG` can be heavily annotated, we know quite precisely the state of things at this line. Namely all threads are synchronised before the increment to `phase[C]`, and the value of this phase defines the current stage. Similarly, during the updates to `stage`, the value of the phase is precisely known.

We subsequently establish the stability of `synch_protocol` under this new definition of `Gs`. By annotating the code, mainly to track the value of the `stage` and ghost variables during the collector's cycle, we can prove that we soundly refined our guarantees: they indeed embed the operational semantics of the code.

5.4 PROOF METHODOLOGY

At this stage of our development, we have established the validity of `synch_protocol`, an invariant describing the synchronisation protocol holding about a code partially annotated, modelled by a set of guarantees whose preconditions are weakly constrained. Armed with this first step, we naturally want to build upon `synch_protocol` and prove additional invariants.

However acting upfront turns out to be highly impractical: for the new invariants to hold, the guarantees will have to be further refined to take into account new subtleties about the code. But `synch_protocol` having been proved stable with respect to the previous definition of `Gs`, all proofs of stability might break. Intuitively, all lemmas of stability we established should still hold since we only gather more information about the execution, hence restrict the possible interferences. In practice however, mechanised proof scripts are extremely fragile objects, and constant refactoring of proofs is a daunting and time consuming task. We therefore formalise the intuition to build a methodology allowing us to actually work incrementally upon this first step.

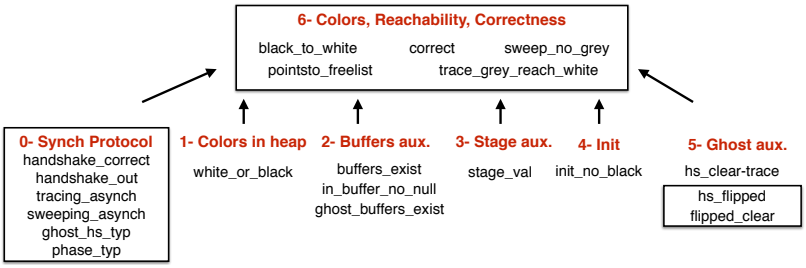


Figure 21: Main Invariants of the GC. Numbers are timestamps in the incremental proof methodology. Dependencies are shown with boxes (inter-dependency) and arrows.

5.4.1 *Workflow*

In order to work incrementally, we organise the invariants in groups, as illustrated on Figure 21. In each boxed group, invariants are inter-dependent, while arrows indicate a dependency of the target group on the source group.

While RG proofs are thread-modular, using RG does not solve the inter-dependency problem, since invariants, guarantees and code annotations all interact in proofs. To maximise proof reuse, we use a simple mechanism: in our Coq development, the invariants `I` and guarantees `G` are indexed by a natural number – morally a timestamp of their introduction into the development (see Figure 21). The first established set of invariants, `synch_protocol`, is hence defined as `(I 0)`. When introducing a new increment to `I`, all invariants with a lower timestamp are therefore not modified. Since the same mechanism goes for the guarantees and code annotation, the previous proofs of stability are not modified, resulting in an incremental and non-destructive methodology. More concretely, at each level:

1. we enrich the invariant, refine the guarantees and code annotations;
2. we prove the new stability proof obligations, for which we can reuse prior stability proofs, and we use automation to discharge as many obligations as possible;
3. we adapt sequential Hoare proofs, and prove that enriched guarantees are still valid.

This workflow proved robust during our development, allowing for an incremental and manageable proof effort. We detail below the first two items of this methodology.

5.4.2 *Incremental proofs*

Let us focus on proof obligation `RGp_I` from Chapter 3, which requires establishing that the invariant is stable under all threads' guarantees.

Let us index both the invariant and guarantee by `n`. The obligation we consider is thus `(stable TTrue (I n) (G n))`. Let us now see how we establish `(stable TTrue (I n+1) (G n+1))` by using the already proved `(stable TTrue (I n) (G n))` obligation.

5.4.2.1 *Monotonicity of `I` and `G`.*

We build `(I n+1)` as a conjunction of the prior established invariant `(I n)`, and the increment at the current level: `(I n+1)` $\triangleq$ `(I n)` $\between$ `(Ic n+1)`. Hence, we have that $\forall$`n,(I n+1)` $\longrightarrow$ `(I n)`.

Recall that in our proof system, guarantees are expressed through the effect of a command, under certain hypotheses on the pre-state. At each level, the command will not change – it is effectively executed by the code. Levels are rather used to refine the hypotheses on the pre-state. Therefore, guarantees are monotonic in the sense that $\forall$`n,(G n+1)` $\subseteq$ `(G n)`: they are made more precise as the level index increases.

In order to reuse the existing proof of the prior set of invariants, we start by proving that prior invariant `(I n)` is stable under refined guarantee `(G n+1)`, *i.e.* `(stableTTrue (I n) (G n+1))` holds. To this end, we reuse our previous proofs at level `n` and conclude with the following lemma using guarantee monotonicity.

```
Lemma stable_refineG: ∀ C I G1 G2,
  G2 ⊆ G1 ∧ stable C I G1 → stable C I G2.
```

5.4.2.2 *New Invariant Stability.*

It remains to prove the stability of increment `(Ic n+1)` under refined guarantee `(G n+1)`.

In the simplest case, we can prove `(stable TTrue (Ic n+1) (G n+1))` independently from prior invariants. In this situation, by a simple lemma, we combine the stability of `(Ic n+1)` and `(I n)` into the one of `(I n+1)`:

```
Lemma stable_and: ∀ C I1 I2 G,
  stable C I1 G ∧ stable C I2 G → stable C (I1⩓I2) G.
```

However, the situation is often more involved, requiring prior invariants to prove the stability of `(Ic n+1)`. Formally, we have `(stable (I n) ((Ic n+1)) (G n+1))`. We can then combine the stability of `(I n)` and `(Ic n+1)` under `(G n+1)` using this stronger lemma:

```
Lemma stable_with: ∀ C I1 I2 G,
  stable C I1 G ∧ stable I1 I2 G → stable C (I1⩓I2) G.
```

Finally, code annotations go through the same indexing treatment in order to preserve their previous proof of stability. We introduce an indexed conjunctive operator `and_at`, denoted by •step $\wedge$ iter•,

which only adds the annotation when the code is considered at an iteration `iter` greater than its step of introduction:

```
Definition and_at_iter (step iter: nat) (A B: pred): pred :=
if leb step iter
then fun gs le ⇒ A gs le ∧ B gs le
else A.
```

We then define a fixpoint `comm_impl` computing the minimal set of proof obligations required to leverage a proof of stability of the code from one iteration index to the next. The proper definition of `comm_impl` is quite technical, but the intuition behind it is simple. When considering two commands c_1 and c_2, with the intend to be provided with two versions of a piece of code, it inductively traverses both commands simultaneously. If any operational difference is found, it generates an impossible proof obligation, `False`. Otherwise, it compares the annotated proof annotations P_1 and P_2 and generates a proof obligation expressing both (i) that P_2 is a strengthening of P_1, i. e. we indeed added annotations to the code, and (ii) that P_2 is stable under the current guarantees, assuming the stability of P_1 that we are meant to have previously established.

These generated proof obligations are proved to be correct in the sense that they are sufficient to lift a previous proof of stability of a piece of code:

```
Lemma AllStable_strengthenC: ∀ rt I c1 c2 G S,
  comm_impl rt I c1 c2 G →
  AllStable rt I c1 G S →
  AllStable rt I c2 G S.
```

Assisted with dedicated tactics, we obtain at annotated code level the same benefits we obtained for invariants' stability.

5.4.3 *Proof Scalability*

To tackle the blowup of stability checks alluded to earlier, we built a toolkit of structural stability lemmas, and develop some tactic-based partial automation. This allowed us to discharge automatically 186 obligations among the 306 obligations induced by `RGp_I`. The remaining obligations are also partially reduced by the automation.

Structural lemmas serve three purposes. First, they are critical to enable the incremental methodology delineated above. Second, they allow for complex stability proof obligations to be simplified: both annotations, invariants, and interferences can be structurally split up. Thus, intrinsically complex arguments are isolated from trivial ones, that are automatically discharged. Finally, to reuse as much proofs as possible, we rely on a custom notion of stability under extra-hypotheses: we reestablish after interference a predicate `P` (for instance, the invariant of a new layer) by assuming that another pred-

icate H holds before and after the interference (for instance, the invariants established at previous layers).

```
Definition stable_hyps (I: gpred) (H P: pred) (R: rg): Prop :=
  ∀ gs1 gs2 l,
  I gs1 ∧ H gs1 l ∧ P gs1 l ∧ R gs1 gs2 ∧ I gs2 ∧ H gs2 l → P gs2
    l.
```

Typically, this notion allows to leverage stability results from previous levels, notably in the following lemmas:

```
Lemma stable_weakI: ∀ I1 I2 P G, I2 ⊆ I1 → stable I1 P G → stable I2 P
     G.
Lemma stable_weakH : ∀ I (H P: pred) R,
    stable I H R → stable_hyps I H P R → stable I (H⩛P) R.
```

By decomposing annotations and relaxing interferences, we can factor out the proof of stability of annotations that reappear in the code.

5.4.3.1 *Automation.*

We developed a set of tactics that decomposes stability goals into elementary ones before attempting to solve them. This leads to clearer goals and more tractable proof contexts. The tactics combine our structural lemmas with two additional ideas: systematic inversion on guarantee actions – defined operationally using commands –, and rewriting in predicates.

5.5 PROOF OF CORRECTNESS

We now turn to the full proof of correctness. The complete set of invariants, organised in successive groups, has been alluded to on Figure 21. Our core target of interest is the final batch of inter-dependent invariants containing the correctness invariant, that we will refer to as `colouring_correct`. Using the incremental methodology described previously, we have been able to insert on the fly additional auxiliary invariants, indexed from 1 to 5 in Figure 21. The methodology allowed us to establish them once and for all in a proof context free of any consideration related to colouring and path invariants, and recover them in the ongoing effort to prove `colouring_correct`.

5.5.1 *The correctness set of invariants*

Figure 22 defines the main colouring invariants whose correctness is inter-dependent with the final correctness invariant of the GC.

The `correct` invariant itself is no surprise: as introduced in Chapter 4, no reference reachable by a mutator is `Blue`, i. e. in the freelist.

Two invariants, also introduced in Chapter 4, specify paths along the memory graph. First, the `black_to_white` invariant specifies that

```
Definition sweep_no_grey: gpred :=
  fun gs ⇒
    stage[C] gs = SWEEPING →
    ∀ r, ¬ Grey TID gs r.

Definition trace_grey_reach_white : gpred :=
  fun gs ⇒
    stage[C] gs ≠ CLEARING →
    ∀ t,
      In t TID →
      phase[t] gs = ASYNCH →
      ∀ r,  reachable_from t gs r →
            (∃ r0, Grey TID gs r0 ∧ reachable gs r0 r) ∨ Black gs r.

Definition black_to_white : gpred :=
  fun gs ⇒
    ge gs ghost_flipped = 1 →
    ∀ r1 r3 p, path gs r1 r3 p →
               Black gs r1 →
               White TID gs r3 →
               (∃ r2, Grey TID gs r2 ∧ In r2 p).

Definition pointsto_freelist: gpred :=
  fun gs ⇒ ∀ r1 r2,
      reach_single gs r1 r2 →
      in_freelist gs r2 →
      (White TID gs r1 ∨ Grey TID gs r1).

Definition correct: gpred :=
  fun gs ⇒
    ∀ t gs r,
      In t TID →
      reachable_from t gs r →
      ¬ Blue gs r.
```

Figure 22: The set of invariants characterising the functional correctness of the garbage collector.

a path from a `Black` reference to a `White` one necessarily passes through a `Grey` reference. We only add one restriction: this invariant is broken at one of the collector's point of program, that we rule out using the `ghost_flipped` variable. This small technicality arises from our model of bitflip and allocation. Indeed, switching to the `CLEARING` stage, on line 2 in Listing 1, atomically changes the colour of allocation to `White`. This operation is separated from the bitflip itself, occurring right after. In the algorithm from Domani et al. however, the bitflip takes care of both side effects at the same time, switching the significance of the `Black` and `White` colours, and thus changing simultaneously the allocating colour and the current colour of the heap. To handle this difference, we therefore relax the `black_to_white` invariant in between the change of stage and the bitflip.

Second, the `trace_grey_reach_white` makes things precise once the trace procedure has started: a reference reachable by a mutator is either `Black` or guarded by a `Grey` reference. We capture the part of the timeline following the start of the trace procedure by checking that we are no longer in the `CLEARING` stage, and that the mutator from which we can reach the reference has already collaborated for the trace procedure to start, i. e. is `ASYNCH`.

The `sweep_no_grey` invariant asserts that during the trace procedure, the collector indeed completed its work, and its result is not invalidated by further interferences by the mutators: during the `TRACING` stage, no `Grey` cells remain. Combined with `trace_grey_reach_white`, we immediately obtain that no reachable reference is `White`, hence such references can be safely reclaimed.

Finally, the `pointsto_freelist` asserts that if a reference `r` points to another reference which is contained in the freelist, then `r` cannot be `Black`. This invariant is a convenient way to establish that allocation will not create ill-formed paths. Indeed, when allocating a new reference `r2`, this reference could still be pointed at by another reference `r1`. This is indeed prohibited if `r1` was reachable by a mutator, since `r2` was in the freelist, but is not ruled out for unreachable parts of the memory. Proving that `r1` cannot be `Black` is sufficient to ensure stability of our invariants, and can be established.

We now focus on two major elements of the formal proof: the correctness of the write barriers and of the trace procedure.

5.5.2 *Write barriers*

Despite being a quite short piece of code, write barriers are extremely subtle: we exhibit their fully annotated implementation on Figure 23. The `update_pre` annotation simply expresses the fact that `x` and `v` are roots of mutator `m`. We introduce the other predicates in the following. Two main properties of the write barriers need to be established.

First, if a reference r undergoes the markGrey procedure during the SWEEPING stage, then r has to be Black. Indeed, that would otherwise make r a Grey reference, and contradict invariant sweeping_no_grey.

The second concern naturally occurs when the update is performed. Special care is required when the invariant trace_grey_reach_white is active to prevent breaking it. First, the new reference v should be either Grey or Black since it is a new reference reachable from a mutator. The situation of the erased value, x.f, is more subtle. Note that in general this reference is not the same as the one stored in old, since interferences may have changed this fact. As explained in Section 4.2.2.4, there are cases where we need to have marked it Grey, but as shown in Section 4.2.2.6, that is not always true. What does hold is that this reference is necessarily Grey or Black once any potential concurrent update from another mutator necessarily also goes through the write barrier, i. e. once they have all undergone at least one handshake. We consequently need to establish the following property when the proper update occurs:

```
(stage[C] gs = TRACING → ∃ old,
    pointsto gs (le x) f old ∧ Grey_or_Black TID gs old) ∧
((stage[C] gs = TRACING ∨ phase[t] gs ≠ ASYNCH) →
    Grey_or_Black TID gs (le v)).
```

Once again, note however that synchronisation is quite loose: a mutator might enter the write barrier due to the collector being in the TRACING stage, but only actually perform the update once the collector is back in the CLEARING stage. In particular, a property such as

```
(stage[C] gs = TRACING ∨ phase[t] gs ≠ ASYNCH) →
      Grey_or_Black TID gs (le v)
```

is unstable: we need to strengthen annotations to ensure their stability.

We thus have to consider the two reasons for which the mutator m might have entered the write barrier. First, if m was not ASYNCH we do not have much to do: we know in that case that we cannot be in the SWEEPING stage, as enforced by invariant sweeping_asynch. And since no one can change m's phase but itself, this situation is stable. Now if m was ASYNCH and the collector in the TRACING stage when the mutator entered the write barrier, we know in particular that the invariant trace_grey_reach_white was valid at this time. Hence references held in v and x, naturally reachable by m, are either Black or guarded as long as the stage remains TRACING, and necessarily Black if it is SWEEPING. We embody this intuition in the following wb_annot annotation.

```
Definition wb_annot (t: tid) (r:ref) : pred :=
  fun gs le ⇒
    le tmp_stageC = TRACING →
    le tmp_phase = ASYNCH →
```

```
(stage[C] gs = TRACING → Black gs r ∨ ∃ r0, Grey TID gs r0 ∧
reachable gs r0 r)
∧ (stage[C] gs = SWEEPING → Black gs r).
```

We crucially propagate inside the write barrier the knowledge that references `x`, `v`, and hence `old` once loaded, all satisfy this predicate. We therefore establish in particular that references marked grey during the `SWEEPING` stage are `Black`. However, the resulting assertion is still not stable for the `old` variable. The reference the `old` variable contains could be tempered with by another mutator, and hence not be reachable anymore by `m`. Indeed, contrary to `v`, `old` is not a client variable, and hence does not behave as a root. We therefore introduce `wb_annot_old` and `annot_old` specifying the situation of this particular variable:

```
Definition wb_annot_old (t: tid) (r:ref) : pred :=
  fun gs le ⇒
    (stage[C] gs = TRACING ∨ phase[t] gs = SYNCH2) →
    (∃ o, pointsto gs r f o ∧
              (le old = o ∨ Grey_or_Black TID gs o)).

Definition annot_old (t: tid) (r:ref) : pred :=
  fun gs le ⇒
    (stage[C] gs = TRACING ∨ phase[t] gs = SYNCH2) →
    (∃ o, pointsto gs r f o ∧ Grey_or_Black TID gs o).
```

If we were in `SYNCH 1`, `old` could indeed end pointing to a `White` reference: this is the bug described in Section 4.2.2.6. However, the introduction of a third handshake as prescribed by Doligez fixed this issue: as soon as the mutator is in `SYNCH 2`, any concurrent update also necessarily goes through the write barrier. As a consequence, the reference to `old` points through `f` may indeed have been changed, but the mutator having made this changed necessarily enforced the desired properties, i. e. this new reference is still either `Grey` or `Black`. Once `old` has been marked, we are hence certain that it points to a `Grey` or `Black` reference, as stated by `annot_old`.

5.5.3 *Verifying the `trace` procedure*

As stated in Chapter 4, the code of the trace routine is the most challenging to verify. While describing the behaviour of the procedure, we stated that it is safe to exit the loop once all mutators' buffers have been witnessed empty during a simultaneous iteration. We explicit here the rationale behind this statement.

Crucially, at the exit of the external loop constituting `trace`, we are able to prove that: (i) there are no more `GREY` objects, (ii) all objects reachable from the mutators roots are `BLACK`, and consequently

```
  {update_pre m}
  phasem = phase[m]
  {update_pre m •6 ∧ iter• (fun gs le ⇒ phase[m]gs = le tmp_phase)}
  stageC = stage[C]
  if (phasem != ASYNCH  stageC == TRACING) then
    {update_pre m •6 ∧ iter• (fun gs le ⇒
      phase[m]gs = le tmp_phase
      ∧ (le tmp_phase ≠ ASYNCH ∨ le tmp_stageC = TRACING)
      ∧ wb_annot m (le v) gs le
      ∧ wb_annot m (le x) gs le)}
    old = x.f
    {update_pre m •6 ∧ iter• (fun gs le ⇒
      phase[m]gs = le tmp_phase
      ∧ (le tmp_phase ≠ ASYNCH ∨ le tmp_stageC = TRACING)
      ∧ wb_annot m (le v) gs le
      ∧ wb_annot m (le x) gs le
      ∧ wb_annot m (le old) gs le
      ∧ wb_annot_old m (le x) gs le)}
    bufferm = buffer[m]
    {update_pre m
      •2 ∧ iter• (fun gs le ⇒ le old ≠ NULL)
      •6 ∧ iter• (fun gs le ⇒
      phase[m]gs = le tmp_phase
      ∧ (le tmp_phase ≠ ASYNCH ∨ le tmp_stageC = TRACING)
      ∧ buffer[m]gs = le tmp_buffer
      ∧ wb_annot m (le v) gs le
      ∧ wb_annot m (le x) gs le
      ∧ wb_annot m (le old) gs le
      ∧ wb_annot_old m (le x) gs le)}
    markGrey(bufferm,old)
    {update_pre m
      •2 ∧ iter• (fun gs le ⇒ le v ≠ NULL)
      •6 ∧ iter• (fun gs le ⇒
      phase[m]gs = le tmp_phase
      ∧ (le tmp_phase ≠ ASYNCH ∨ le tmp_stageC = TRACING)
      ∧ buffer[m]gs = le tmp_buffer
      ∧ wb_annot m (le v) gs le
      ∧ annot_old m (le x) gs le
      ∧ wb_annot m (le x) gs le )}
    markGrey(bufferm,v)
{update_pre m •6 ∧ iter• (fun gs le ⇒
  phase[m]gs = le tmp_phase
  ∧ annot_old m (le x) gs le
  ∧ ((phase[t] gs ≠ ASYNCH ∨ stage[C] gs = TRACING) →
  Grey_or_Black TID gs (le v)))}
x.f = v
```

Figure 23: Annotated code for a write barrier performed by a mutator m.

