Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection with Separation Logic

ALEXANDRE MOINE, Inria, France
ARTHUR CHARGUÉRAUD, Inria & Université de Strasbourg, CNRS, ICube, France
FRANÇOIS POTTIER, Inria, France

We present IrisFit, a Separation Logic with space credits for reasoning about heap space in a concurrent call-by-value λ-calculus equipped with tracing garbage collection and shared mutable state. Space credits, a purely logical device, are consumed when a heap block is allocated and recovered when a block becomes provably unreachable. For each allocated address, "pointed-by-heap" and "pointed-by-thread" assertions record which heap blocks point to this address and which threads hold this address as a root. We point out a fundamental difficulty in the analysis of the worst-case heap space complexity of concurrent programs in the presence of tracing garbage collection: if garbage collection phases and steps of the program's threads can be arbitrarily interleaved, then there exist undesirable scenarios where a root held by a sleeping thread prevents a possibly large amount of memory from being freed. This phenomenon leads to degraded worst-case heap space complexity bounds and to more complex logical specifications: for example, in a naive implementation of Treiber's lock-free stack, one cannot prove that "pop frees up one list cell worth of heap space". To remedy this problem, we propose two language features, namely *protected sections*, where garbage collection is disabled, and *polling points*, instructions that block the current thread if garbage collection has been requested. Protected sections can be exploited by the programmer to eliminate undesirable scenarios and thereby obtain better worst-case heap space complexity. Polling points can be inserted by the compiler to guarantee liveness. The heart of our contribution is IrisFit, a novel program logic that can establish worst-case heap space complexity bounds and whose reasoning rules can take advantage of the presence of protected sections. We construct IrisFit inside the Coq proof assistant on top of the Iris Separation Logic framework. We prove that IrisFit offers both a safety guarantee—programs cannot crash and cannot exceed a heap space limit—and a liveness guarantee—every memory allocation request is satisfied after a bounded number of execution steps by other threads. We illustrate the use of IrisFit via a number of case studies, including a version of Treiber's stack that is correctly decorated with protected sections.

CCS Concepts: • **Theory of computation → Separation logic**; **Program verification**.

Additional Key Words and Phrases: separation logic, tracing garbage collection, concurrency, program verification

ACM Reference Format:
Alexandre Moine, Arthur Charguéraud, and François Pottier. 2018. Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection with Separation Logic. 1, 1 (April 2018), 57 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Program Verification. The most common aim of program verification is to establish the *safety* and *functional correctness* of a program, that is, to prove that this program does not crash and computes a correct result. In the area of deductive program verification [Filliâtre 2011], a program is usually verified with the help of a *program logic*, that is, a set of deduction rules whose logical soundness has been demonstrated once and for all. Separation Logic [Reynolds 2002] and Concurrent Separation Logic [Brookes and O'Hearn 2016; O'Hearn 2019; Jung et al. 2018] are examples of program logics

Authors' addresses: Alexandre Moine, Inria, Paris, France, alexandre.moine@inria.fr; Arthur Charguéraud, Inria & Université de Strasbourg, CNRS, ICube, Strasbourg, France, arthur.chargueraud@inria.fr; François Pottier, Inria, Paris, France, francois.pottier@inria.fr.

2018. ACM XXXX-XXXX/2018/4-ART
https://doi.org/XXXXXXX.XXXXXXX

that allow compositional reasoning (that is, reasoning about a program component in isolation) in the presence of challenging features such as dynamic memory allocation, mutable state, and shared-memory concurrency.

Verification of Resource Bounds. Beyond safety and functional correctness, it may be desirable to establish bounds on *resource consumption*, that is, to prove that the resource requirements of a program (or program component) do not exceed a certain predictable bound. Indeed, a program that requires an unexpectedly large amount of *time* may be unresponsive. A program that requires an unexpectedly large amount of *stack space* may crash with a stack overflow. A program that requires an unexpectedly large amount of *heap space* may exhaust the available memory and make the system unstable.

Assuming that one is able to tell where in the code the resource of interest is consumed and produced, and how much of it is consumed or produced, reasoning about resource consumption can be reduced to reasoning about safety. To do so, one can construct a variant of the program that is instrumented with a *resource meter*, that is, a global variable whose value indicates what amount of the resource remains available. In this instrumented program, one places assertions that cause a runtime failure if the value of the meter becomes negative. If one can verify that the instrumented program is safe, then one has effectively established a bound on the resource consumption of the original program.