(iii) there are no `WHITE` objects reachable from any of the mutators roots.

Property (i), namely that all buffers are *simultaneously* empty at the end of tracing (Listing 7, line 39), is particularly difficult to establish, given that mutators may concurrently execute write barriers. We prove that this property is valid at line 4 of the last iteration of the enclosing while loop. We proceed as follows. We first prove that, at line 4, `buffer[C]` is always empty. As for mutators' buffers, we use ghost variables `ghost_buffer[m]` to take their snapshot at line 4. Mutators can only push on their buffers, so, in a given iteration of the enclosing while loop, if a mutator buffer is empty, so was its ghost counterpart during the same iteration. In the last iteration of the while loop, all buffers are witnessed empty, one at a time. But this implies that all ghost buffers are simultaneously empty at line 8. This, in turn, implies that all buffers are, this time *simultaneously*, empty at line 4. This property remains true until line 39: it is both stable under mutators' guarantees, and preserved by the while loop. Finally, if all buffers are empty (there are no `GREY` objects), the above invariant `trace_grey_reach_white` implies that both the old and new objects that `markGrey` could push on a buffer are in fact `BLACK`, and thus not pushed on any buffer (Listing 6). As a consequence, no reference is pushed on the collector's buffer (line 15).

5.6 RELATED WORK

Garbage collectors have always been an emblematic verification challenge. Many prior efforts have tackled various instances of the sequential case. A similar copying collector algorithm has been proved correct both in the CakeML compiler [99] and the Milawa prover [25]. Some impressive projects providing end-to-end verified specialised operating systems include a verified garbage collector, such as Ironclad Apps [54] and Verve [146].

Ericsson et al. [125] have recently equipped the CakeML compiler with the first verified generational copying garbage collector. They managed to quite elegantly structure the proof such that a partial collection of the heap over a generation is proved correct by being simply interpreted as a collection over a smallest heap from which exiting pointers are ignored. The CertiCoq project [8] also reports on the verification of a similar algorithm.

Hawblitzel and Petrank [53] introduced an interesting different line of work in 2009. They rely on the Boogie verification condition generator [81] in combination with the Z3 SAT/SMT solver [26] to verify down to assembly two sequential garbage collectors, an elementary Mark and Sweep algorithm and a Cheney copying algorithm. The approach consists in implementing the algorithms in x86 assembly, and

heavily annotating the code. Boogie and Z3 then take care of generating and discharging the corresponding proof obligations.

Garbage collectors have also been a playground for separation logic in recent years. Torp-Smith et al. [134] use it to verify a copying garbage collector. McCreight et al. [92] extend CompCert with a new intermediate language, GCminor, equipped with memory management primitives and providing a target of compilation for managed languages. Using separation logic, they subsequently partially prove a Cheney copying collector. In 2010, McCreight et al. [91] prove a machine-level implementation of a variety of standard garbage collector implementations.

All the previously cited works only deal with *sequential* garbage collectors. The first arguments of correctness of a concurrent garbage collector were already introduced by Dijkstra et al. [29] with notably the three colours abstraction. The proof is however not fully formal. The first mechanised proof was presented by Gonthier [46], and conducted in the TLP system. Unlike ours, Gonthier's proof rests on an abstract encoding of the algorithm. The work of Havelund [52] in PVS follows the same pitfall. Pavlovic et al. [114] propose a different approach, synthesising a more realistic concurrent garbage collector from a simpler, abstract, initial implementation proved correct. While the approach is certainly appealing, they do not manage to reach algorithms as fine-grained as the one we consider.

Gammie et al. [45] verified in 2015 an On-The-Fly algorithm close to ours. Similarly to Gonthier's work, they do not perform the proof over the code itself, but over an abstract model of their algorithm. They also design a more monolithic proof, following an Owicki-Gries approach to the problem. However they take into account a more realistic memory model than we do. Indeed, they account for the fact that processors are *not* sequentially consistent: they may exhibit behaviours which do not match any interleaving of the atomic instructions. More specifically, Gammie et al. support the *Total Store Order* (TSO) memory model, well known to be the one exhibited by x86 processors. It is however worth mentioning that they restrained Domani et al.'s algorithm by enforcing additional synchronisations to ease their reasoning. Extending our own work to TSO is one of our major perspectives.

Separation logic [121] having been democratised in the sequential case, the application of its concurrent extension [15, 106] to the proof of garbage concurrent collectors is a promising line of research. However, to the best of our knowledge, the only successful attempt to date is the work by Liang et al. [84, 85]. They introduce a Rely-Guarantee-based Simulation (RGSim), embarking separation logic assertions, for compositional verification of concurrent transformations. It can be thought of as a syntactic proof system to build simulations between concurrent programs. They use this system to prove

a mostly-concurrent garbage collector in a seemingly pleasantly concise way. While the garbage collector they handle is still simpler than ours and its proof is not mechanised (only the meta-theory of the proof system is), they offer here a very promising line of research.

By sidestepping any modelling in our formal development by proving directly the implementation in RTIR, we argue that our work constitutes the closest-to-be-executable verified On-The-Fly garbage collector to date.

5.7 CONCLUSION

We gave in this chapter a description of the formal proof of the GC we introduced in Chapter 4, using the program logic presented in Chapter 3, conducted inside the Coq proof assistant.

The proof turned out to be challenging from a conceptual perspective, notably through the interaction of fine-grained synchronisation with subtle colouring invariants, as well as from a mechanisation standpoint. To handle the complexity, we combined a methodological contribution – from the design of RTIR and our logic, to the development of an incremental methodology – with pragmatic engineering – through handcrafted partial automation. The complete development ends up weighing a little over 20kloc.

In this manuscript, we put a special emphasis towards operational aspects: for the long term goal to be the construction of a verified compiler for a Java like language, the midterm goal must be to verify an executable implementation of the runtime. In the following chapter, we tackle the main obstacle separating us from this objective: refining the abstract data-structures natively embedded in RTIR.

6

COMPILATION OF LINEARIZABLE DATA STRUCTURES

We introduced in Chapter 4 an On-The-Fly realistic garbage collector inspired by the work of Domani et al. [35]. We presented its implementation in RTIR, our dedicated intermediate representation, and discussed the formal proof of its functional correctness in the Coq proof assistant in Chapter 5. Additionally to methodological improvements, among which the use of a Rely-Guarantee logic, we argued that our intermediate representation allowed us to be closer than other existing approaches to an actual executable formally verified GC– and hence to be able to embed it into a verified compiler. Indeed, as underlined in Chapter 5, a common practice consists in abstracting the algorithm of interest as a transition system. This simpler object is then proved correct. While this approach has several benefits, simplifying the reasoning and helping to focus on the logics of the algorithm, it pulls us away from a verified embedding into a compiler. In contrast, our approach strives for proving the code itself.

Using the Most General Client, one can build a refinement proof between an IR with implicit memory management and RTIR. We would then need to perform three kinds of refinements over the GC to make it executable: cleaning ghost code, coding iterators as low-level macros, and implementing abstract concurrent data structures natively supported by RTIR. While the two first tasks are essentially administrative, the third encompasses an inherently non executable layer of the intermediate representation.

This is however not some hazardous design flaw: the intermediate representation has naturally been designed with this aspect in mind. We explain in this chapter how RTIR can be soundly compiled towards an executable language by implementing its native data-structures with so-called *linearisable* operations. To do so, we first take a detour to lay out the two core notions we need. First, we go back in Section 6.1 to the context of this work, verified compilation, in order to introduce the notion of correctness we need: semantic refinement. We then consider in Section 6.2 the very reason those data-structures can be refined by reviewing the coherence criterion of linearisability. We finally get in Section 6.3 to the crux of our contribution by providing a formalised, sound, semantic foundation to a proof methodology introduced by Vafeiadis in his thesis [137]. In particular do we show how to instantiate the resulting meta-theorem on the core challenge we are concerned with the mark buffers. Finally, we discuss in Section 6.4 potential improvements to our results, as well as some related work.

6.1 VERIFIED COMPILATION: A CHAIN OF SIMULATIONS

We introduced in Chapter 1 the notion of verified compilation. Verified compilers are a mean to achieve end-to-end verification of software, as well as to address the empirical observation that modern

compilers *are* buggy [147]. To this day, great success has been achieved with respect to realistic, sequential, low-level languages such as C, through most notably the CompCert compiler [82].

The work described in this chapter fits within the long term goal of extending verified compilation to concurrent, managed program languages. The question asked really is: "how far are we from a verified compiler for a concurrent programming language embarking a rich runtime?" The specificity of this context influences the nature of the correctness guarantees we are concerned with, and the way we need to formulate them. In order to illustrate this specificity, we introduce in this section some terminology and tools related to verified compilation: first the general structure of a verified compiler, then the notion of correctness it carries on, and finally the proof tool of simulations.

6.1.1 *Optimisations and changes of IR: a chain of program transformations*

Compilers are often organised as a succession of transformations. The program under compilation undergoes successive optimisations, and may be transformed into different intermediate representations until reaching the target language. These intermediate representations are carefully chosen to ease specific optimisations and static analyses. Famous such intermediate representations include SSA (Single Static Assignment) and its variants [20, 24], whose LLVM's IR [78, 150] is a typed instance, as well as Haskell's core [115, 116].

While this methodology is general, it is even more salient when it comes to verified compilation. Indeed, planning on proving the whole compiler in a monolithic way would be ludicrous. Not only would handling all transformations at once be realistically impossible, but the resulting development would be impossible to maintain. Crucially, each transformation has to be carefully isolated and proved in a compositional manner. In the context of verified compilation, intermediate representations need therefore to be carefully chosen not only to enable optimisations, but also to ease their proof of correctness, as has been emphasised by Demange in her PhD thesis [27].

This fundamental design choice is immediately apparent in both major existing verified compilers. Figure 24 depicts the CompCert chain of transformations, while Figure 25 depicts the one of CakeML.

6.1.2 *Correct transformations: traces and semantic refinements*

A compiler is built as a chain of program transformations. A verified compiler is therefore built as a chain of composable proofs of correctness for each transformation.

Intuitively, a correct transformation should not modify the semantics of the program. Making this statement precise is however nontrivial. First, the transformation may go from one language to another,

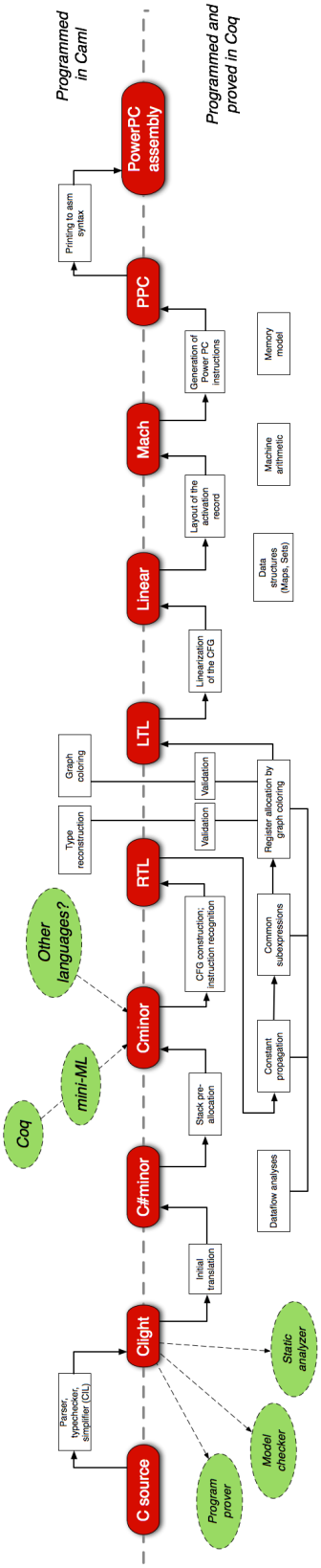


Figure 24: The CompCert chain of compilation, taken from [82].

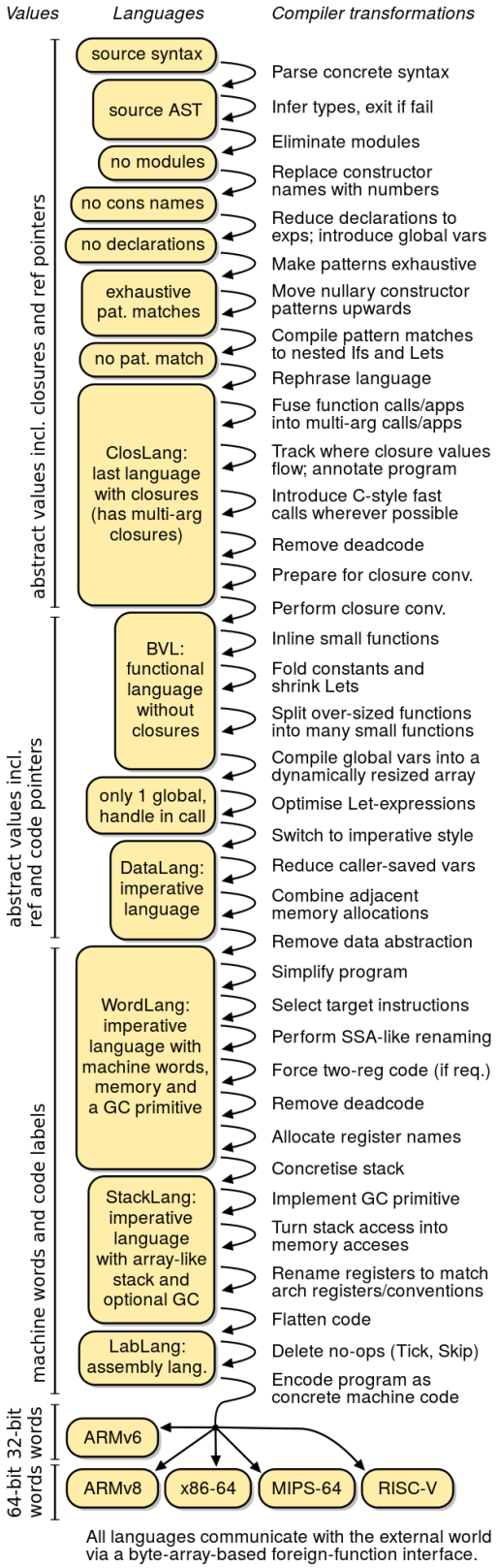


Figure 25: The CakeML chain of compilation, taken from [132].

hence changing the nature of a semantic state: we need to express how these different notions of states relate, and in particular what it means for them to embody the same semantics. Second, the relevant notion of semantics we are concerned with depends on the intent. We naturally at least want to terminate in states having the same semantics, but might or not want to be termination sensitive, or to preserve intermediate events such as prints.

In order to make things precise, we therefore need to fix which parts of an execution we choose to preserve, i. e. which set `events` of events the external world may observe. We refer to a list of such events as a behaviour. The choice of these events is naturally crucial to the notion of correctness we want to establish.

The behaviour of a program, or its set of behaviours in the case of a non-deterministic program, constitutes the crux of its operational semantics with respect with which the correctness of a transformation should be phrased. We therefore consider languages whose semantics collect traces of events.

Definition 4 (Language)**.** *A language is given by a type of states* `state`*, two subsets* $\mathcal{I}$ *and* $\mathcal{F}$ *of initial and terminal such states and a stepping relation* $\xrightarrow{\mathtt{beh}}$ *between states emits a list (potentially empty, denoted* τ*)* $\mathtt{beh}$ *of events. We write* $\xrightarrow{\mathtt{beh}}^{*}$ *for its finite reflexive transitive closure appending behaviours to collect them and* $\xrightarrow{\mathtt{beh}}^{\omega}$ *for its infinite transitive closure collecting events, where* $\mathtt{beh}$ *can be infinite.*

A transformation may in all generality impact a program in two ways: by introducing new behaviours, or by removing some.

From a safety standpoint, removing behaviours is always safe: if all behaviours of the original program were deemed safe, so will be the behaviours of the transformed program. On the contrary, adding an unforeseen behaviour can be harmful. The correctness of a program transformation is hence defined as an inclusion of behaviours, that we refer to as a *semantic refinement*:

Definition 5 (Semantic refinement)**.** *A transformation* $\mathcal{T}$ *from a language* $\mathcal{L}_1$ *to a language* $\mathcal{L}_2$ *is correct if it does not add any observable behaviour:*

$$\forall c \in \mathcal{L}_1,\ \mathtt{beh}_{\mathcal{L}_2}(\mathcal{T}(c)) \subseteq \mathtt{beh}_{\mathcal{L}_1}(c).$$

Assuming we fixed the same notion of events all along the compilation chain, we can compose the correctness of the successive program transformations to obtain the correctness of the compiler. Any safety property of programs which can be expressed in terms of their traces is guaranteed to be preserved by compilation.

6.1.3 *The bread and butter of semantic refinement: simulations*

While we strive for semantic refinement as an endgame result, we naturally need a stronger, inductive result to obtain it. The traditional

elementary brick used to prove a semantic refinement is a so-called *simulation*. We provide in this section a short reminder of this well-established technique, tailored to our specific needs.

6.1.3.1 *Behaviours*

Before introducing the notion of simulation, we need to formally define the behaviours. To do so, we consider languages whose semantics collect traces of events.

A language is then equipped with two kinds of potential behaviours.

Definition 6 (Behaviours)**.** *Assuming a language $\mathcal{L}$, and taking* $\mathtt{beh}$ *(respectively* $\mathtt{beh}_{\mathtt{inf}}$*) for a list (respectively stream) of events, two kinds of behaviours are considered:*

- *finite, correct execution:* $\mathtt{Term}(\mathtt{beh})$*;*
- *infinite, noisy execution:* $\mathtt{React}(\mathtt{beh}_{\mathtt{inf}})$*.*

Their intuitive semantics is made precise by defining admissible behaviours of a state.

Definition 7 (Reactive behaviours of states)**.** *The set* $\mathcal{R}\mathtt{eact}(s)$ *of reactive behaviours of a state* s *is co-inductively defined as:*

$$\begin{aligned}\forall \mathtt{beh} \neq \tau, \mathtt{beh}_{\mathtt{inf}}, s',\\ s \xrightarrow{\mathtt{beh}}{}^{*} s' &\implies \mathtt{React}(\mathtt{beh}_{\mathtt{inf}}) \in \mathcal{R}\mathtt{eact}(s')\\ &\implies \mathtt{React}(\mathtt{beh} {+\!\!+} \mathtt{beh}_{\mathtt{inf}}) \in \mathcal{R}\mathtt{eact}(s)\end{aligned}$$

Definition 8 (Admissible behaviours of states)**.** *The set* $\mathcal{B}\mathtt{eh}_S(s)$ *of admissible behaviours of a state* s *is inductively defined as:*

- $\forall s_f,\ s \xrightarrow{\mathtt{beh}}{}^{*} s_f \implies s_f \in \mathcal{F} \implies \mathtt{Term}(\mathtt{beh}) \in \mathcal{B}\mathtt{eh}_S(s)$
- $\mathtt{React}(\mathtt{beh}_{\mathtt{inf}}) \in \mathcal{R}\mathtt{eact}(s) \implies \mathtt{React}(\mathtt{beh}_{\mathtt{inf}}) \in \mathcal{B}\mathtt{eh}_S(s)$

Finally, the behaviours of a language are simply the collection of behaviours of the initial states.

Definition 9 (Admissible behaviours)**.** *The set* $\mathcal{B}\mathtt{eh}(\mathcal{L})$ *of admissible behaviours of a language $\mathcal{L}$ is inductively defined as:*

$$\forall s_i,\ s_i \in \mathcal{I} \implies \mathtt{beh} \in \mathcal{B}\mathtt{eh}_S(s_i) \implies \mathtt{beh} \in \mathcal{B}\mathtt{eh}(\mathcal{L})$$

6.1.3.2 *Simulations*

A semantic refinement of a program transformation $\mathcal{T}$ expresses that events are preserved all along the execution of a program transformed by $\mathcal{T}$. In order to prove this result, we need an elementary lemma

about one step of execution which is strong enough to hold inductively. This lemma is expressed with a simulation, and more specifically a *backward simulation*[1].

The idea behind a simulation is to exhibit a relation between the source state and the target state which is strong enough to entail the preservation of the behaviour by an elementary step, as well as to be itself stable by this step. Any step of the target program must be matched at the source level. This matching may take different forms depending on how close both programs are. In the simplest case, they might match one-to-one in the sense that any step by the target is directly matched by a single step in the source. We will consider a slightly more complex situation where the source might wait without stepping during some steps by the target. The motivation for this choice will be made clear in Section 6.3.

Formally, a backward simulation is defined by a relation satisfying three properties:

Definition 10 (Backward simulation)**.** *Given two languages $\mathcal{L}_1$, $\mathcal{L}_2$, a backward simulation is given by:*

- *a relation $\sim \subseteq \texttt{state}_1 \times state_2$*
- *an initialisation condition:*

$$\begin{aligned}&\forall s_2 \in \mathcal{I}_2,\\ &\exists s_1 \in \mathcal{I}_1 \textit{ such that } s_1 \sim s_2\end{aligned}$$

- *a finalisation condition:*

$$\begin{aligned}&\forall s_2 \in \mathcal{F}_2, \forall s_1 \textit{ such that } s_1 \sim s_2,\\ &\exists s_1' \in \mathcal{F}_1, s_1 \xrightarrow{\tau}{}^{*} s_1' \wedge s_1' \sim s_2\end{aligned}$$

- *a matching condition:*

$$\begin{aligned}\forall s_1,\ s_2,\ s_2',\ s_1 \sim s_2 \wedge s_2 \xrightarrow{e} s_2' \Rightarrow\\ \exists s_1',\ s_1' \sim s_2' \wedge s_1 \xrightarrow{e}{}^{*} s_1'\end{aligned}$$

The matching condition expresses the expected preservation of matching when the target state steps. The other two properties relate respectively initial and final states. The initialisation condition enforces that any target initial state should be matched to a source initial state. The finalisation condition expresses that any target final state is only related to source states able to silently reduce to a final state.

1 Various terminologies exist across different communities to refer to simulations. We follow here the compiler verification community [82]'s school of conventions.

6.1.3.3 *From simulations to preservation of behaviours*

By co-induction, one can derive the desired result of preservation of behaviours from the existence of a backward simulation. Intuitively, the finite case is easy since it perfectly admits the simulation as its inductive case. The reactive case is made possible since a non-silent step cannot be matched by the source without stepping. It therefore guarantees the co-inductive production condition.

Lemma 1. *If two languages $\mathcal{L}_1, \mathcal{L}_2$ admit a backward simulation, we have the following iteration of the matching condition:*

$$\forall s_1,\ s_1',\ s_2,\ s_1 \sim s_2 \wedge s_1 \xrightarrow{e}{}^{*}\ s_1',$$
$$\exists s_2',\ s_1' \sim s_2' \wedge s_2 \xrightarrow{e}{}^{*}\ s_2'$$

Proof. By straightforward induction over $\dot{\to}^*$ □

We then show by co-induction the preservation of reactive behaviours.

Lemma 2. $\forall s_1\ s_2\ \mathrm{beh}_{inf},\ s_1 \sim s_2 \implies \mathtt{React}(\mathrm{beh}_{inf}) \in \mathcal{R}\mathrm{eact}(s_2) \implies \mathtt{React}(\mathrm{beh}_{inf}) \in \mathcal{R}\mathrm{eact}(s_1)$

Proof. We proceed by co-induction.
There exists s_2',

$$s_2 \xrightarrow{\mathrm{beh}}{}^{*}\ s_2' \wedge \mathrm{beh}_{inf} = \mathrm{beh} +\!\!+ \mathrm{beh}_{inf}' \wedge \mathtt{React}(\mathrm{beh}_{inf}') \in \mathcal{R}\mathrm{eact}(s_2')$$

By Lemma 1,

$$s_1 \xrightarrow{\mathrm{beh}}{}^{*}\ s_1' \wedge s_1' \sim s_2'$$

Hence by coIH, $\mathtt{React}(\mathrm{beh}_{inf}') \in \mathcal{R}\mathrm{eact}(s_1')$
And by definition of $\mathcal{R}\mathrm{eact}$, we can conclude by wrapping a constructor on top of our inductive call. □

It remains only to take care of the initial and final states.

Theorem 1 (Preservation of behaviours)**.** *If two languages $\mathcal{L}_1, \mathcal{L}_2$ admit a backward simulation, then we have*

$$\mathcal{B}\mathrm{eh}(\mathcal{L}_2) \subseteq \mathcal{B}\mathrm{eh}(\mathcal{L}_1)$$

Proof. We have two cases to prove.

- First, assume $s_2 \in \mathcal{I}_2$, $s_2 \xrightarrow{\mathrm{beh}}{}^{*}\ s_{f_2} \in \mathcal{F}_2$.

By virtue of the initialisation condition, there exists $s_1 \in \mathcal{I}_1$ such that $s_1 \sim s_2$.
Using Lemma 1, we therefore get s_1' such that $s_1 \xrightarrow{\mathrm{beh}}{}^{*}\ s_1'$ and $s_1' \sim s_{f_2}$.
Finally by the termination condition, we get $s_{f_1} \in \mathcal{F}_1$ such that $s_1' \xrightarrow{\tau}{}^{*}\ s_{f_1}$, hence

$$\mathtt{Term}(\mathrm{beh}) \in \mathcal{B}\mathrm{eh}(\mathcal{L}_1)$$

• Suppose now that $s_2 \in \mathcal{I}_2$, $s_2 \xrightarrow{\mathrm{beh}_{\mathrm{inf}}}{}^{\omega}$
By virtue of the initialisation condition, there exists $s_1 \in \mathcal{I}_1$ such that $s_1 \sim s_2$.
We can conclude by Lemma 2.

□

6.2 LINEARISABILITY

Abstraction is at the hearth of any reasoning about software, be it formal or informal. In order to be able to build increasingly sizeable pieces of code, one needs to be able to abstract away from the operational semantics of a function. In a sequential world, this notion is well-known and defined since the earliest days of computer science. Things are however not as easy when it comes to concurrent programming.

6.2.1 *Traditional definition*

We briefly review in this section a few notions of coherence of concurrent objects the concurrent programming community historically introduced. In particular, Leslie Lamport introduced the notion of sequential consistency at the end of seventies [77], ten years before Herlihy and Wing defined linearisability [56]. A more detailed introduction to these notions may be found notably in *The Art of Multiprocessor Programming* monography by Herlihy and Shavit [57].

The crux of the problem lies in methods being non-atomic, hence stretching in time, and therefore happening concurrently. This phenomenon of overlapping methods immediately rises the subtle question of precedence of one method over the other. However before getting into this problem, one should not forget a fundamental principle: both methods should not corrupt each other.

Indeed, imagine an architecture where writing a signed integer takes two atomic methods: first the processor writes the bit encoding the sign of the integer, then it writes at once its absolute value. Now consider two threads trying to write concurrently at the same address respectively the values $+7$ and -3. While as expected the address could end up containing either intended values, two of the four possible executions could also lead up to incoherent results: -7 or $+3$.

This dire situation is excluded by enforcing a sanity principle stating that any execution should at least correspond to some execution where each method happened faithfully, one at a time.

Principle 1. *Any method call should appear to happen in a one-at-a-time order.*

With such a sanity ensured, one can actually reason about methods by thinking about their functional effect on the state. This baseline however tells us nothing about the order in which these operations should occur. In particular, a natural question is to wonder if in the case of a single thread, our definition coincides with what we are used to in a sequential context. To do so, we consider the notion of *program order*, i. e. the sequential order defined over operations of a given thread by the order in which they appear.

6.2.1.1 *A natural notion of coherence: sequential consistency*

While trying to order executions of methods across threads, a quite natural coherence constraint we might desire is to respect the program order. This intuitive principle is referred to as *sequential consistency* [77].

Principle 2 (Sequential consistency)**.** *Any method calls should appear to respect program order.*

The proper notion of sequential consistency as a whole consists in the pairing of Principles 1 and 2.

Sequential consistency is particularly appealing in that purely sequential executions are enforced to behave as they traditionally do. This principle however suffers from a major drawback: it is not compositional. Composing multiple sequentially consistent objects is not necessarily sequentially consistent. For instance, assume a data-structure implementing *First In First Out* (FIFO) queues equipped with two methods $\mathtt{push}$ and $\mathtt{pop}$. Consider the following execution of threads t_1 and t_2 manipulating two queues p and q, where x and y are the values pushed or popped from the data structures. We assume that both queues are initially empty, and that x and y hold distinct values.

$$\begin{array}{lllll} t_1 : & p.\mathtt{push}(x) & q.\mathtt{push}(x) & p.\mathtt{pop}(y) \\ t_2 : & q.\mathtt{push}(y) & p.\mathtt{push}(y) & q.\mathtt{pop}(x) \end{array}$$

Although p and q are both individually sequentially consistent, this execution resulting of the use of both queues is not. Indeed, suppose by absurd that there is a sequentially consistent ordering for this execution. Then for t_1 to return y when popping p and to respect the FIFO discipline, t_2's operation $p.\mathtt{push}(y)$ would have to happen before t_1's $p.\mathtt{push}(x)$. Similarly, t_1's operation $q.\mathtt{push}(x)$ would have to happen before t_2's $q.\mathtt{push}(y)$. But both those constraints cannot be combined with the program order enforced by principle 1 without creating a cycle: the scenario is absurd. Intuitively, the issue is that each object constraints the program order in non compatible ways.

6.2.1.2 *A compositional coherence criterion: linearisability*

Compositionality being crucial to build up complex concurrent systems, Herlihy and Wing introduced a new notion of coherence: lin-

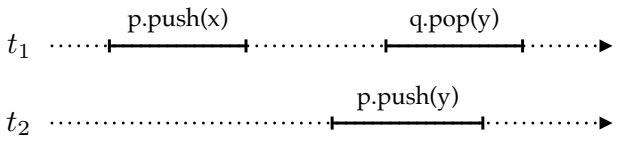


Figure 26: Example of a sequentially consistent, but non-linearisable, execution of two threads manipulating a FIFO queue.

earisability [56]. Linearisability is a compositional strengthening of the notion of sequential consistency. Intuitively, the principle goes as follows.

Principle 3 (Linearisability)**.** *Any method call should appear to take effect instantaneously at some moment between its call and return.*

Note that sequential consistency cares little for real-time order. Figure 26 depicts an example of a sequentially consistent execution of a FIFO queue manipulated by two threads, where the horizontal lines depicts real-time. The FIFO discipline is indeed respected if we consider the execution to amount for instance to the following sequence of operations: $\mathtt{p.push(y)}$; $\mathtt{q.pop(y)}$; $\mathtt{p.push(x)}$. Hence the execution is sequentially consistent, although it contradicts real-time: the operation pushing the value y logically happens before the one pushing the value x, although the real timeline establishes the converse.

In contrast, linearisability puts tighter constraints over executions. As with sequential consistency, method calls should respect their thread's program order. But additionally, two operations whose execution do not overlap cannot be reordered. Intuitively, linearisable operations are operations which appear to take place atomically somewhere in between their call and their return. The execution on Figure 26 is therefore not linearisable.

Making this notion formal is however more complex. The original definition is built over the notion of histories, i. e. sequences of method calls and returns. Such a history abstracts a concurrent execution. In particular, if every call to a method is immediately followed by its return in the history, methods' executions do not overlap: we say that the history is sequential. Histories fix a total order over all threads' calls and returns. Two histories are then said equivalent if for any thread t, their restrictions to the events issued by t are the same.

Among those histories, some may not make any sense with respect to the specification of our concurrent object. For instance, in the case of the FIFO queue considered in the previous section, we rejected executions resulting in a sequence of push and pop operations that does not respect the FIFO discipline. To generalise this idea, linearisability assumes we have a specification deciding if a sequence of operations over an object is legal. A history H is then said legal if for any object, any sub-sequence of H made by restricting H to this object's operations is legal.

Proving that a history H is linearisable therefore essentially consists in finding an equivalent, legal, sequential history S by reordering H: being sequential encodes the fact that all methods appear atomically. To ensure that we did not allow method calls to be reordered out of their frame of call and return, we enforce an additional constraint: method calls take place in the same order both in H and S.

The last technical detail to consider is that an execution may include pending calls, i. e. ongoing executions of methods which did not return yet. The definition of linearisability therefore imposes S to be equivalent to some alternate version of H where pending calls may have been either removed, or completed.

Wrapping everything together, linearisability is defined as follows.

Definition 11 (Linearisability)**.** *A history* H *is linearisable if its pending invocations can be completed and/or removed into a history* H′ *admitting a legal sequential history* S *such that:*

- H′ *is equivalent to* S*;*
- *method calls admit the same order in* H *and* S*.*

A concurrent object is then said to be linearisable if all its histories are linearisable.

Linearisability clearly entails sequential consistency, and can be shown to be compositional. For these reasons, it has been the golden standard in concurrent programming.

This definition is however unsatisfactory for formal reasoning, and in particular for our case of use. First, establishing linearisability under this form is non-local since we have to handle global traces of executions. Rely-Guarantee logic having provided us with thread local reasoning, we would like to recover this when establishing that a concurrent data structure is linearisable. Second, histories are non-operational abstractions. We need to close the gap to our operational semantics. Third, the refinement of RTIR through the implementation of its native abstract data-structure is meant to take place among a verified compiler. We therefore need to link the linearisability of the data-structure to a notion of semantic refinement.

In order to address these three issues, we argue that the definition of linearisability in terms of history is unnecessary. Instead, we directly rephrase linearisability in terms of semantic refinement and reduce it to proof obligations phrased in terms of Rely-Guarantee reasoning, providing strong semantic foundations to a methodology introduced by Vafeiadis in his PhD thesis [137]. We present our approach and our main result in the remainder of this section.

6.2.2 *Linearisability as semantic refinement*

We laid out in Section 6.1 the context where our work takes place: we seek to embed our results in a verified compiler, where the correct-

ness standard is the one of semantic refinement, built upon simulations. While this refinement exists because the data-structure is linearisable, the theorem we want is the so-called atomicity refinement: the implementation may be soundly replaced by an atomic, abstract data-structure.

The first authors who formally addressed the distance between the notion of history-based linearisability, golden standard in the distributed community, and the one of observational refinement, more common in the programming language community, were Filipović et al. [43] in 2009. Under assumption of freedom interference [57, p.198], i. e. that data-structures can only be modified via their methods, they prove that history-based linearisability is indeed equivalent to the existence of an atomic refinement.

Still, their work suffers from some limitations: the approach is neither operational, since the semantics considered is an action trace model [15], nor mechanised, and only observe the initial and finite values of the program. However, this work builds a strong case to free ourselves from Herlihy and Wing's definition and directly target atomic refinement.

Following this intuition, we explicitly rephrase linearisability in terms of atomic refinement. To this end, we consider a concurrent language $\mathcal{L}$ as our target language, whose syntax and semantics will be made precise in Section 6.3.2. Our source language is $\mathcal{L}^\sharp$, a language identical to $\mathcal{L}$ but extended with so-called *abstract data structures*, admitting a set $\mathcal{I}$ of atomic methods. We unwind the methodology all along this chapter through two examples: a toy example illustrating the principle, and the main data-structure we need for our garbage collector.

Example 1 (Spinlock – Abstract data-structure, operations)**.** *A common way to work with coarse-grain concurrency is to rely on locks. When a thread wants to access a shared data, it first tries to acquire a lock that only one thread can hold at once. By restraining threads from accessing the data until they hold the lock, we ensure that no interference will disturb the thread. Once a thread is done interacting with the shared data, it releases the lock for others to be able to take their turn.*

A client manipulating carelessly such locks could face two issues. First, a thread could access the shared data without holding the lock, and more generally two concurrent accesses could still occur, negating the utility of the lock. Second, two or more threads could wait for each other to release the lock, preventing any progress to be made: we say a deadlock has been reached.

Being able to abstract away from the concrete implementation of the lock may simplify reasoning to prove for instance that a program is indeed deadlock-free. We can achieve this through our methodology, proving that the lock is linearisable, i. e. can be atomically refined.

Suppose therefore that $\mathcal{L}^\sharp$ offers abstract locks, each providing two methods: $\mathcal{I} = \{\mathtt{acquire}, \mathtt{release}\}$. At the level of $\mathcal{L}^\sharp$, an abstract lock can be seen as a simple two valued data-type:

$$\mathtt{Lock} := \mathtt{Locked} \mid \mathtt{Unlocked}.$$

The abstract semantics of both lock's methods are straightforward:

$$[\![\mathtt{acquire}]\!]^\sharp(\mathtt{Unlocked}, v) = (\mathtt{Locked}, \mathtt{Null})$$
$$[\![\mathtt{release}]\!]^\sharp(\mathtt{L}, v) = (\mathtt{Unlocked}, \mathtt{Null})$$

In both cases, the input value is discarded, and the return value is set to Null, being irrelevant for those methods. In order to acquire the lock, we check that the lock was unlocked: if so we switch its value to `Locked`*, otherwise the semantics is blocking. Releasing the lock is similar, but never blocking. Indeed since the client should only call the release method after it acquired the lock, no test over its value should be needed.*

Example 2 (Buffers – Abstract data-structure, operations)**.** *The second case study we consider is the one of buffers, which* RTIR *took for granted. The compiler allocates a part of the heap to the collector for the buffers it needs. The notion of abstract buffer we consider here is therefore a queue of bounded size SIZE, that we model by a simple mathematical list of values.*

$$\mathtt{Buffer} := \mathtt{Nil} \mid v :: \mathtt{Buffer}.$$

A buffer is pushed on one end of the list, and popped off from the other end. As expected, buffers provide the four methods used in the code of the GC: $\mathcal{I} = \{\mathtt{isEmpty}, \mathtt{top}, \mathtt{pop}, \mathtt{push}\}$. Their respective abstract semantics are:

$$\begin{aligned}
[\![\mathtt{isEmpty}]\!]^\sharp(\mathtt{ab}, v) &= (\mathtt{ab}, 1) \textit{ if } \mathtt{ab} = \mathtt{Nil} \\
&= (\mathtt{ab}, 0) \textit{ otherwise} \\
[\![\mathtt{top}]\!]^\sharp(x :: \mathtt{ab}, v) &= (x :: \mathtt{ab}, x) \\
[\![\mathtt{pop}]\!]^\sharp(x :: \mathtt{b}, v) &= (\mathtt{b}, \mathit{Null}) \\
[\![\mathtt{push}]\!]^\sharp(\mathtt{b}, v) &= (\mathtt{b}^{++}[v], \mathit{Null}) \textit{ if } |\mathtt{b}| < \mathit{SIZE} - 1.
\end{aligned}$$

Those semantics are mostly straightforward. Once again, isEmpty, top *and* pop *discard their input, while* pop *and* push *return the default value* Null*. The method* isEmpty *returns the value* 0 *or* 1 *to indicate whether the abstract buffer is equal to* Nil*, and leaves it untouched. Both the methods* top *and* pop *have the peculiarity to be blocking if the buffer is empty. Buffers being of bounded size, the* push *method may also block, if the size of the abstract buffer is greater or equal than SIZE.*

Given an implementation for each atomic method in $\mathcal{L}$, we consider a program transformation, meant to be embedded in a verified compiler. This transformation, $\mathtt{compile}\langle\mathcal{I}\rangle \in \mathcal{L}^\sharp \to \mathcal{L}$, replaces the abstract data structures with their fine-grained concurrent implementation in

$\mathcal{L}$. Our goal is to prove for linearisable implementations of the abstract data-structure that this compiling pass is correct, in the sense that it preserves the observable behaviours of source programs. The sought theorem is therefore of the form

$$\forall p \in \mathcal{L}^\sharp,\ \mathtt{beh}(\mathtt{compile}\langle\mathcal{J}\rangle(p)) \subseteq \mathtt{beh}(p). \tag{1}$$

Example 3 (Spinlock – Concrete implementation)**.** *The abstract spinlock can be implemented using a single boolean field* `flag`*. The value of the flag denotes the state of the lock. The implementation of the* `acquire` *and* `release` *methods are given on Figure 27.*

Releasing the lock simply consists in setting the flag to 0*. Indeed, the corresponding abstract semantics was also straightforward. The abstract* `acquire` *method however was blocking when the lock was* `Locked`*. This behaviour is matched by the use of a waiting loop. Acquiring the lock requires the use of a compare-and-swap instruction. If the lock was indeed available, we simply set the flag to* 1*. Otherwise, we loop and retry until the lock is available.*

```
def acquire() ::=
  ok = 0
  do {
   ok=cas(this.flag,0,1)
  } while (ok == 0)
  return
```

```
def release() ::=
  this.flag = 0
  return
```

Figure 27: Spinlock in $\mathcal{L}$.