The principle of a resource meter has been exploited in many papers, using various frameworks for establishing safety. For instance, Crary and Weirich [2000] exploit a dependent type system; Aspinall et al. [2007] exploit a VDM-style program logic; Carbonneaux et al. [2015] exploit a Hoare logic; He et al. [2009] exploit Separation Logic. The manner in which one reasons about the value of the meter depends on the chosen framework. In the most straightforward approach, the value of the meter is explicitly described in the pre- and postcondition of every function. This is the case, for instance, in He et al.'s work [2009], where two distinct meters are used to measure stack space and heap space. In a more elaborate approach, which is made possible by Separation Logic, the meter is not regarded as an integer value but as a bag of *credits* that can be individually *owned*. This removes the need to refer to the absolute value of the meter: instead, the specification of a function may indicate that this function requires a certain number of credits and produces a certain number of credits.

Verification of Heap Space Bounds, without Garbage Collection. A programming language that does not have garbage collection offers an explicit memory deallocation instruction. Thus, it is easy to tell where heap space is consumed and produced: an allocation instruction consumes the amount of space that it receives as an argument; a deallocation instruction recovers the space occupied by the heap block that is about to be deallocated.

In such a setting, traditional Separation Logic, extended with space credits, can be used to establish verified heap space bounds. To the best of our knowledge, such a variant of Separation Logic does not exist in the literature. However, Hofmann's work on the typed programming language LFPL [2000] can be viewed as a precursor of this idea: LFPL has explicit allocation and deallocation, which consume and produce values of a linear type, written $\diamond$, whose inhabitants behave very much like space credits.

Verification of Heap Space Bounds, with Garbage Collection. In the presence of garbage collection, how does one reason about heap space? In this setting, the programming language does not have a memory deallocation instruction. Thus, it is not evident at which program points space can be reclaimed. A tracing garbage collector (GC) can be invoked at arbitrary points in time, and may deallocate any subset of the *unreachable blocks*. An unreachable block is a block that is not *reachable* from any *root* via a *path* in the heap. Thus, reasoning about heap space in the presence of garbage collection requires somehow reasoning about roots and unreachability.

Madiot and Pottier [2022] make a first step towards addressing this problem. They extend Separation Logic with several concepts. To keep track of free space, they use space credits. They view memory deallocation as a *logical operation*: it is up to the person who verifies the program to decide at which points this operation must be used and which memory blocks must be *logically deallocated*. This decision is subject to a proof obligation: a memory block can be logically deallocated only if it is unreachable. Unfortunately, the concept of unreachability is not local: that is, this concept cannot be easily expressed in terms of Separation Logic assertions. Therefore, Madiot and Pottier rephrase this proof obligation as follows: a memory block can be logically deallocated if it has no predecessors and is not a root. To record the predecessors of every memory block, they use *pointed-by* assertions [Kassios and Kritikos 2013]. To record which blocks are roots, they focus their attention on a low-level language, where the stack is explicitly represented in the heap as a collection of "stack cells". Then, a block is a root if and only if it is a stack cell.

In previous work [Moine et al. 2023], we scale Madiot and Pottier's ideas up to a high-level language, where the stack is implicit. We introduce *Stackable* assertions to implicitly record which memory locations are "invisible roots", that is, which memory locations are roots because they appear in some indirect caller's stack frame.

Neither of these papers focuses on concurrency. Madiot and Pottier [2022] technically support concurrency, but only for a low-level language with stack variables explicitly allocated in the heap, and without any concurrent example covered. Moine et al. [2023] do not support it. By design, their *Stackable* assertion keeps track of a single stack. Extending it with support for multiple stacks is a priori not straightforward.

Verification of Heap Space Bounds, With Garbage Collection and Concurrency. In the present paper, we target a high-level programming language equipped with garbage collection and shared-memory concurrency. In such a setting, multiple threads run concurrently. They share a common heap; each thread has its own implicit stack.

Our initial aim in this project was to *propose a program logic* that allows its user to reason about heap space, and to verify heap space complexity bounds, in a concurrent setting. However, in the course of this work, we came to realize that, unless some care is taken, concurrent programs can have bad worst-case heap space complexity—that is, worse complexity than one might naively imagine. Thus, constructing a program logic to *describe* the worst-case scenarios was not sufficient. Instead, we first propose *programming language features* that let the programmer *eliminate* some of the worst-case scenarios that we have discovered. The new features that we propose include *protected sections*—sections of the code where garbage collection is disabled—and *polling points*—instructions that block the current thread if garbage collection has been requested by one or more other threads. Then, we propose a program logic that lets users exploit the presence of protected sections to establish *improved* worst-case heap space complexity bounds (§3).