Example 4 (Buffers – Concrete implementation)**.** *We turn to the concrete implementation of the buffers. The fine-grained implementation we prove is similar to that of Domani et al. [35], except that we use bounded-sized buffers. Buffers are objects with three fields (see Figure 28). Field* `data` *contains a reference to an array of fixed size SIZE, containing the elements of the buffer. Two other fields,* `next_read` *and* `next_write`*, indicate the bounds, within the array, of the effective content of the buffer. Field* `next_read` *contains the array index from which to read, while* `next_write` *contains the index of the first free slot in the array.*

A buffer is empty if and only if `next_read` *and* `next_write` *are equal. Pushing a value on a buffer consists in writing this value in the array, at position* `next_write`*, and then incrementing* `next_write`*. Conversely, popping a value from a buffer is done by incrementing* `next_read`*. To consult its top value, one reads the array element at position* `next_read`*. In fact, the* `data` *array can be populated in a modulo fashion (see right example in Figure 28). The code for implementing buffers is given in Figure 29, and follows the above principles. Our core language does not include proper arrays, but we encode them with appropriate macros. The code is blocking when trying to pop on an empty buffer, or trying to push on a full buffer, as is the case*

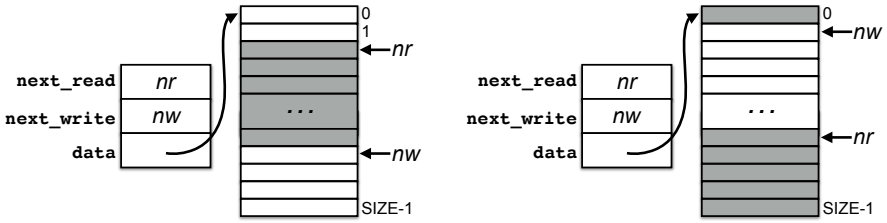


Figure 28: Concrete buffers layout (examples). Elements contained in the buffer are coloured in grey. Example on the right shows how the array is populated circularly.

```
def top() ::=
  nr = this.next_read
  nw = this.next_write
  assume(nr≠nw) // buffer ≠ Nil
  d = this.data
  res = d[nr]
  return res

def pop() ::=
  nr = this.next_read
  nw = this.next_write
  assume (nr≠nw) // buffer ≠ Nil
  nr = (nr+1) mod SIZE
  this.next_read = nr
  return
```

```
def isEmpty() ::=
  nr = this.next_read
  nw = this.next_write
  return (nw==nr)

def push(v) ::=
  nw = this.next_write
  nr = this.next_read
  d = this.data
  d[nw] = v
  nw = (nw+1) mod SIZE
  assume (nr≠nw) // no overflow
  this.next_write = nw
  return
```

Figure 29: Buffers in $\mathcal{L}$.

for the abstract version. This is no limitation in practice: the size of buffers is chosen at initialisation time, and can be upgraded at will.

Note that the implementation avoids the use of any synchronisation mechanism. This would naturally be unsound in general, but relies on the client respecting a protocol: the queues are assumed to be single-writer, single-reader. This protocol matches the one followed by the garbage collector: every shared buffer is only read by the collector, and only written to by its associated mutator. Crucially, our methodology is indeed able to encode such protocols, as described in Section 6.3.8.

As we explained in Section 6.1, proving a theorem like (1) is done with a simulation: for any execution of the target program, we must exhibit a matching execution of the source program. Between $\mathcal{L}$ and $\mathcal{L}^\sharp$, the simulation is however particularly difficult to establish.

We illustrate the general situation with Figure 30. Execution steps labelled with $\lightning$ are those where the effect of a method in $\mathcal{I}$ becomes visible to other threads, and thus determines the behaviour of other methods in $\mathcal{I}$ executed concurrently.

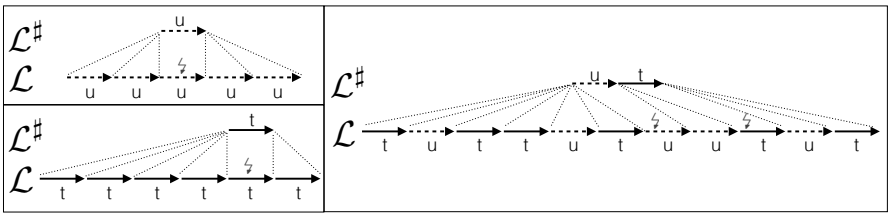


Figure 30: Intra and inter-thread matching step relations.

At the intra-thread level (Figure 30, left), we need to relate several steps of a thread in the target program to a single step in the source. The situation is even more difficult at the inter-thread level (Figure 30, right): the interleaving of threads at the target level (first u, then t in the example) *must* sometimes be matched at the source level by another interleaving (first t then u in the example). Indeed, it all depends on which thread will be the first to execute its $\lightning$ step in the concrete execution. The matching step for a given thread hence depends on the execution of its environment.

Our main result, that we present in Section 6.3.9, removes this difficulty by establishing, under some hypotheses, a *generic* simulation that entails semantic refinement. This is a meta-theorem that we establish once and for all, independently of the abstract data-structures available in $\mathcal{L}^\sharp$.

6.3 PROVING LINEARISABILITY THROUGH RELY-GUARANTEE REASONING

As argued in Section 6.2, we would like to avoid having to use the history-based definition of linearisability. This definition is indeed intrinsically non-local and hardly operational. Moreover, we argued that the notion we actually seek is atomicity refinement, a notion that has been proved equivalent to linearisability in the past.

Proving directly atomicity refinement is however not much more appealing. Following Section 6.1.3, refinements are consequences of simulations. But building simulations is a complex and tedious task, especially in our particular setup, as highlighted in Figure 30.

The solution we developed has therefore been to rephrase, formalise and provide with strong semantic foundations a methodology first introduced by Vafeiadis in his PhD [137]: proving linearisability through Rely-Guarantee reasoning.

6.3.1 *Intuitive idea*

Linearisable operations appear to take place atomically, somewhere between their invocation and their return. The atomic operation at

which this happens corresponds to a program point that we refer to as a *linearisation point*. Identifying these linearisation points in the implementations is the first key towards proving linearisability.

Indeed, while Rely-Guarantee reasoning embodies no native support to capture linearisability, Vafeiadis devised the idea to work upon *hybrid* methods, explicitly annotated with their linearisation points. These hybrid methods are the concrete, implemented ones, except that they proceed upon a hybrid state. Additionally to the concrete, expected, component of the state, we also store the value of the abstract data-structure. The method being annotated with linearisation points, we can give them semantics: since they characterise the moment when the effect becomes visible, they should trigger the abstract semantics of the method at the same instant.

Example 5 (Spinlock – Linearisation annotation)**.** *Figure 31 shows once again the code for the spinlock implementation, but annotated with explicit linearisation points.*

The effect of releasing a lock is no mystery: the concrete implementation is already atomic, its instruction is therefore obviously the linearisation point. We encode this annotation by adjoining to the instruction another instruction, `Lin(true)`*. The boolean* `true` *simply states that this instruction unconditionally linearises the method when executed.*

On the contrary, the `acquire` *method will only perform its visible effect once the compare-and-swap operation has succeeded. This operation is thus the linearisation point, but only acts as such when the variable* `ok`*, holding the result of the compare-and-swap, contains the value* `1`*.*

```
def acquire() ::=
  ok = 0 ;
  do {
   atomic ⟨
    ok=cas(this.flag,0,1)
    Lin(ok==1) ⟩
  } while (ok == 0)
  return
```

```
def release() ::=
  atomic ⟨
    this.flag = 0
    Lin(true) ⟩
  return
```

Figure 31: Spinlock in $\mathcal{L}^{\flat}$.

Example 6 (Buffers – Linearisation annotations)**.** *Spinlock's operations in the previous example demonstrate the fact a linearisation point may only act as such during an execution granted a condition is satisfied. However their identification was straightforward in the sense that they happened at the only time we wrote in the shared memory.*

Buffers' annotations, as given on Figure 32 show more interesting concerns.

The `top` *operation does not perform any modification of the shared data-structure. One could therefore think that any program point would fit as a*

```
def top() ::=
  nr = this.next_read
  nr = this.next_write
  assume(nr≠nw) // buffer ≠ Nil
  d = this.data
  res = d[nr]
  Lin(true)
  return res

def pop() ::=
  nr = this.next_read
  nw = this.next_write
  assume (nr≠nw) // buffer ≠ Nil
  nr = (nr+1) mod SIZE
  atomic ⟨ this.next_read=nr ;
           Lin(true) ⟩
  return
```

```
def isEmpty() ::=
  nr = this.next_read
  nr = this.next_write
  Lin(true)
  return (nw==nr)

def push(v) ::=
  nw = this.next_write
  nr = this.next_read
  d = this.data
  d[nw] = v
  nw=(nw+1) mod SIZE
  assume (nr≠nw) // no overflow
  atomic ⟨ this.next_write=nw ;
           Lin(true) ⟩
  return
```

Figure 32: Buffers in $\mathcal{L}^{\flat}$.

linearisation point. If we stick to the only argument that linearisation occurs when the effect of the method takes place, this reasoning would be sound. However, the same way the linearisation point characterises the point in time at which the effect of the method over the environment takes place, it also determines the point in time where the environment can no longer influence the result of the operation. Linearisation in the top *operation therefore only takes place after the value of* d[nr] *has been read. Note also that if we had decided to return a default value if the buffer is empty, i.e. replaced the* assume *operation by a test returning if succeeding, we would have had two linearisation points: one when the test is performed, linearising the method if the test succeeds, and the current one. The* isEmpty *operation is similar.*

Both the pop *and* push *operations perform their linearisation point at the end of their executions, when they move the relevant pointer indicating either the beginning or the end of the buffer in the linked list. Note that they do manipulate the shared data-structure beforehand. However, the interpretation of the structure is to consider only the elements held between the* this.next_write *and* this.next_read *pointers as being part of the buffer. The changes they may perform to the remainder of the structure therefore cannot be observed by the environment before the pointer is moved.*

Vafeiadis has introduced a hybrid entity that can be reasoned about using Rely-Guarantee in order to prove linearisability. Intuitively, by expressing a coherence invariant relating an abstract data-structure to its implementation, one can use Rely-Guarantee reasoning to ensure that any execution of a method will trigger exactly one linearisation point, and that the coherence between the abstract and concrete

views of the structure will remain. While the approach is elegant, it is not formally linked neither to the history-based definition of linearisability, nor to the one we are interested in, semantic refinement. We propose here such a formalised semantic foundation to the methodology by automatically deriving an atomicity refinement from proof obligations expressed in RG.

6.3.2 *Languages*

As explained, we consider in this work a target language $\mathcal{L}$, as well as a source language $\mathcal{L}^\sharp$ including $\mathcal{L}$, as well as an abstract data-structure equipped with atomic operations. However, as already hinted, we need to introduce a third, hybrid, language $\mathcal{L}^\flat$ in order to apply Rely-Guarantee reasoning. This new language contains all features of $\mathcal{L}$, but also linearisation annotations, and a state enhanced with ghost elements used to conduct the proof of linearisability. While programs in $\mathcal{L}^\flat$ cannot explicitly call abstract methods, their semantics is implicitly embedded in the linearisation annotations.

To lighten the presentation, we will focus on one language when the underlying concept is common to all three languages. This language is $\mathcal{L}^\flat$ enhanced with abstract methods, that includes all features. We shall keep in mind that source programs in $\mathcal{L}^\sharp$ do not include any linearisation instrumentation, while target programs in $\mathcal{L}$ do not contain abstract method calls nor linearisation instrumentation.

$\mathcal{L}^\flat$ is quite similar to RTIR but for a few enhancements. First, abstract data-structures are made explicit: the language is parameterised by one such structure[2] and its operations, including a constructor. Additionally, $\mathcal{L}^\flat$ has explicit support for method calls to smoothen the definition of the transformation, substituting an abstract method call by a concrete one. Finally, as argued, it supports linearisation instrumentation.

To sum up the language, $\mathcal{L}^\flat$ is a concurrent imperative language, with no dynamic creation of threads. It is dynamically typed, and features a simplified object model: objects in the heap are just records, and rather than virtual method calls, the current object – the object whose method is being called – is an extra function argument, passed in the reserved variable `this`. In the sequel, *Var* is a set of variables identifiers, method names range over $\mathtt{m} \in \mathit{Methods}$, and fields identifiers range over $\mathtt{f} \in \mathit{Fields}$.

2 The current development does not support several different data-structures, but the process of refinement could be chained with very little adaptation.

$$
\begin{array}{rrcl}
\text{<expr>} & e & ::= & n \mid \mathtt{null} \mid x \mid e + e \mid - e \mid e \bmod n \mid \ldots \\
\text{<bexpr>} & b & ::= & \mathtt{true} \mid e == e \mid e <> e \mid b \;\&\&\; b \mid b \,\|\, b \mid !b \\
\text{<comm>} & c & ::= & \bullet \mid \mathtt{assume}(b) \mid \mathtt{print}(e) \\
 & & & \mid x = e \mid x = y.f \mid x.f = y \\
 & & & \mid x = \mathtt{new}(f,\ldots,f) \mid \mathtt{return}(e) \mid x = y.m(z) \\
 & & & \mid c \,;\, c \mid c + c \mid \mathtt{loop}(c) \mid \mathtt{atomic}\ \langle c \rangle \\
\text{<comm>}^\sharp & c^\sharp & ::= & c \mid x \mathrel{*}= \#y.m(z) \\
\text{<comm>}^\flat & c^\flat & ::= & c \mid \mathtt{Lin}\ (b)
\end{array}
$$

Figure 33: Language syntax.

6.3.2.1 *Values and Abstract Data Structures*

We use the domain of values $Val = \mathbb{Z} + Ref + Null$, where Ref is a countable set of references. A central notion in the language is the one of abstract data structure, which are specified with an atomic specification. All our development and our proofs are parameterised by an abstract data structure specification. It could naturally be in particular either of the examples introduced, abstract locks or buffers.

Definition 12 (Abstract data structure specification)**.** *An abstract data structure is specified by a tuple* $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$ *where* $A^\sharp$ *is a set of abstract objects;* $\mathcal{I} \subseteq$ *Methods is a set of abstract methods identifiers, whose atomic semantics is given by the partial map* $[\![.]\!]^\sharp \in \mathcal{I} \to (A^\sharp \times Val) \hookrightarrow (A^\sharp \times Val)$*, taking as inputs an object and a value, and optionally returning an updated object and a value;* $\mathcal{P} \subseteq$ *Fields reserves private field identifiers for the concrete implementation of abstract methods in* $\mathcal{I}$*.*

Abstract objects in $A^\sharp$ are the possible values that an instance of a data structure can take. We use private fields to express the property of *interference freedom* from Herlihy and Shavit [57]. Namely, client code can only use public fields in $Fields \setminus \mathcal{P}$, and concrete implementations of abstract methods in $\mathcal{I}$ use private fields only. Note that allowing the abstract semantics to be a partial map is crucial to allow refinement of implementations which loops until a condition is satisfied. For instance, a pop method over a queue which would not return a default value when the queue is empty, but rather wait until another thread pushes a value would be simulated by an abstract semantics for the pop method which is blocking over the empty queue.

Example 7 (Spinlock – Abstract data structure specification)**.** *In the spinlock example, we therefore have the previously introduced objects:* $A^\sharp := \mathtt{Locked} \mid \mathtt{Unlocked}$*, and abstract methods* $\mathcal{I} = \{\mathtt{acquire}, \mathtt{release}\}$*. Lock implementations use the single private field* $\mathcal{P} = \{\mathtt{flag}\}$*.*

6.3.2.2 *Language Syntax*

In the sequel, we fix an abstract data structure specification $(A^\sharp, \mathcal{I}, [\![.]\!]^\sharp, \mathcal{P})$. The syntax of the language $\mathcal{L}^\flat$ is detailed on Figure 33. The language provides constants (n, $\mathtt{null}$, $\mathtt{true}$), local variables (x,y,z ...), and arithmetic and boolean expressions (e, b). Regular commands (c) are standard, and common to the three languages. They include $\bullet$ (skip), an $\mathtt{assume}(e)$ statement, a $\mathtt{print}(e)$ instruction that emits the observable value of e, variable assignment of an expression, fields reads and updates, record allocation, non-deterministic choice ($+$), loops, and atomic blocks $\mathtt{atomic}\ \langle c \rangle$. Concrete method calls are written $x = y.m(z)$.

Some instructions are specific to a language level. In $\mathcal{L}^\sharp$, abstract method calls on an abstract object are written $x *= \#y.m(z)$. For any $m \in \mathcal{I}$, such a call in a $\mathcal{L}^\sharp$ program is compiled to a concrete call $x = y.m(z)$ in the $\mathcal{L}$ program. In $\mathcal{L}^\flat$, the $\mathtt{Lin}(b)$ instruction is used to annotate a linearisation point.

Finally, a client program is defined by a map from method names in $\mathit{Methods} \setminus \mathcal{I}$ to their command, and a map from thread identifiers to their initial command. In the sequel, we will write $m.comm$ for getting the command of method m, leaving the underlying program implicit.

6.3.3 *Semantics*

We present here the essential elements of our semantics, and refer the reader to the formal development[3] for full details[4].

We assume a standard semantics $[\![\cdot]\!]$ for expressions, omitted here. Abstract objects are stored in an abstract heap, ranged over by $h^\sharp \in H^\sharp = \mathit{Ref} \to A^\sharp$. At the concrete level, abstract objects are implemented by regular, concrete objects, living in a concrete heap $h \in H = (\mathit{Ref} \times \mathit{Fields}) \to \mathit{Val}$. A shared memory, ranged over by $\sigma \in H^\sharp \times H$ is made of an abstract heap and a concrete heap.

An intra-thread state $ts = \langle m, c, l, ls \rangle$ includes the name of the current method m, a current command c, a local environment $l \in \mathit{Lenv} = \mathit{Var} \to \mathit{Val}$, and a linearisation state $ls \in \mathit{LinState}$, that we explain below. The intra-thread operational semantics, partially shown in the top four rules of Figure 34, is a transition relation $\cdot \xrightarrow{\cdot} \cdot$ on intra-thread states. It is labelled with observable events ranged over by o. An observable event is either a numeric value or the silent event τ.

The print instruction (rule PRINT) is the only one that emits an observable value, namely the value of the expression that is printed.

3 http://www.irisa.fr/celtique/ext/simulin/

4 In our formal development, we use a continuation-based semantics to handle atomic blocks and method calls. This has proven to lighten the mechanisation of many proofs, by removing any recursion from the small step semantics. It also allowed for reasoning of intermediate states of atomic executions.

$$\frac{[\![e]\!]l = v \quad m \notin \mathcal{I}}{(\langle m,\mathsf{print}(e), l, ls\rangle, \sigma) \xrightarrow{v} (\langle m, \bullet, l, ls\rangle, \sigma)}\ \textsc{Print}$$

$$\frac{\begin{array}{c} l(y) = r \quad h^\sharp(r) = a \quad l(z) = v \\ m' \in \mathcal{I} \quad [\![m']\!]^\sharp(a, v) = (a', v') \\ ts' = \langle m, \bullet, l[x \mapsto v'], ls\rangle \quad \sigma' = (h^\sharp[r \mapsto a'], h) \end{array}}{(\langle m, x *= \#\, y.m'(z), l, ls\rangle, (h^\sharp, h)) \xrightarrow{\tau} (ts', \sigma')}\ \textsc{Acall}$$

$$\frac{\begin{array}{c} ls' = \text{ if } m' \in \mathcal{I} \text{ then } \mathsf{Before}(r, v) \text{ else } \mathsf{Nolin} \\ l(y) = r \quad l(z) = v \quad l' = [m'.\mathsf{this} \mapsto r,\ m'.\mathsf{arg} \mapsto v] \end{array}}{(\langle m, x = y.m'(z), l, \mathsf{Nolin}\rangle, \sigma) \xrightarrow{\tau} (\langle m', m'.\mathsf{comm}, l', ls'\rangle, \sigma)}\ \textsc{Ccall}$$

$$\frac{\begin{array}{c} [\![b]\!]l = \mathsf{true} \quad m \in \mathcal{I} \\ h^\sharp(r) = a \quad [\![m]\!]^\sharp(a, v) = (a', v') \\ ts' = \langle m, \bullet, l[x \mapsto v'], \mathsf{After}(r, v, v')\rangle \quad \sigma' = (h^\sharp[r \mapsto a'], h) \end{array}}{(\langle m, \mathsf{Lin}(b), l, \mathsf{Before}(r, v)\rangle, (h^\sharp, h)) \xrightarrow{\tau} (ts', \sigma')}\ \textsc{LinTrue}$$

$$\frac{[\![b]\!]l = \mathit{false}}{(\langle m, \mathtt{Lin}(b), l, \mathsf{Before}(r, v)\rangle, \sigma) \xrightarrow{\tau} (\langle m, \bullet, l, \mathsf{Before}(r, v)\rangle, \sigma)}\ \textsc{LinFalse}$$

$$\frac{\gamma(t) = ts \quad (ts, \sigma) \xrightarrow{o} (ts', \sigma') \quad \forall t' \neq t,\ \neg \mathit{inAtomic}(\gamma(t'))}{(\gamma, \sigma) \xrightarrow{o} (\gamma[t \mapsto ts'], \sigma')}\ \textsc{Intl}$$

Figure 34: Semantics (excerpt).

Neither the local state nor the shared memory are modified by this instruction. Print instructions are only allowed outside abstract methods implementations.

An abstract method call (rule ACALL) $x *= \#\, y.m(z)$ is executed according to the abstract semantics $[\![m]\!]^\sharp$, and modifies only the abstract heap.

Concrete method calls (rule CCALL) behave as expected, but additionally manage the local linearisation state. This linearisation state notably keeps track of whether the execution of the current method is before its linearisation point (Before) or not (After). It also keeps track of the reference to the abstract object on which the method is called, the value of the call argument, and the value returned by its abstract semantics. This information is instrumental in proving that the method execution may be faithfully simulated by its abstract alternative when the linearisation point is reached.

Initially when executing an concrete method call, the linearisation state is set to Nolin. When control transfers to a method in $\mathcal{I}$ through

a concrete method call, the linearisation state changes from Nolin to Before (see rule CCALL). It switches to After when executing a Lin instruction (rule LIN), and then back to Nolin on method return. Linearisation states are used in the simulation proof, and instrument $\mathcal{L}^\flat$ only.

At the $\mathcal{L}^\flat$ level, the Lin instruction also accounts for the effect on the abstract heap of concrete methods in $\mathcal{I}$: it performs the abstract atomic call $[\![m]\!]^\sharp$ to the enclosing method m, updating the local environment and abstract heap.

The interleaving of threads is handled in rule INTL,with relation $(\gamma, \sigma) \xrightarrow{o} (\gamma', \sigma')$ between global states (γ, σ), where γ maps thread identifiers to thread local states and σ is a shared memory. Mutual exclusion between atomic blocks is ensured by the $\neg$*inAtomic* side condition.

Finally, we need to introduce our program behaviours. Following Section 6.1.3, they are generated by the values printed by the program taken for events. They therefore are defined on top of the interleaving semantics, as expressed by the following definition.

Definition 13 (Program behaviour)**.** *The observable behaviour of a program* p *from an initial shared memory* σ_i*, written* $\mathtt{beh}(p, \sigma_i)$*, is a set of either finite traces of values emitted by a finite sequence of transitions or an infinite trace of values emitted by an infinite sequence of transitions.*

6.3.4 *Definition of the transformation*

We introduced in Section 6.2.2 the transformation compile, for which we want a refinement. This transformation should take any client in $\mathcal{L}^\sharp$ and compile it into $\mathcal{L}$ by substituting its abstract method calls for their concrete counterpart. However, due to the intermediate language $\mathcal{L}^\flat$ we need to reason about, we actually split the compile function into two successive transformations: $\mathtt{compile} = \mathtt{clean} \circ \mathtt{concretize}$.

First, a client $p \in \mathcal{L}^\sharp$ is transformed into $\mathtt{concretize}\langle\mathcal{I}\rangle(p) \in \mathcal{L}^\flat$ where abstract methods are implemented by hybrid, annotated code. This annotated code is naturally supplied by the programmer of the library. Then a second compilation phase $\mathtt{clean}\langle\mathcal{I}\rangle \in \mathcal{L}^\flat \to \mathcal{L}$ takes care of removing Lin instructions in the target program.

We prove in Coq that the compiler is correct, providing that hybrid methods in $\mathcal{I}$ are proved correct *w.r.t.* a RG specification, RGspec, that we carefully define, in terms of $\mathcal{L}^\flat$ semantics, to prove, via the aforementioned simulation:

$$\mathsf{RGspec}\,(\mathcal{I}) \Longrightarrow \forall p \in \mathcal{L}^\sharp,\ \mathtt{beh}(\mathtt{compile}\langle\mathcal{I}\rangle(p)) \subseteq \mathtt{beh}(p)$$

RGspec is a set of proof obligations under the form of RG proofs of the instrumented methods, as well as constraints over their components: its precise definition is exposed in section 6.3.9. The judgement

embeds the requirements sufficient to simulate the hybrid method by its abstract counterpart, as well as to show that Lin instructions can be safely removed by the `clean` phase. This ensures that, despite their operational nature, Lin instructions are only passively instrumenting the program and its semantics.

6.3.5 *Using our result*

Before laying down our result, we recapitulate its aim, and detail how it can be used from a client's perspective. The intent is to provide atomic refinement, a strong notion of correctness easily embedded in particular into a compiler, for linearisable, fine-grained, concurrent data-structures. This refinement is not to be built by hand but automatically derived from proof obligations expressed in RG. This is made possible by a meta-theorem proved in Coq, and parameterised by any abstract data-structure, its implementation and its proof obligations.

The typical workflow for using our generic result is therefore to

(i) define the abstract data structures specification, *i.e.* their type, and the atomic semantics of methods in $\mathcal{I}$,

(ii) provide a concrete implementation, *i.e.* the representation in the heap of the instances of the data-structure, and a fine-grained hybrid implementation of methods,

(iii) define a coherence invariant between abstract and concrete data structures that formalises the link between a concrete data structure and its abstract view,

(iv) define the rely and guarantee of each method,

(v) prove the RG specification of each method using a dedicated program logic and

(vi) apply our meta-theorem to get the global correctness result.

We have successfully used this workflow to prove the correctness of the above illustrative spinlock example, as well as the concurrent buffers we presented in this chapter, solving the need initiated by the garbage collector formalisation.

6.3.6 *Notations*

In the following, we use the following notations and vocabulary. For a set A, an A predicate P is a subset of A. An element $a \in A$ satisfies the A predicate P, written $a \models P$, when $a \in P$. Given two sets A and B, a relation R is an $A \times B$ predicate. We use infix notations for relations. State predicates are $(H^\sharp \times H \times \mathit{Lenv} \times \mathit{LinState})$ predicates,

specifying shared memories and intra-thread states. A shared memory interference is a binary relation on $H^\sharp \times H$, and is used for relies and guarantees. We refer to both state predicates and shared memory interferences as *assertions*. It is important to note in the following that the rely-guarantee reasoning is done at the intermediate level $\mathcal{L}^\flat$, on instrumented programs, more precisely on the hybrid code of abstract methods implementations. Hence, assertions specify properties about the concrete and abstract heaps simultaneously.

6.3.7 *An enriched semantic judgement for RG triples*

We shall express RG proof obligations over hybrid methods entailing a semantic refinement. To do so, we need to take a moment to refine our semantic judgement of a RG triple. a hybrid method m, whose body is c, must indeed be specified with a semantic RG judgement of the form $R, G, I \models_m \{P\}\ c\ \{Q\}$, in the exact same fashion introduced in Chapter 3, and heavily used to prove the GC. A couple of twists are however needed. First, state predicate I takes here a particular meaning: it is in particular meant to specify the coherence invariant between abstract objects and their representation in the concrete heap. It is asked to be proved invariant separately (see Definition 17).

More importantly, the RG judgement is now made of three items. The already introduced intuition, typical of RG reasoning, is still soundly grounded in conditions (1) and (2) below.

Definition 14. *Judgement* $R, G, I \models_m \{P\}\ c\ \{Q\}$ *holds whenever:*

1. *The post-condition is established from pre-condition and invariant:*
   $(\langle m, c, l, ls\rangle, \sigma) \to_R^* (\langle m, \bullet, l', ls'\rangle, \sigma')$
   and $(l, ls, \sigma) \models P \cap I$
   implies $(l', ls', \sigma') \models Q$

2. *Instructions comply with the guarantee:*
   $(\langle m, c, l, ls\rangle, \sigma) \to_R^* (\langle m, c', l', ls'\rangle, \sigma') \to (\langle m, c'', l'', ls''\rangle, \sigma'')$
   and $(l, ls, \sigma) \models P \cap I$
   implies $\sigma'\ G\ \sigma''$

3. *Linearisation points are unique and non-blocking:*
   $(\langle m, c, l, ls\rangle, \sigma) \to_R^* (\langle m, \mathtt{Lin}(b), l', ls'\rangle, (h^\sharp, h))$
   and $(l, ls, \sigma) \models P \cap I$
   and $[\![b]\!] l' = \mathtt{true}$
   implies $\exists r, a, a', v, v'$, *such that* $h^\sharp(r) = a$
   and $[\![m]\!]^\sharp(a, v) = (a', v')$
   and $ls' = \mathsf{Before}(r, v)$

Condition (3) in Definition 14 is however novel, and more subtle. It captures a necessary requirement to ensure that Lin instructions do not block programs ($[\![m]\!]^\sharp$ is defined), and are unique (the linearisation state is Before). This condition is essential to ensure that we can clean up the Lin instrumentation of hybrid programs, and that our semantic refinement is not vacuously true. We come back to condition (3) in Section 6.3.10.

6.3.8 *Specifying hybrid methods: A RG Specification Entailing Semantic Refinement*

We now explain the precise RG specification we require for hybrid methods.

SINGLE OBJECT ASSERTIONS. The above RG judgement involves state predicates and shared memory interferences. In fact, we build them from elementary bricks, *object predicates* and *object interferences*, that consider one object — one instance of a data structure — at a time, pointed to by a given reference.

Definition 15 (Object predicate, object interference)**.** *Let* $r \in Ref$*. An* object predicate P_r *is a predicate on pairs of an abstract object and a concrete heap:* $P_r \subseteq A^\sharp \times H$*. An* object interference R_r *is a relation on pairs of an abstract object and a concrete heap:* $R_r \subseteq (A^\sharp \times H) \times (A^\sharp \times H)$*.*

Example 8 (Lock – Object invariant, object guarantees, and object relies)**.** *The coherence invariant specifies that an abstract* Locked *(resp.* Unlocked*) lock is implemented in the concrete heap as an object whose field* flag *is set to* 1 *(resp.* 0*). It is formalised as the following object predicate, parameterised by a reference* r*:*

$$\begin{aligned} \mathit{ILock}_r \triangleq\ & \{(\mathtt{Locked}, h) \mid h(r, \mathtt{flag}) = 1\} \\ & \cup \{(\mathtt{Unlocked}, h) \mid h(r, \mathtt{flag}) = 0\} \end{aligned}$$

Object guarantees for acquire *and* release *express the effect of the methods on the shared memory when called on a reference* r*. They are defined as the following object interferences.*

$$G_r^{rel} \triangleq \{((a, h_1), (\mathtt{Unlocked}, h_2)) \mid h_2 = h_1[r, \mathtt{flag} \leftarrow 0]\}$$

$$\begin{aligned} G_r^{acq} \triangleq \{((\mathtt{Unlocked}, h_1), (\mathtt{Locked}, h_2)) \mid\ & h_1(r, \mathtt{flag}) = 0 \\ & \wedge\ h_2 = h_1[r, \mathtt{flag} \leftarrow 1]\} \end{aligned}$$

In G_r^{acq}*, the assignment to* flag *is performed only if the* cas *succeeds.*

Finally, both acquire *and* release *have the same object rely, when called on a reference* r*: indeed, another thread could call both methods on the same reference. So we define the following object interference: for* $m \in \{rel, acq\}$*,* $R_r^m \triangleq G_r^{rel} \cup G_r^{acq}$*.*

Example 9 (Buffers – Object invariant, object guarantees, and object relies)**.** *The coherence invariant between an abstract buffer and its concrete implementation essentially consists in that every cell in the* data *array between* next_read *and* next_write *matches, in order, the elements of the list representing the abstract buffer – in particular, the number of valid cells in the array must be equal to the length of the abstract buffer.*

We express it as the following object predicate, where function range(s, e) *computes the list of successive indices between two integers* s *and* e*, modulo SIZE.*

$$
\begin{aligned}
&\mathit{IBuck}_r \subseteq (\texttt{list value}) \times H\\
&\mathit{IBuck}_r \triangleq \{(b, h) \mid \quad b = b_1 :: \ldots :: b_s \wedge s{<}\mathit{SIZE} \wedge \mathit{Separation}(h, \mathtt{data})\\
&\qquad\qquad \wedge h(r, \mathtt{next\_write}) = nw \wedge h(r, \mathtt{next\_read}) = nr\\
&\qquad\qquad \wedge nw{<}\mathit{SIZE} \wedge nr{<}\mathit{SIZE} \wedge \mathit{range}(nr, nw) = i_1 :: \ldots :: i_s\\
&\qquad\qquad \wedge \forall j \in \{1, ..s\},\ h(h(r, \mathtt{data}), i_j) = b_j\}
\end{aligned}
$$

The property $\mathit{Separation}(h, f)$ *states that all values of* f *fields in concrete heap* h *are unique. In the definition of* IBuck_r*, this encodes the separation of arrays (all* data *field are distinct references), thus reflecting the ownership of buffers by each thread.*

Methods guarantees reflect each basic shared memory effect of the methods. First, methods `top` *and* `isEmpty` *have no effect over the shared memory. Their object guarantee is therefore the identity, i. e.*

$$G_r^{top} = G_r^{isEmpty} = \{((a, h), (a, h))\}.$$

Methods `pop` *and* `push` *have been explained informally previously. Their guarantee* G_r^{pop} *and* G_r^{push} *reflect both their effect on the concrete heap, as well as the effect on the abstract buffer. The later is specified to occur atomically with the instruction annotated by the* `Lin` *instruction (see Figure 29). More precisely, the guarantee are defined as follows:*

$$
\begin{aligned}
G_r^{pop} \triangleq \{&((ab, h), (ab', h')) \mid\\
&[\![pop]\!]^\sharp(ab, \mathit{Null}) = (ab', \mathit{Null}) \wedge\\
&h(r, \mathtt{next\_read}) = nr \ \wedge\ h(r, \mathtt{next\_write}) = nw \ \wedge\ nr \neq nw\\
&\wedge\ h' = h[r, \mathtt{next\_read} \leftarrow (nr + 1) \bmod \mathit{SIZE}]\ \}
\end{aligned}
$$

The method `pop` *has only one observable effect, which occurs at the linearisation point. Its guarantee therefore relates abstract buffers which are related by* $[\![pop]\!]^\sharp$*. Assuming that the buffer is initially not empty, i. e. that the field* next_read *and* next_write *do not contain the same value, the concrete heap is see its* next_read *value incremented, modulo SIZE.*

The push *method on the other hand has two distinct observable effects, reflected in its guarantee.*

$$
\begin{aligned}
G_r^{push} \triangleq\ & \{((ab,h),(ab,h')) \mid \\
& h(r,data) = d \wedge h(r,next\_write) = nw \wedge \\
& h' = h[d,i_{nw} \leftarrow v]\} \\
\cup\ & \{((ab,h),(ab,h')) \mid \\
& [\![push]\!]^\sharp(ab,v) = (ab',\mathit{Null}) \wedge \\
& h(r,next\_read) = nr \wedge h(r,next\_write) = nw \wedge \\
& nr \neq (nw+1) \bmod \mathit{SIZE} \wedge \\
& h(r,data) = d \wedge h(d,i_{nw}) = v \wedge \\
& h' = h[r,next\_write \leftarrow (nw+1) \bmod \mathit{SIZE}]\ \}
\end{aligned}
$$

The first effect describes the storing of a pushed value v *into the concrete buffer. Since the value will only be visible once the value of* next_write *is updated, the operation is non-linearising, and hence the abstract buffer is left untouched. The second effect corresponds to the linearisation point. Once again, the abstract buffers are related by the abstract semantics* $[\![push]\!]^\sharp$*, while the value of* next_write *is updated, granted the buffer is not full.*

Finally, we express the usage protocol of single-writer, single-reader by defining the respective rely of methods. A thread should tolerate at least any interference that a method the thread calls describes in its rely. We therefore state that the rely of method push *is reduced to the effect of the method* pop*,* G_r^{pop}*. In particular, since it does not contain* G_r^{push}*, it encodes the fact that a thread calling the method* push *over a buffer will not be composed with another doing the same, it is the only pusher allowed over this buffer.*

$$R_r^{isEmpty} \triangleq G_r^{push} \quad R_r^{pop} \triangleq G_r^{push} \quad R_r^{top} \triangleq G_r^{push} \quad R_r^{push} \triangleq G_r^{pop}$$

LIFTING SINGLE OBJECT ASSERTIONS. Object predicates and object interferences allow to specifically describe predicates and relations over a sub-part of the concrete and abstract heap, describing one single instance of the abstract data-structure. They are very convenient to describe finely the effect a method called over an object r has, and expect from the environment. Indeed, it should not have to explicitely consider effects of methods over objects, as well as effects from the client code.

However, this means that in order to enunciate the RG specifications of hybrid implementations, state predicates and shared-memory interferences need to be *lifted*, cast into state predicates and shared-memory which describe the memory as a whole. The challenge here is twofold: make the specification effort relatively light for the user, and, more importantly, sufficiently control the specifications so that we can derive our generic result.