Our protected sections and polling points are inspired by mechanisms found in real-world language implementations, such as Ocaml 5's "safe points". However, we believe that our design is better behaved (§11.1, §11.2) and introduces an important distinction between a construct that is inserted by the programmer and that is required to ensure good worst-case heap space complexity (namely, protected sections), and a construct that can be automatically inserted by the compiler and that is required to ensure liveness (namely, polling points).

Contributions. The main contributions of this paper are the following:

- We present LambdaFit (§2, §4), an imperative λ-calculus with shared-memory concurrency and tracing garbage collection. The novel aspects of LambdaFit include protected sections and polling points.
- We introduce IrisFit (§5, §6, §9), a Separation Logic that allows establishing *safety*, *functional correctness*, and *worst-case heap space complexity* properties of concurrent programs, in the presence of garbage collection, and allows *compositional reasoning*.
- We prove the soundness of IrisFit (§7). More specifically, we establish both a *safety* theorem, which guarantees that a verified program cannot crash, and a *liveness* theorem, which guarantees that, provided enough polling points are present, no thread can be forever blocked by a memory allocation request.
- We encode *closures* in LambdaFit and show how to reason about them with IrisFit (§8). Compared with our previous paper [Moine et al. 2023], we propose an improved treatment of closures: the *Spec* predicate, describing the behavior of a closure, is persistent.
- We verify several case studies (§10), namely: an implementation of "fetch-and-add" as a CAS loop; a concurrent counter that is encapsulated as a pair of closures; a library for async/finish parallelism; and Treiber's lock-free stack [1986]. This gallery of challenging examples illustrates the expressive power of IrisFit and how protected sections let us obtain and verify desired heap space complexity bounds.

All of our results, including the validity of our reasoning rules, our soundness theorems, and our case studies, are mechanized using the Coq proof assistant and the Iris framework [Jung et al. 2018]. For details, we refer the reader to our mechanization [Moine 2024].

Because we wish to make the present paper self-contained, we borrow some text from our previous paper [Moine et al. 2023]. The re-used material amounts to roughly 8 pages in total. The main re-used passages are the beginning of this introduction, the design and explanation of the pointed-by-heap assertion (§5.6), the discussion and definition of the closure macros (§2.6, §8.2), the concept and presentation of triples with souvenir (§9), and part of the discussion of the related work (§11.3, §11.4, §11.5).

2 OVERVIEW

LambdaFit is a call-by-value λ-calculus with dynamic memory allocation, mutable state, shared-memory concurrency, and tracing garbage collection. Its syntax and semantics are standard, save for a few original aspects.

First, LambdaFit exhibits a number of non-standard features related with memory management. Its operational semantics defines the concept of a *root* and has explicit garbage collection steps (§2.1). Furthermore, its operational semantics is parameterized with a *maximum heap size*. A memory allocation request that would cause this limit to be exceeded is *blocking* (§2.2). There is a notion of *protected section* where garbage collection cannot take place (§2.3) and a notion of *polling point*, an instruction that blocks the current thread if garbage collection has been requested by other threads (§2.4). After detailing these aspects, we give a high-level overview of how IrisFit provides reasoning rules for all these constructs (§2.5).

Second, LambdaFit is restricted to closed functions, also known as *code pointers*. We encode closures as heap-allocated objects that store code and data (§2.6).

2.1 Roots and Garbage Collection

To be able to talk and reason about the heap space complexity of LambdaFit programs, we must first equip LambdaFit with a semantics where garbage collection is explicit. Garbage collection [Jones and Lins 1996] deallocates some or all unreachable memory blocks, where a block is *reachable* if there exists a path from some *root*, through the heap, to this block. Thus, the semantics of LambdaFit, and the notion of heap space complexity, depend on an answer to the question: what is a root?

How can the intuitive concept of a root be formally defined in the setting of a small-step, substitution-based operational semantics? Before addressing this question, let us recall a few fundamental aspects of such a semantics. In an *operational* semantics, a program state, which represents the state of a running program, is a syntactic object. Here, because we are interested in concurrent programs with dynamic memory allocation, a program state includes a thread pool (a list of threads) and a heap (a finite map of memory locations to memory blocks). In a *small-step* semantics, the manner in which the program state evolves over time is described by a reduction relation, that is, a binary relation on program states. In a *substitution-based* semantics, within the thread pool, each running thread is represented as a closed term, that is, a term without free variables. The reduction rules ensure that, whenever the scope of a variable is entered, a closed value is substituted for this variable. Thus, a closed term that represents a running thread describes both the code that this thread is about to execute and the data to which this thread has access. In particular, a memory location ℓ is a closed value, and a closed term that represents a running thread can contain memory locations.