An object predicate P_r is lifted to a state predicate $\widehat{P}_r$ by further specifying that, in the abstract heap, r points to an abstract object satisfying P_r:

$$\widehat{P}_r = \{(h^\sharp, h, l, ls) \mid \exists a, h^\sharp(r) = a \wedge (a, h) \models P_r\}$$

Similarly, for an object guarantee G_r, reference r should point to an abstract object in the abstract heap. Moreover, its effect on this object should be reflected in the resulting abstract heap. Formally:

$$\widehat{G}_r = \{((h^\sharp, h_1), (h^\sharp[r \mapsto a_2], h_2)) \mid \exists a_1,\ h^\sharp(r) = a_1 \wedge (a_1, h_1)\ G_r\ (a_2, h_2)\}$$

Lifting relies is a bit more subtle. When executing a hybrid implementation m, one should account for two kinds of concurrent effects: the client code, and the rely of the method itself. To model the client code effect, we introduce a public shared memory interference, written R^{pub}, that models any possible effect on the concrete heap, except modifying private fields in $\mathcal{P}$:

$$R^{pub} = \{((h^\sharp, h_1), (h^\sharp, h_2)) \mid \forall r, f,\ f \in \mathcal{P} \Rightarrow h_1(r, f) = h_2(r, f)\}$$

As for the method's rely R_r, we should consider that it could occur on *any* abstract object present in the abstract heap. Hence, a lifted rely $\widetilde{R}$ includes (i) the client public interference, and (ii) the method's rely R_r quantified over all r:

$$\widetilde{R} = R^{pub} \cup \{((h_1^\sharp, h_1),(h_2^\sharp, h_2)) \mid \exists r, a_1, a_2, h_1^\sharp(r) = a_1 \wedge (a_1, h_1)\ R_r\ (a_2, h_2) \wedge h_2^\sharp = h_1^\sharp[r \mapsto a_2]\}$$

THE RG SPECIFICATION. Before we define the RG proof obligation asked of hybrid method implementations, let us first recall the definition of stability.

Definition 16 (Stability)**.** *State predicate* P *is* stable *w.r.t. shared memory interference* R *if* $\forall l, ls, \sigma_1, \sigma_2,\ (\sigma_1, l, ls) \models P$ *and* $(\sigma_1 R\ \sigma_2)$ *implies* $(\sigma_2, l, ls) \models P$*.*

Now, we fix an invariant I_r. For a method $m \in \mathcal{I}$, let G_r^m and R_r^m be the object guarantee and rely of m, as illustrated in Example 8 above. A RG specification for m includes an RG semantic judgement, and stability obligations:

Definition 17 (RG method specification)**.** *The RG specification for method* $m \in \mathcal{I}$ *includes the three following conditions:*

- *For all* $r \in Ref$*,* $\widetilde{R}^m, \widehat{G}_r^m, \widehat{I}_r \models_m \{\mathbb{P}_r\}\, m.comm\, m\{\mathbb{Q}_r\}$
- *For all* $r \in Ref$*, predicate* $\widehat{I}_r$ *is stable w.r.t.* $\widetilde{R}^m$

- *For all* $r, r' \in \mathit{Ref}$*, predicate* $\widehat{I}_r$ *is stable w.r.t.* $\widehat{\overline{G}^m_{r'}}$

In the above judgement, we impose the pre- and post-condition $\mathbb{P}_r$ and $\mathbb{Q}_r$. $\mathbb{P}_r$ expresses that (i) r points to an abstract object in the abstract heap, (ii) the linearisation state is set to Before and (iii) the reserved local variable this of method m is set to r. $\mathbb{Q}_r$ expresses that (i) the linearisation state is set to After, and (ii) the value virtually returned by the abstract method (when encountering the Lin instruction) matches the value returned by the concrete code. Finally, the stability requirements in the specification intuitively ensure that $\widehat{I}_r$ is indeed an invariant of the whole program.

6.3.9 *Main theorem*

So far, we have expressed requirements on hybrid methods, each taken in isolation. The last requirement we formulate is the consistency between relies and guarantees of methods. For a method m, to ensure that R^m is indeed a correct over-approximation of its environment, we ask that R^m includes any guarantee $G^{m'}$, where m' is a method that may be called concurrently to m. This requirement is formalised by the following definition.

Definition 18 (RG consistency)**.** *For all threads* t, t' *such that* $t \neq t'$*, all methods* $m, m' \in \mathcal{I}$ *and all* $r \in \mathit{Ref}$*,* $\mathsf{is\_called}(t, m) \wedge \mathsf{is\_called}(t', m') \Rightarrow G^{m'}_r \subseteq R^m_r$ *where* $\mathsf{is\_called}(t, m)$ *indicates that* m *syntactically appears in the code of* t*.*

Relying on predicate is_called allows for accounting for data structures used according to an elementary protocol (such as single-writer, single-reader buffers).

We finally package the formal requirements on hybrid implementations into the RGspec specification and use it to state our main result, establishing that the target program semantically refines the source program.

Definition 19 (RGspec specification)**.** *Let* $\mathcal{I} = \{m_1 \ldots, m_n\}$*.* $\mathcal{I}$ *satisfies* RGspec*, written* $\mathsf{RGspec}(\mathcal{I})$*, if* $\forall\ i \in [1, n]$*, a RG method specification is provided for* m_i*, and RG consistency holds.*

Theorem 2 (Compiler correctness)**.** *Let* σ_i *an initial shared memory satisfying the invariant* I_r *for all* $r \in \mathit{Ref}$ *allocated in it. If* $\mathsf{RGspec}(\mathcal{I})$*, then* $\forall p \in \mathcal{L}^\sharp,\ \mathit{wf}(p) \rightarrow \mathsf{beh}(\mathtt{compile}\langle\mathcal{I}\rangle(p), \sigma_i) \subseteq \mathsf{beh}(p, \sigma_i)$.

We emphasise the fact that the client program p is arbitrary, modulo some basic syntactical well-formedness conditions defined in the predicate *wf*. This predicate checks that the client code only accesses public fields, and does not contain linearisability annotations. The implementations of methods in $\mathcal{I}$ on the other hand are assured to only

access private fields, and are not allowed to neither contain abstract calls nor printing instructions.

Theorem 2 is phrased and proved *w.r.t.* an RG semantic judgement. In our formal development, we have equipped the language $\mathcal{L}^{\flat}$ with a sound, syntax-directed proof system similar to the one introduced for RTIR and described in Chapter 3. This system is used to discharge the RG semantic judgement, and have successfully been applied to prove the implementation of the spinlock, as well as the buffer data structure.

6.3.10 *Establishing the Generic Simulation from* RGspec

Theorem 2 is proved by establishing, from the RGspec($\mathcal{I}$) hypothesis, a simulation between the source program and its transformation by `compile`. As exposed in Section 6.1, we do so through the use of a *backward* simulation. Having two successive transformations, we actually establish accordingly two backward simulations, from $\mathcal{L}$ to $\mathcal{L}^{\flat}$ and from $\mathcal{L}^{\flat}$ to $\mathcal{L}^{\sharp}$, which we compose.

6.3.10.1 *Leveraging* RGspec($\mathcal{I}$)

A key point is to carry, within the simulation, enough information to leverage RGspec($\mathcal{I}$). Indeed, the RGspec($\mathcal{I}$) hypothesis contains in particular RG assertions. Such judgements gives us information about states resulting from a partial, or complete, execution of a method in $\mathcal{I}$, starting from suitable initial states, with respect to an abstractly interleaved semantics. However during the proof of a simulation, we typically consider a given state, and inquiry the resulting state after a step of the concrete semantics. We therefore first have to make a link between these two contexts in order to be able to use our hypothesis.

Since this is necessary for both simulations, we factorise the work by expressing a rich semantic invariant $\mathbb{I}$ over the execution of the $\mathcal{L}^{\flat}$ program. To simplify its definition and its proof, $\mathbb{I}$ is built as a combination of thread-local invariants. Indeed, as rely-guarantee reasoning suggests, one always wants to avoid reasoning about all threads simultaneously. We therefore retrieve a notion of thread modularity by proving a family of invariants, one for each thread, and combine them afterwards to build a global invariant of the program.

THREAD LOCAL INVARIANT. For a thread t, the invariant $\mathbb{I}_t$ includes three kinds of information. Its formal definition is laid down on Figure 35, we now go over it. As can be expected, few things are enforced when we are not inside a method in $\mathcal{I}$, the left handside of the figure, while the right handside describes the situation when inside a method.

First, the invariant ensures various well-formedness properties of intra-thread states. Those very boring predicates, wf_{out} and wf_{in} are

$$\frac{\begin{array}{c} ls = \mathsf{Nolin} \\ wf_{out}(i) \\ m \notin \mathcal{I} \\ \forall r,\ (\sigma, ls, l) \models \widehat{I}_r \end{array}}{(\langle m, i, l, ls\rangle, \sigma) \in \mathbb{I}_t^R} \qquad \frac{\begin{array}{c} ls = \mathsf{Before}(this, arg) \vee ls = \mathsf{After}(this, arg, res) \\ wf_{in}(i) \wedge m \in \mathcal{I} \wedge \exists ab,\ \sigma.h^\sharp(this) = ab \\ \forall r,\ (\sigma, ls, l) \models \widehat{I}_r \\ (m, m.instr, l_0, \sigma_0) \rightarrow_{R_t} (\langle m, i, l, ls\rangle, \sigma)) \\ \sigma_0 \models \mathbb{P}_r \cap \widehat{I}_{this} \end{array}}{(\langle m, i, l, ls\rangle, \sigma) \in \mathbb{I}_t^R}$$

Figure 35: Major invariant for $\mathcal{L}^\flat$.

simply there to carry over the execution the well-formed assumptions wf our main theorem take for hypothesis.

Second, $\widehat{I}_r$, the coherence invariant, holds for all objects r.

The third information is more subtle. When executing a hybrid method m called on a reference r, in order to leverage its specification $\widetilde{R}^m, \widehat{G}_r^m, \widehat{I}_r \models_m \{\mathbb{P}_r\}\, m.comm\, \{Q_r\}$, we keep track, in $\mathbb{I}_t$, that the state is reachable from a state satisfying $\widehat{I}_r$ and $\mathbb{P}_r$. Indeed, as previously stated, recall that Definition 14 uses the abstract semantics $\rightarrow_{\widetilde{R}^m}$. In order to retrieve from this RG statement any information about the state at the end of the execution of the method, we therefore need to exhibit a trace of this execution, with respect to $\rightarrow_{\widetilde{R}^m}$.

To do so, we first need to find a suitable rely. Indeed, the thread may call several different methods in $\mathcal{I}$ during its execution, and they might all have different $\widetilde{R}^m$. We hence generalise all relies $\rightarrow_{\widetilde{R}^m}$ of methods that the thread may call, i. e. whose call appear syntactically in its initial code, into a relation $\rightarrow_{R_t}$. This relation is defined as the intersection of these relies: $R_t \triangleq \bigcap_{m \in \mathsf{is\_called}(t)} \widetilde{R}^m$. Intuitively, each method m puts an hard constraint on the table: it shall not tolerate any more interferences than described in $\widetilde{R}^m$. If a thread wants to call several methods, hence putting them into a common environment, it shall accommodate everyone: the best we can do is therefore the intersection. Rely R_t hence over-approximates interferences of threads concurrent to t, while being precise enough to deal with any method m called by t, since $R_t \subseteq \widetilde{R}^m$.

Now equipped with our new rely, we can keep track of the ongoing execution of a method. When making the call, we record the state σ_0 from which the execution has started. This state satisfied the preconditions $\mathbb{P}$ and $\widehat{I}_{this}$ needed. Finally, we build the abstract execution of the method, starting from σ_0 and an initially empty local environment l_0 with respect to $\rightarrow_{R_t}$.

GLOBAL INVARIANT. We define the global invariant as $\mathbb{I} \triangleq \{(\gamma, \sigma) \mid \forall t, (\gamma(t), \sigma) \models \mathbb{I}_t\}$ expressing that all thread-local invariants simultaneously hold. We now want to prove that $\mathbb{I}$ is stable under the interleaving semantics.

To do so, we perform two steps. First, we prove that all thread local invariants deserve their name: $\mathbb{I}_t$ is preserved by the intra-thread semantics. Each thread's invariant being stable under the thread's own steps is naturally insufficient in general for $\mathbb{I}$ to be invariant: a step performed by a thread t' could break $\mathbb{I}_t$. We therefore additionally prove their preservation by other threads' steps:

Lemma 3. *Let* γ, σ *and* $t \neq t'$ *be such that* $(\gamma(t), \sigma) \models \mathbb{I}_t$ *and* $(\gamma(t'), \sigma) \models \mathbb{I}_{t'}$. *If* $(\gamma(t'), \sigma) \xrightarrow{o} (ts', \sigma')$, *then* $(\gamma(t), \sigma') \models \mathbb{I}_t$.

6.3.10.2 *Simulation Relations*

Mimicking the approach we took to build the invariant, we try to keep as much work as possible local within a thread. Therefore, for both compilation phases, we build an intra-thread, or *local*, simulation that we then lift at the inter-thread level. Both relations are defined using the same pattern: in a pair of related states, the $\mathcal{L}^\flat$ state satisfies $\mathbb{I}_t$. It remains to encode in the relation the matching between execution states.

LOCAL HIGH SIMULATION. For the first compilation phase, a whole execution of a hybrid method is simulated by a single abstract step, occurring at the linearisation point. We therefore build a *1-to-0/1* backward simulation.

Relation $\mathcal{S}^\sharp_\flat$ states that shared memories are equal on the heaps domains in $\mathcal{L}^\sharp$. Local environments are trickier to relate. In client code, they simply are equal. During a hybrid method call $x = y.m(z)$, before the Lin point, the abstract environment is equal to the environment of the $\mathcal{L}^\flat$ caller. After the Lin point, the only mismatch is on variable x, which has been updated in $\mathcal{L}^\sharp$, but not yet in $\mathcal{L}^\flat$.

Proving that $\mathcal{S}^\sharp_\flat$ is a simulation follows the above three phases. Steps by client code are matched *1-to-1*; inside a hybrid method, steps match *1-to-0* until the Lin point; the Lin step is matched *1-to-1*; after the Lin point, steps match *1-to-0* until the return instruction. At method call return, we use $\mathsf{RGspec}(\mathcal{I})$ via $\mathbb{I}_t$, and in particular $\mathbb{Q}_r$, to prove that environments coincide again on x.

LOCAL LOW SIMULATION. When simulating from $\mathcal{L}$ to $\mathcal{L}^\flat$, Lin instructions have been replaced by a $\bullet$. It is therefore a *1-to-1* backward simulation. Recall that $\mathcal{L}$ semantics contains no *LinState* nor abstract heap. Relation $\mathcal{S}^\flat$ therefore only states the equality of local environments and concrete heaps.

Proving this simulation is what makes the third item in Definition 14 necessary. Indeed, we can match the $\bullet$ step in the $\mathcal{L}$ program only if the Lin instruction in the $\mathcal{L}^\flat$ program is non-blocking.

GLOBAL SIMULATIONS. Once again, establishing a family of thread-local simulations define a natural global simulation relation by assert-

$$
\begin{array}{ccccccc}
 & & & & t \neq t' & & \\
(st_2\ t, \sigma_2) & \to^{\{0,1\}} & (ts_2, \sigma_2') & & (st_2\ t', \sigma_2) & & (st_2\ t', \sigma_2') \\
\wr_t & & \wr_t & \wedge & \wr_{t'} & \Rightarrow & \wr_{t'} \\
(st_1\ t, \sigma_1) & \to^{\{0,1\}} & (ts_1, \sigma_1') & & (st_1\ t', \sigma_1) & & (st_1\ t', \sigma_1')
\end{array}
$$

Figure 36: Sufficient condition to lift a family of thread-local simulations. The relation of a thread t' should be stable under a matching diagram performed by another thread t.

ing that all thread-local relations simultaneously hold. A sufficient condition for this relation to define a global simulation is described in Figure 36. Given two distinct threads t and t', consider two global states, source and target, whose respective projections are related by their thread-local relations. The stability condition states that the effect on the shared memory a thread-local matching diagram performed by t has should not break the relation of thread t'.

In our case, having proved independently the invariant simplifies this stability condition. Indeed, except for the part about $\mathbb{I}_t$, that is already proved invariant by other threads' steps, relations keep track of the same information for all threads. Hence, their preservation by the interleaving essentially comes for free.

6.4 RELATED WORK

The literature on linearisability verification is vast. Dongol *et al.* [36] provide a comprehensive survey of techniques for verifying linearisability *w.r.t.* the seminal definition of Herlihy [56].

The closest approach to ours is probably the work by Derrick et al. [28]. Their work, formalised in the proof assistant KIV, follows a similar structure to ours. They also derive systematically a simulation from higher level proof obligations over the implementation. The core difference between our contributions is in the nature of the proof obligations. They reason thread locally and decouple a notion of sequential refinement from a notion of stability, but their proof obligations are semantic. In contrast, we express them directly with respect to our proof system.

Notably, a number of works use combinations of rely-guarantee and concurrent separation logics. One of the first approaches of this nature is the PhD of Vafeiadis from which our own work is inspired. Using RGSep [138], he introduced the idea of reasoning over hybrid implementations annotated with linearisation points and proved linearisable several algorithms [137, Chapter 5]. This aspect of his work was however not formalised, and most importantly had no rigorous

semantic ground. Vafeiadis also developed a non-verified tool written in OCaml based on this approach to automatically prove the linearisability of concurrent data-structures [139].

Our approach is sufficient to tackle the refinement of the mark buffers used in Domani et al.'s algorithm [35]. We would also be able to handle Treiber's stack [135] in order to implement the freelist. However, some more complex data-structures are currently out of our reach. Indeed, through the use of the `Lin(b)` instruction, we can capture linearisation points occurring inside the method's body when a condition is satisfied. However, two kinds of more involved situations can occur.

First, linearisation can be *external*, i. e. the linearisation of a method executed by a thread may occur during a computation step performed by another thread. An example of this kind is the HSY elimination-based stack [55]. This algorithm implements a fine-grained concurrent stack, much like our own implementation of buffers. However, the key idea behind the algorithm is to let a `push` and `pop` occurring simultaneously cancel each other. If both methods occur concurrently, we can indeed always allow the `push` to resolve before the `pop`. But if we do so, then the data-structure, being pushed and then popped, will end up in the same state as it was initially. We therefore might as well simply leave the data-structure untouched, and directly return in the `pop` method the pushed value. To do so, when a thread starts a method, it writes its identity and the description of the operation in a shared array. When a thread tries to push a value, it first checks if there is a pending `pop`. If so, it uses the shared array to directly communicate the desired value, linearising in the process the popping thread.

Second, linearisation can be *future dependent*. The linearisation of an operation happens at a fixed program point inside the method. In our instruction `Lin(b)`, whether or not an execution actually linearises or not the method is conditioned by `b`. At *future dependent* linearisation points, whether the instruction indeed linearises the method depends on the validity of a condition *at another program point*, further in the code. Intuitively, the second program point decides whether the effect of the operation has indeed been propagated, but the structure is already in a different state such that the effect cannot be interpreted as occurring at this point. An example a future dependent linearisable algorithm is Harris et al. [51] multi-word compare-and-swap operation.

Liang and Feng [83], building upon their work on RGSim [84], have developed a proof system able to handle both cases. To tackle external linearisation, their approach extends the state with a pending thread pool. This pool is essentially similar to our `Before/After` annotations, but describes all threads simultaneously. Their `Lin` annotation is parameterised by a thread identifier, allowing one thread's concrete step

to trigger the abstract semantics of another thread's method registered in the thread pool. To handle helping, they introduce two new annotations. A `TryLin` instruction signals that the method might linearise at this point, but that it shall be confirmed later. The semantics is therefore simply to duplicate the state, and simulate both possibilities in parallel. Then the `Commit(b)` instruction only keeps the scenario which turns out to be correct based on the value of `b`.

Numerous program logics [30, 40, 41, 68, 128, 138] have sought to develop general principles to reason about concurrent programs, beyond the particular case of linearisability. While they introduce countless principles that can be leveraged, they usually cannot be used *off the shelf* to address our problem at hand. Indeed, they typically permit to prove that a concurrent data-structure satisfies specifications of various complexity. However they *cannot* prove a refinement of a data-structure, such as the one we have established. An interesting exception is to be noted. Krebbers et al. [72, Section 6] use their general purpose logic Iris to embed logical relations which do entail a refinement.

6.5 CONCLUSION

The mechanised verification of the garbage collector we described in Chapter 5 has left a significant gap towards an executable certified runtime: RTIR embarks atomic concurrent data-structures. In this chapter, we filled this gap by designing and mechanising an approach to reduce the observational atomicity refinement of a linearisable data-structure to rely-guarantee proof obligations. We illustrated the viability of the process by instantiating it to implement the mark buffers used in RTIR.

7

CONCLUSION

7.1 SUMMARY

With software percolating in every aspects of our life, formal methods have become mandatory in an attempt to retain trust in safety-critical software. A fundamental link in the chain of trust of a system is the compiler. If it were to modify the admissible behaviours of the compiled program, any reasoning performed at the source level could be invalidated. However, modern optimising compilers are so complex that it is extremely hard to trust them, and for good reason. In 2011, Yang et al. [147] thoroughly tested state-of-the-art industrial compilers and reported the following.

> "We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 325 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs."

Verified compilation, whose most influential proponent is probably the C compiler CompCert [22, 82], aims at addressing the issue. A formal semantics of both the source and the target languages is introduced, and the compiler is proved to introduce no new behaviours during the process of compilation. There is however still a significant gap to fill before being able to achieve such a result for a managed, concurrent, high-level language such as Java. In particular, handling the sophisticated concurrent reasoning to verify the runtime is a major challenge.

In this thesis, we have contributed to shorten this gap. We have verified in the Coq proof assistant a state-of-the-art On-The-Fly garbage collector inspired by the work of Domani et al. [35]. Achieving such a result is a combined challenge from the theoretical, methodological and proof engineering standpoints. To this end, we put all along our work a peculiar emphasis on three key aspects.

- Proofs must be conducted with respect to the operational semantics of the program. Verified compilation crucially requires to be able to prove the *executable code* itself and not the underlying algorithm. To this end, we make sure to only introduce abstractions that we realistically know how to refine.
- Proofs must be conducted in a methodological way. Formal proofs of programs are always complex, formal proofs of concurrent programs are significantly harder. We argued that separation of concerns must be sought at all cost.
- Correctness results must be expressed in terms of observational refinement, the standard in verified compilation.

In accord with these principles, we have grounded our formal development in the design of a dedicated intermediate representation, RTIR. This language strives to embed the right abstractions instrumental to ease the formalisation and proof of concurrent runtime implementations, while being amenable to be refined into a fully executable language. These abstractions aim at two essential goals. First, they provide facilities for introspection, most notably through native support for roots and freelist, as well as dedicated iterators. Second, they provide abstract specifications of concurrent data-structures, and more specifically of the mark buffers which are used by the garbage collector to implement the pending queue of cells to be explored.

After the design of RTIR, the second methodological step has been the choice of a reasoning framework. We decided to strike a balance between expressiveness and ease of mechanisation by having our heart set on the well established rely-guarantee methodology. The design of our specific proof system is novel, putting significant emphasis on mechanisation and separation of concerns. First, we combine syntax guided rules over annotated code with base rules expressed in terms of operational interpretation of the logic to ease automation of the sequential proofs of programs. Second, we unravel the interlacing between sequential reasoning and stability obligations to defer them in two distinct places. The resulting logic is proved sound with respect to the operational semantics of RTIR, fitting our goal towards the proof of executable code.

Even so, proofs of programs as complex as the garbage collector we consider remain a major challenge to get through. To ease further the process, we identified a need for an incremental workflow. This extremely natural idea is easily compromised in a concurrent setup, due to the interdependence between the predicates we proved over a thread and the abstraction we make of the environment. Naively refining the invariants requires to refine the guarantees, which invalidates the previous proof of validity of the invariant. To tackle this issue, we designed an incremental methodology based on metatheorems of our logic allowing to refine in a monotonic way our development, without breaking previously established proof scripts. We successfully followed this strategy, allowing us to split the development in seven layers. Most crucially, we managed to reason about the synchronisation protocol as a preliminary step, independently from reachablity and colouring considerations.

Coming back to our core design principles, we promised to only introduce abstractions that we deem possible to formally refine. However, this task is far from trivial when it comes to fine-grained concurrent data-structures such as the mark buffers supported abstractly by RTIR. To address this concern, we devised a light weight methodology, based on our rely-guarantee proof system, to refine linearisable data-structures. The approach is inspired by the seminal work of

Vafeiadis [137]. In order to fit our context, we however provide strong semantic, operational, foundations to the technique. For any abstract data-structure $\mathcal{D}$, we derive from rely-guarantee proof obligations an observational refinement from a source language equipped with an abstract version of $\mathcal{D}$, such as RTIR, to one implementing concretely $\mathcal{D}$. We proceed so by deriving complex backward simulations from the semantic proof obligations. As opposed to alternate characterisations of linearisability, our formulation of the problem allows for a straightforward embedding in a verified compilation context. Finally, we successfully applied our method to implement the mark buffers, the central concurrent data-structure manipulated abstractly by RTIR, justifying the original choice of design.

The contributions of this thesis have led to two formal Coq developments. First, i) the design of RTIR, ii) the design of our proof system, iii) the conception of an incremental methodology, iv) the implementation of the garbage collector in RTIR and v) its verification are wrapped up in a first development available online[1].

Second, i) the formalisation of our rely-guarantee-based methodology for atomic refinement of linearisable data-structures, ii) its semantic proof of soundness with respect to observational refinement and iii) the instantiation of the technique to prove the refinement of marked buffers is handled in a second Coq development, also available online[2].

7.2 PERSPECTIVES

7.2.1 *A verified garbage collector on an executable* RTIR

The first middle-term perspective would be to follow to the end the lead we started by transporting the correctness result of the garbage collector towards a more operational version of RTIR.

The major conceptual work has been done through the formalisation of a verified methodology for atomic refinements of linearisable data-structures, as described in Chapter 6. A first step would therefore be to plug both developments together. While conceptually straightforward, this might represent quite a refactoring effort.

Beyond this administrative aspect, two essential improvements are needed. First, we demonstrated how to refine the challenging feature of RTIR, the mark buffers. Yet, other aspects of the language need to undergo a similar process. The management of roots should be implemented as a data structure. However, although for convenience we tracked them in our development in the shared state, the data-structure is purely thread-local. This refinement would therefore rise very little challenges. The implementation of the `freelist` is signifi-

1 http://www.irisa.fr/celtique/ext/cgc/
2 http://www.irisa.fr/celtique/ext/simulin/

cantly harder, and would likely constitute a contribution of its own. At its core, the data-structure is a shared list, typically implemented as a Treiber's stack [135] or a Michael-Scott queue [94]. On top of it, fragmentation of the memory need to be taken into account. To simplify things, it should be possible to handle the management of the fragmentation at a thread local level in order to restrict the use of the shared data-structure to its simplest. Refining the freelist would therefore be an interesting work involving both a careful design of the algorithm, and some non-trivial fine-grained reasoning.

The second improvement has to do with blocking behaviours. Indeed, the semantics of RTIR has several blocking behaviours, such as trying to dereference a non-allocated cell. Naturally, we took special care to code our garbage collector in such a way that we do not end up in those cases. A necessary step would nonetheless be to equip the language with a simple type system entailing that well-typed programs never block.

7.2.2 *A modular proof of a concurrent generational garbage collector*

From the perspective of concurrency, the algorithm we verified is a state-of-the-art On-The-Fly garbage collector. Yet, we dismissed orthogonal optimisations and refinements that a modern compiler must handle.

Some of those refinements relate to the precise memory model considered. By adapting DLG's algorithm [32, 33] to Java, Domani et al. [35] covered features such as *finalisation* and *weak references*. These extensions would eventually have to be tackled, but strongly depend on the language whose compilation is targeted.

A higher level consideration is the question of combining optimisations. How should a proof of a concurrent, generational garbage collector be conducted? Ericsson et al. [125] have recently verified a sequential, generational garbage collector as part of the CakeML compiler. To tackle a garbage collector combining both On-The-Fly concurrency and generations, we could naturally start the proof from scratch, and conceive a new monolithic one. However, the arguments related to the correctness of both aspects should intuitively be independent. Being able to design the proof in such a modular way that we could essentially plug the arguments we developed with the ones from Ericsson could result in an elegant contribution.

7.2.3 *A verified synchronised monitor*

Through this work, we focused our attention towards the emblematic challenge of verifying an On-The-Fly garbage collector. However, we designed RTIR as a more general purpose intermediate representa-

tion for the implementation and verification of concurrent runtime implementations.

A fruitful follow work would therefore be to audit the legitimacy of this claim of generality by proving other concurrent runtime implementations in the same framework. In particular, an interesting use case would be the verification of synchronised monitors, such as the ones provided by Java [3]. Monitors, invented by Hansen and Hoare [50, 59] in the seventies, are a service destined to ease concurrent programming. A monitor provides thread safe accesses to shared data-structures by taking care of ensuring mutual exclusion. Their rigorous use is therefore a convenient way for the programmer to manipulate a shared data-structure while ensuring that no data-race occurs. Naturally, this comes at the cost of a significant loss of performance when compared to the kind of fine-grained concurrency manipulated by the garbage collector or the implementation of mark buffers we tackled. Yet, the trade-off for simplicity is worthy for many applications.

The verification of such monitors could lead to two interesting lines of investigation. First, they offer yet another verification challenge in concurrent programming to test the expressiveness of RTIR and our proof system. In particular, it might be the case that RTIR needs to be extended to manipulate thread identifiers at the value level. Second, monitors make for a great use case to investigate the verification of progress properties. Indeed, a major risk of mutual exclusion is to lead to *dead lock* situations: all threads are waiting for one another, preventing further progress.

Reasoning principles for progress properties have been developed in recent years, most notably by Liang et al. [86, 87]. Yet, the topic has received to date significantly little interest compared to the one of correctness.

7.2.4 *Weak memory models*

Our work assumes a *sequentially consistent* (SC) memory model. The set of admissible executions of a program is the set of possible interleavings of the atomic instructions of the program's threads. In particular, the operations of a given thread may not be executed consecutively, but their order will always remain the same.

Unfortunately, in order to enable aggressive optimisations, neither processors nor higher level concurrent programming languages satisfy this hypothesis. They allow for *more* executions than the SC ones, supporting so-called *weak memory models*. Depending on the architecture or the language, various weak memory models exist, and defining faithful formal models for all architecture is an active topic of research. A first major trend to address this problematic is embodied in the work of Alglave et al. [4, 6] which defines an axiomatisation of

a wide variety of models in a unified framework. For each model, a set of dependency relations between instructions is defined. The set of admissible instructions is then the set of instructions which do not entail a cycle among these relations.

These axiomatic models are a precious tool to conduct formal reasoning upon these models, notably allowing for model checking [5]. In some contexts, and notably in verified compilation, operational characterisations are however arguably necessary, both for informal and formal reasoning. Such operational models have been introduced for all mainstream architectures during the last decade. Notable such works include an abstract machine for x86 [108], an abstract machine for POWER [126], and a more involved operational model for ARMv8 [44]. In addition to memory models for processor architectures, the C/C++11 memory model has been the subject of much attention on recent years. Providing it with an operation model is especially challenging in that it is defined inherently in an axiomatic way, aiming to be sound with respect to various compiler and architecture optimisations. Nonetheless, Nienhuis et al. [103] managed to provide a model, sound with respect to this axiomatic specification, and operational in the sense that its allowed behaviours can be computed incrementally by starting from an initial state, as opposed to being global properties as may be the case with axiomatic models.

These major works therefore lay the ground required for sound, formal, reasoning with respect to these weak memory models. However, most of those models are complex entities to apprehend and manipulate, and each bring their own subtle reasoning principles. Being able to conduct complex formal reasoning on programs, such as the ones involved in this thesis, with respect to each of these memory models is therefore a significantly ambitious perspective.

A first significant step would however be to first tackle the arguably simpler, relative to the others, *Total Store Order* (TSO) weak memory model. This model is of significant practical interest since it is the one exhibited by x86 processors. Yet, it both remains quite intuitive to reason about, and admits an appealing operational characterisation. The following minimal two-threaded program is commonly used to illustrate a non-SC behaviour admissible under TSO. Upper case (resp. lower case) variables are global (resp. local), and we assume an initial shared state in which $X = Y = 0$.

```
// Thread t          // Thread t'
  X = 1                Y = 1
  x = Y                y = X
```

Under SC assumptions, only three possible resulting pairs of final values for (x, y) are admissible: $\{(1,1);(1,0);(0,1)\}$. However one can actually experimentally observe $(0,0)$. Indeed, in TSO, writes can be reordered *after* reads. Said differently, it is possible to read an out-

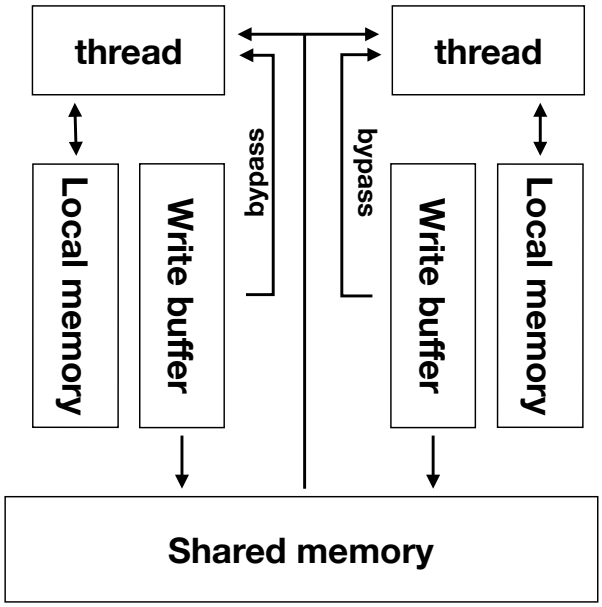


Figure 37: Simplified TSO abstract machine.

dated value. The corresponding abstract machine, whom a simplified version is depicted on Figure 37, clarifies the situation. As usual, each thread is equipped with a local memory, and they share a global memory. However, they now additionally own a *write buffer*[3].

These buffers model the processor's caches, while the shared memory models the RAM. Since accesses to the RAM are much more costly, we try to lower their amount. When a thread tries to write a value into the global memory, this value is first written to its buffer. Reciprocally when it tries to read a variable, it first checks its own buffer. If the buffer contains a write to the variable, then the thread directly reads this value; only otherwise does the thread read the value stored in shared memory. Periodically, or through dedicated instructions, the buffers are flushed into the memory.

Since buffers are not shared, we now understand how $(0, 0)$ can be observed in the previous example. Both threads have immediately written 1 to respectively X and Y, but those writes are pended in their buffers. When they then read the global variables, they cannot see each other writes, since those have not been flushed yet, and therefore both read 0.

Reasoning formally about TSO remains extremely hard to date, but constitutes a necessary stepping stone towards a verified compiler for a concurrent language. On the compilation side, the only attempt to the best of our knowledge is the work by Sevcik et al. [130] around CompCertTSO. While the project left open a vast amount of problems to investigate, through notably more complex optimisations

3 In practice, buffers are associated to processors. We assume here for simplicity a single thread per processor.

than fence-elimination, it lays a precious foundation for future work. The work by Jagannathan et al. [63] could also be instrumental by proposing an atomicity refinement methodology sound under TSO memory model. The approach is however not powerful enough to handle linearisable implementations such as the ones we tackled is Chapter 6.

On the verification of runtime implementation side, the impressive work by Gammie et al. [45] is the staple. Their approach nonetheless leaves for improvements. First, Domani et al.'s algorithm [35] has been designed and is strongly believed to remain correct under a TSO memory model. On the contrary, Gammie et al. introduced additional synchronisation points to reduce the amount of races involved, source of complex reasoning in TSO. Second, and once again, they abstracted the code of the algorithm and proved the resulting transition system. Through this abstraction and a significant effort, they managed to conduct the proof with a straightforward Owicki-Gries approach. We would rather be interested in using more local reasoning, at least to the level we achieved through the use of rely-guarantee reasoning.

Developing program logic for TSO memory model is however far from trivial. Ridge [122] proposed in 2010 a straightforward extension of rely-guarantee logic for TSO memory models. The approach is however quite crude: predicates explicitly characterise the state of the buffers, and extend all relies with flushes. Proofs in this system therefore appear to be extremely verbose to conduct, and in particular fail to retain conciseness in the case of sufficiently synchronised code. Sieczkowski et al. [131] introduced a more principled approach in 2015. Their logic, iCAP-TSO, is based on separation logic and structured in two levels. At the low level, the logic allows for explicit reasoning about the buffers. However, their logic also provides an *fiction of sequential consistency*, allowing to switch to SC reasoning when the data structure manipulated provides enough synchronisation.

In complement to program logics, alternate approaches have also been developed in order to reduce reasoning in presence of a relaxed memory model to SC reasoning. A commonly used class of well-behaved programs are the so-called data-race free (DRF) programs, i. e. programs such that no two threads can simultaneously attempt to access to the same memory address, one of them trying to write. It is well-known that under a TSO memory model, any execution of a DRF program admits an equivalent SC execution. Hence, if one can prove that a program is DRF, one can fall back on SC reasoning to prove it. Consequently, a rich line of work has flourished to enforce data-race freedom, notably statically through type systems such as in Mezzo [11] or Rust [69], as well as dynamically [89]. DRF principle is however too restrictive of a principle for high performance needs. Owens [107] introduced in 2010 a weaker trace-based property, so-

called triangular-race free, which exactly captures in a TSO context the programs for which a SC reasoning can be soundly performed. This approach, while appealing, is once again restricted to a subclass of programs, and suffers from a lack of compositionality.

Finally, a significant line of logics for the C/C++11 memory model have been developed. Lahav and Vafeiadis [76] proved that the classical Owicki-Gries proof system is unsound with respect to the C11 memory model, and strengthened the approach to handle the Release-Acquire fragment of this model. In parallel, significant effort has been invested to bring the most recent separation logics progress to C11 memory model [31, 140], culminating with a framework to reason about Release-Acquire consistency embedded in the Iris framework [70].

BIBLIOGRAPHY

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. "Rodin: An Open Toolset for Modelling and Reasoning in Event-B." In: *International Journal on Software Tools for Technology Transfer* (Nov. 2010).

[2] Agda Development Team. *The Agda theorem prover, version 2.5.3*. Available at http://wiki.portal.chalmers.se/agda/pmwiki.php. 2017.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. *An Efficient Meta-lock for Implementing Ubiquitous Synchronization*. Tech. rep. 1999.

[4] Jade Alglave. "A Formal Hierarchy of Weak Memory Models." In: *Formal Methods in System Design* 41.2 (2012), pp. 178–210.

[5] Jade Alglave, Daniel Kroening, and Michael Tautschnig. "Partial Orders for Efficient Bounded Model Checking of Concurrent Software." In: *International Conference on Computer Aided Verification*. Proc. of CAV'13. 2013.

[6] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory." In: *TOPLAS'14: ACM Transactions on Programming Languages and Systems* (July 2014).

[7] Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joseph Tuong. "Complx: A Verification Framework for Concurrent Imperative Programs." In: *Conference on Certified Programs and Proofs*. Proc. of CPP'17. 2017.

[8] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. "CertiCoq: A verified compiler for Coq." In: *CoqPL'17: International Workshop on Coq for Programming Languages*. 2017.

[9] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[10] E. A. Ashcroft and Z. Manna. "Formalization of properties of parallel programs." In: *Machine Intelligence 6*. 1971.

[11] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. "Type Soundness and Race Freedom for Mezzo." In: *Functional and Logic Programming*. Proc. of FLOPS. 2014.

[12] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. "CAVEAT: a Tool for Software Validation." In: *Dependable Systems and Networks*. 2002.

[13] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "A Static Analyzer for Large Safety-Critical Software." In: *Programming Language Design and Implementation*. Proc. of PLDI'03. 2003.

[15] Stephen Brookes. "A Semantics for Concurrent Separation Logic." In: *Theoretical Computer Science* 375.1-3 (Apr. 2007), pp. 227–270.

[16] C. J. Cheney. "A Nonrecursive List Compacting Algorithm." In: *Communication of the ACM* 13.11 (Nov. 1970), pp. 677–678.

[17] Adam Chlipala. "A Verified Compiler for an Impure Functional Language." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'10. 2010.

[18] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[19] Joey W. Coleman and Cliff B. Jones. "A Structural Proof of the Soundness of Rely/guarantee Rules." In: *Journal of Logic and Compution* 17.4 (2007), pp. 807–841.

[20] Collective. *Static Single Assignment Book*. 2015. URL: http://ssabook.gforge.inria.fr/latest/book.pdf.

[21] George E. Collins. "A Method for Overlapping and Erasure of Lists." In: *Communication of the ACM* 3.12 (Dec. 1960), pp. 655–657.

[22] Compcert Development Team. *The CompCert verified compiler, version 3.0*. Available at http://compcert.inria.fr. 2017.

[23] Coq Development Team. *The Coq Reference Manual, version 8.4*. Available at https://coq.inria.fr/distrib/8.4/refman. 2012.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *TOPLAS'91: ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490.

[25] Jared Davis and Magnus O. Myreen. "The Reflective Milawa Theorem Prover is Sound (Down to the Machine Code That Runs It)." In: *Journal of Automated Reasoning* 55.2 (Aug. 2015), pp. 117–183.

[26] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. 2008.

[27] Delphine Demange. "Semantic foundations of intermediate program representations." Theses. École normale supérieure de Cachan - ENS Cachan, Oct. 2012. URL: https://tel.archives-ouvertes.fr/tel-00905442.

[28] J. Derrick, G. Schellhorn, and H. Wehrheim. "Mechanically Verified Proof Obligations for Linearizability." In: *TOPLAS'14: ACM Transactions on Programming Languages and Systems* (2011).

[29] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation." In: *Communication of the ACM* 21.11 (1978), pp. 966–975.

[30] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent Abstract Predicates." In: *European Conference on Object-oriented Programming*. Proc. of ECOOP'10. 2010.

[31] Marko Doko and Viktor Vafeiadis. "A Program Logic for C11 Memory Fences." In: *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'09. 2016.

[32] D. Doligez and G. Gonthier. "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'94. 1994.

[33] D. Doligez and X. Leroy. "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'93. 1993.

[34] Damien Doligez. "Conception, réalisation et certification d'un glaneur de cellules concurrent." PhD thesis. Université Paris 7, May 1995.

[35] T. Domani, E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover. "Implementing an On-the-Fly Garbage Collector for Java." In: *Symposium on Memory management*. Proc. of ISMM'00. 2000.

[36] B. Dongol and J. Derrick. "Verifying Linearisability: A Comparative Survey." In: *CSUR'15: ACM Computing Survey* (Sept. 2015).

[37] Mark Dowson. "The Ariane 5 Software Failure." In: *SIGSOFT Software Engineering Notes* 22.2 (Mar. 1997), pp. 84–.

[38] Eric Eide and John Regehr. "Volatiles Are Miscompiled, and What to Do About It." In: *International Conference on Embedded Software*. EMSOFT'08. 2008.

[39] Facebook. *Infer: a static program analyser for Java, C and Objective-C*. http://fbinfer.com/.

[40] Xinyu Feng. "Local Rely-guarantee Reasoning." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'09. 2009.

[41] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. "On the Relationship Between Concurrent Separation Logic and Assume-guarantee Reasoning." In: *European Conference on Programming*. ESOP'07. 2007.

[42] Robert R. Fenichel and Jerome C. Yochelson. "A LISP Garbage-collector for Virtual-memory Computer Systems." In: *Communication of the ACM* 12.11 (Nov. 1969), pp. 611–612.

[43] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for Concurrent Objects." In: *European Symposium on Programming Languages and Systems*. ESOP '09. York, UK, 2009.

[44] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. "Modelling the ARMv8 architecture, operationally: concurrency and ISA." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'16. 2016.

[45] P. Gammie, A. L. Hosking, and K. Engelhardt. "Relaxing safely: verified on-the-fly garbage collection for x86-TSO." In: *Programming Language Design and Implementation*. Proc. of PLDI 2015. 2015.

[46] G. Gonthier. "Verifying the Safety of a Practical Concurrent Garbage Collector." In: *International Conference on Computer Aided Verification*. Proc. of CAV'96. 1996.

[47] Georges Gonthier. "The Four Colour Theorem: Engineering of a Formal Proof." In: *Computer Mathematics, 8th Asian Symposium*. ASCM'07. 2007.

[48] HOL4 Development Team. *The HOL4 theorem prover, version Kananaskis-11*. Available at https://hol-theorem-prover.org. 2017.

[49] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses." In: *IEEE Symposium on Security and Privacy*. SP'08. 2008.

[50] Per Brinch Hansen. "Monitors and Concurrent Pascal: A Personal History." In: *ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. 1993.

[51] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. "A Practical Multi-word Compare-and-Swap Operation." In: *International Conference on Distributed Computing*. Proc. of DISC'02. 2002.

[52] K. Havelund. "Mechanical Verification of a Garbage Collector." In: *International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Proc. of IPPS/SPDP'99. 1999.

[53] C. Hawblitzel and E. Petrank. "Automated verification of practical garbage collectors." In: *Symposium on Principles of Programming Languages*. Proc of POPL'09. 2009.

[54] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-end Security via Automated Full-system Verification." In: *USENIX Conference on Operating Systems Design and Implementation*. Proc. of OSDI'14. 2014.

[55] Danny Hendler, Nir Shavit, and Lena Yerushalmi. "A Scalable Lock-free Stack Algorithm." In: *ACM Symposium on Parallelism in Algorithms and Architectures*. Proc. of SPAA '04. 2004.

[56] M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *TOPLAS'90: ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492.

[57] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[58] Tony Hoare. "An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580.

[59] Tony Hoare. "Monitors: An Operating System Structuring Concept." In: *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Ed. by Per Brinch Hansen. 2002, pp. 272–294.

[60] Martin Hofmann and Mariela Pavlova. "Elimination of Ghost Variables in Program Logics." In: *Conference on Trustworthy Global Computing*. Proc. of TGC'07. 2008.

[61] Isabelle Development Team. *The Isabelle theorem prover, version 2016-1*. Available at `http://isabelle.in.tum.de`. 2016.

[62] Samin S. Ishtiaq and Peter W. O'Hearn. "BI As an Assertion Language for Mutable Data Structures." In: POPL'01 (2001).

[63] S. Jagannathan, V. Laporte, G. Petri, D. Pichardie, and J. Vitek. "Atomicity Refinement for Verified Compilation." In: *TOPLAS'14: ACM Transactions on Programming Languages and Systems* 36.2 (2014), 6:1–6:30.

[64] C. B. Jones. "Tentative Steps Toward a Development Method for Interfering Programs." In: *TOPLAS'83: ACM Transactions on Programming Languages and Systems* 5.4 (1983), pp. 596–619.

[65] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011.

[66] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.

[67] H.B.M. Jonkers. "A fast garbage-compaction algorithm." In: *Information Processing Letters*. Vol. 9. 1979.

[68] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'15. 2015.

[69] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'18. To appear. 2018.

[70] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris." In: *European Conference on Object-Oriented Programming*. Proc. of ECOOP'17. 2017.

[71] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. "Closing the gap – The formally verified optimizing compiler CompCert." In: *Safety-critical Systems Symposium*. Proc. of SSS'17. 2017.

[72] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive Proofs in Higher-order Concurrent Separation Logic." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'17. 2017.

[73] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic." In: *European Symposium on Programming Languages and Systems*. ESOP'17. 2017.

[74] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML." In: Proc. of POPL'14 (2014).

[75] H. T. Kung and S. W. Song. "An efficient parallel garbage collection system and its correctness proof." In: *IEEE Symposium on Foundations of Computer Science*. 1977.

[76] Ori Lahav and Viktor Vafeiadis. "Owicki-Gries Reasoning for Weak Memory Models." In: *International Colloquium on Automata, Languages, and Programming*. Proc. of ICALP'15. 2015.

[77] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." In: *IEEE Transactions on Computers* 28.9 (Sept. 1979), pp. 690–691.

[78] Chris Arthur Lattner. "LLVM: An infrastructure for multi-stage optimization." PhD thesis. University of Illinois at Urbana-Champaign, 2002.

[79] Lean Development Team. *The Lean theorem prover, version 3.3.0*. Available at https://leanprover.github.io. 2017.

[80] Dirk Leinenbach, Wolfgang J. Paul, and Elena Petrova. "Towards the Formal Verification of a Co Compiler: Code Generation and Implementation Correctness." In: *International Conference on Software Engineering and Formal Methods*. SEFM'05. 2005.

[81] K. Rustan M. Leino. *This is Boogie 2*. 2008.

[82] X. Leroy. "A formally verified compiler back-end." In: *Journal on Automated Reasoning* (2009).

[83] H. Liang and X. Feng. "Modular verification of linearizability with non-fixed linearization points." In: *Programming Language Design and Implementation*. Proc. of PLDI'03. 2013.

[84] H. Liang, X. Feng, and M. Fu. "A Rely-Guarantee-based simulation for verifying concurrent program transformations." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'12. 2012.

[85] Hongjin Liang, Xinyu Feng, and Ming Fu. "Rely-Guarantee-Based Simulation for Compositional Verification of Concurrent Program Transformations." In: *ACM Transactions on Programming Languages and Systems* 36.1 (2014), 3:1–3:55.

[86] Hongjin Liang, Xinyu Feng, and Zhong Shao. "Compositional Verification of Termination-preserving Refinement of Concurrent Programs." In: *Conference on Computer Science Logic and Symposium on Logic in Computer Science*. CSL-LICS '14. 2014.

[87] Hongjin Liang, Jan Hoffmann, Xinyu Feng, and Zhong Shao. "Characterizing Progress Properties of Concurrent Objects via Contextual Refinements." In: *International Conference on Concurrency Theory*. CONCUR'13. 2013.

[88] Henry Lieberman and Carl Hewitt. "A Real-time Garbage Collector Based on the Lifetimes of Objects." In: *Communication of the ACM* 26.6 (June 1983), pp. 419–429.

[89] William Mansky, Yuanfeng Peng, Steve Zdancewic, and Joseph Devietti. "Verifying Dynamic Race Detection." In: *Conference on Certified Programs and Proofs*. Proc. of CPP'17. 2017.

[90] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." In: *Communication of the ACM* 3.4 (Apr. 1960), pp. 184–195.

[91] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. "A Certified Framework for Compiling and Executing Garbage-collected Languages." In: *International Conference on Functional Programming*. Proc. of ICFP '10. 2010.

[92] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. "A General Framework for Certifying Garbage Collectors and Their Mutators." In: *Programming Language Design and Implementation*. Proc. of PLDI'07. 2007.

[93] John Mccarthy and James Painter. *Correctness of a compiler for arithmetic expressions*. 1967.

[94] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms." In: *ACM Symposium on Principles of Distributed Computing*. Proc. of PODC '96. 1996.

[95] Mizar Development Team. *The Mizar System, version 8.1.05*. Available at `http://mizar.org/system/`. 2016.

[96] J. Strother Moore. "A Mechanically Verified Language Implementation." In: *Journal on Automated Reasoning* 5.4 (1989), pp. 461–492.

[97] Strother Moore. *Piton: A Mechanically Verified Assembly-level Language*. Kluwer Academic Publishers, 1996.

[98] F. Lockwood Morris. "Advice on Structuring Compilers and Proving Them Correct." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'73. 1973.

[99] M. O. Myreen. "Reusable Verification of a Copying Collector." In: *Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'10. 2010.

[100] *NULLSTONE – Automated Compiler Performance Analysis*. `http://www.nullstone.com/htmls/category/divide.htm`.

[101] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating State Transition Systems for Fine-Grained Concurrent Resources." In: *European Symposium on Programming*. Proc. of ESOP'14. 2014.

[102] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. "How Amazon Web Services Uses Formal Methods." In: *Communications of the ACM* 58.4 (Mar. 2015), pp. 66–73.

[103] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. "An operational semantics for C/C++11 concurrency." In: *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Proc. of OOPSLA'16. 2016.

[104] Leonor Prensa Nieto. "The Rely-guarantee Method in Isabelle/HOL." In: *European Conference on Programming*. Proc. of ESOP'03. 2003.

[105] Tobias Nipkow and Leonor Prensa Nieto. "Owicki/Gries in Isabelle/HOL." In: *Fundamental Approaches to Software Engineering*. Proc. of FASE'99. 1999.

[106] P. W. O'Hearn. "Separation logic and concurrent resource management." In: *International Symposium on Memory Management*. Proc. of ISMM'07. 2007.

[107] Scott Owens. "Reasoning About the Implementation of Concurrency Abstractions on x86-TSO." In: *European Conference on Object-oriented Programming*. Proc. of ECOOP'10. 2010.

[108] Scott Owens, Susmit Sarkar, and Peter Sewell. "A Better x86 Memory Model: x86-TSO." In: *Theorem Proving in Higher Order Logics, 22nd International Conference, (TPHOLs 2009), Munich, Germany, August 17-20, 2009. Proceedings*. 2009, pp. 391–407.

[109] Susan S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1975.

[110] Susan Owicki and David Gries. "Verifying Properties of Parallel Programs: An Axiomatic Approach." In: *Communications of the ACM* 19.5 (May 1976), pp. 279–285.

[111] PVS Development Team. *The PVS Specification and Verification System, version 6.0*. Available at `http://pvs.csl.sri.com`. 2013.

[112] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java hotspotTM Server Compiler." In: *Symposium on JavaTM Virtual Machine Research and Technology*. JVM'01. 2001.

[113] Matthew Parkinson. "The Next 700 Separation Logics." In: *Verified Software: Theories, Tools, Experiments*. VSTTE'10. 2010.

[114] Dusko Pavlovic, Peter Pepper, and Douglas R. Smith. "Formal Derivation of Concurrent Garbage Collectors." In: *Mathematics of Program Construction*. MPC'10. 2010.

[115] Simon L. Peyton Jones and André L. M. Santos. "A Transformation-based Optimiser for Haskell." In: *Science of Computer Programming* 32.1-3 (Sept. 1998), pp. 3–47.

[116] Simon Peyton Jones and Simon Marlow. "Secrets of the Glasgow Haskell Compiler Inliner." In: *Journal on Functional Programming* 12.5 (July 2002), pp. 393–434.

[117] F. Pfenning and C. Elliott. "Higher-Order Abstract Syntax." In: *Programming Language Design and Implementation*. Proc. of PLDI'88. 1988.

[118] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Version 5.0. http://www.cis.upenn.edu/~bcpierce/sf. Electronic textbook, 2017.

[119] Filip Pizlo and Jan Vitek. "Memory Management for Real-Time Java: State of the Art." In: *Symposium on Object Oriented Real-Time Distributed Computing*. Proc. of ISORC '08. 2008.

[120] Leonor Prensa Nieto. "The Rely-Guarantee Method in Isabelle/HOL." In: *European Symposium on Programming*. ESOP'03. 2003.

[121] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Symposium on Logic in Computer Science*. Proc. of LICS'02. 2002.

[122] Tom Ridge. "A Rely-guarantee Proof System for x86-TSO." In: *International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE'10. 2010.

[123] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. 2012.

[124] Miro Samek. *Are we shooting ourselves in the foot with Stack Overflow?* http://embeddedgurus.com/state-space/2014/02/are-we-shooting-ourselves-in-the-foot-with-stack-overflow/ – Blog post. 2014.

[125] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. "A Verified Generational Garbage Collector for CakeML." In: *Interactive Theorem Proving*. Proc. of ITP'17. 2017.

[126] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. "Understanding POWER Multiprocessors." In: *Programming Language Design and Implementation*. Proc. of PLDI'11.

[127] Robert A. Saunders. "The LISP system for the Q-32 computer." In: *The programming language LISP: its operation and applications*. 1974, pp. 220–231.

[128] I. Sergey, A. Nanevski, and A. Banerjee. "Mechanized verification of fine-grained concurrent programs." In: *Programming Language Design and Implementation*. Proc. of PLDI'15. 2015.

[129] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized Verification of Fine-grained Concurrent Programs." In: *Programming Language Design and Implementation*. Proc. of PLDI'15. 2015.

[130] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. "CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency." In: *Journal of the ACM* 60.3 (2013), p. 22.

[131] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. "A Separation Logic for Fictional Sequential Consistency." In: *European Symposium on Programming*. ESOP'15. 2015.

[132] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. "A New Verified Compiler Backend for CakeML." In: Proc. of ICFP'16 (2016).

[133] *The LLVM Compiler Infrastructure*. Available at `http://llvm.org/`.

[134] Noah Torp-Smith, Lars Birkedal, and John C. Reynolds. "Local Reasoning About a Copying Garbage Collector." In: *ACM Transactions on Programming Languages an Systems* 30.4 (Aug. 2008), 24:1–24:58.

[135] R. K. Treiber. *Systems Programming: Coping with Parallelism.* Tech. rep. RJ 5118. IBM Almaden Research Center, Apr. 1986.

[136] David Ungar. "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm." In: (1984).

[137] V. Vafeiadis. "Modular Fine-Grained Concurrency Verification." PhD thesis. University of Cambridge, July 2007.

[138] V. Vafeiadis and M. J. Parkinson. "A Marriage of Rely/Guarantee and Separation Logic." In: *International Conference on Concurrency Theory*. CONCUR'07. 2007.

[139] Viktor Vafeiadis. "Automatically Proving Linearizability." In: *International Conference on Computer Aided Verification*. Proc. of CAV'10. 2010.

[140] Viktor Vafeiadis and Chinmay Narayan. "Relaxed Separation Logic: A Program Logic for C11 Concurrency." In: *International Conference on Object Oriented Programming Systems Languages and Applications*. Proc. of OOPSLA'13. 2013.

[141] Martin Wildmoser and Tobias Nipkow. "Certifying Machine Code Safety: Shallow versus Deep Embedding." In: *Theorem Proving in Higher Order Logics*. TPHOLs'04. 2004.

[142] Paul R. Wilson. "Uniprocessor Garbage Collection Techniques." In: *International Workshop on Memory Management*. Proc. of IWMM'92. 1992.

[143] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[144] Alexandra Witze. "Software error doomed Japanese Hitomi spacecraft." In: *Nature* 533 (May 2016), pp. 18–19.

[145] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness." In: *Information and Computation* 115.1 (Nov. 1994), pp. 38–94.

[146] Jean Yang and Chris Hawblitzel. "Safe to the Last Instruction: Automated Verification of a Type-safe Operating System." In: *Programming Language Design and Implementation*. Proc. of PLDI'10. 2010.

[147] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers." In: Proc. of PLDI'11 (2011).

[148] Yannick Zakowski, David Cachera, Delphine Demange, Gustavo Petri, David Pichardie, Suresh Jagannathan, and Jan Vitek. "Verifying a Concurrent Garbage Collector using a Rely-Guarantee Methodology." In: *Interactive Theorem Proving*. Proc. of ITP'17. 2017.

[149] Yannick Zakowski, David Cachera, Delphine Demange, and David Pichardie. "Compilation of Linearizable Data Structures - A Mechanised RG Logic for Semantic Refinement." In: *Symposium on Applied Computing*. Proc. of SAC'18. 2018.

[150] Jianzhou Zhao, Santosh Nagarakatte, Milo M Martin, and Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations." In: *Symposium on Principles of Programming Languages*. Proc. of POPL'12. 2012.