In such a setting, what is a root? A simple, commonly agreed-upon answer is: *a root is a memory location ℓ that appears in at least one running thread t*. By this, we mean that the closed term t, which represents one of the currently running threads, literally contains one or more occurrences of the memory location ℓ.

This convention is known as the *free variable rule* (FVR) [Felleisen and Hieb 1992; Morrisett et al. 1995]. Intuitively, the FVR states that the (computable) set of memory blocks that are reachable by the locations that appear in threads are a conservative approximation of the (uncomputable) set of memory blocks that might be accessed in the future by any of the threads. However, one must keep in mind that the FVR is not a static approximation *of* the dynamic semantics. Instead, the FVR is *part of* the definition of the dynamic semantics. It defines the concept of root, which in turn is used to define reachability and garbage collection.

The reader may wonder whether real-world programming languages respect the FVR. As far as we know, many real-world implementations of garbage-collected languages, such as OCaml, SML, Haskell, Scala, Java, and more, are meant to respect the FVR. Unfortunately, this intention is often undocumented. A prominent example of a compiler that explicitly respects the FVR is the CakeML verified compiler. Gómez-Londoño et al. [2020] and Gómez-Londoño and Myreen [2021] prove that the CakeML compiler respects a cost model that is defined at the level of the intermediate language DataLang and that includes a form of the FVR.

2.2 Maximum Heap Size and Blocking Memory Allocation

The *default* operational semantics of LambdaFit is parameterized by a maximum heap size S, and is designed in such a way that the heap size always remains less than or equal to S. This property, which is stated by Theorem 4.2 (§4.2.8), is enforced as follows. Let us say that a memory allocation request is *large* if it would cause the heap size to exceed S, that is, if the sum of the current heap size and the number of requested words exceeds S. Otherwise, let us say that the allocation is *small.* Then, a large memory allocation instruction is not allowed to proceed: it is *blocked.* Once garbage collection takes place and is able to free enough space in the heap, this memory allocation instruction may become small, therefore unblocked.

Polling points (§2.4) are another kind of instruction that can be blocked.

By blocking large memory allocation instructions, we ensure that one kind of undesirable behavior, namely *growing the heap too large*, is eliminated a priori. Two kinds of undesirable behavior remain permitted by the operational semantics, namely *crashes* and *deadlocks*: a thread can crash or become forever blocked. Under certain assumptions about the placement of polling points, our program logic statically guarantees that these undesirable behaviors cannot arise: this is stated by our *safety* and *liveness* theorems (Theorems 7.1 and 7.2).

An alternative approach would be to adopt a simpler *oblivious* operational semantics, where no instruction is ever blocked and where there is no space limit. Then, a different kind of undesirable behavior, namely *deadlocks*, is eliminated a priori. The undesirable behaviors that remain permitted by the operational semantics are *crashes* and *growing the heap too large*. In such a setting, our program logic, which is parameterized by an initial amount of available space S, statically provides the following guarantees: first, no thread can crash; second, when every thread is outside a protected section, the live heap space is bounded by S. We define this alternative operational semantics and establish this result: this is our *core soundness* theorem (Theorem 7.3). We use this theorem as a stepping stone in the proof of Theorems 7.1 and 7.2.

Because in the oblivious semantics an instruction is never blocked, whereas in the default semantics the same instruction can be blocked, the oblivious semantics is a superset of the default semantics. A program has a wider set of possible behaviors in the oblivious semantics than in the default semantics. This is why, with respect to the default semantics, our program logic is able to offer a stronger static guarantee. Indeed, with respect to the default semantics, it guarantees that the heap size *never* exceeds S, whereas with respect to the oblivious semantics it guarantees that *when every thread is outside a protected section* the live heap space is at most S.

In summary, there is a choice between two operational semantics for LambdaFit. This choice influences which undesirable behavior is eliminated a priori and which ones are eliminated by the program logic. Because the two semantics are not equivalent (one is a strict subset of the other), this choice is not just a matter of presentation: by choosing the more complex and more restrictive semantics, we are able to offer a simpler and stronger static guarantee.

2.3 Protected Sections

We equip LambdaFit with *protected sections*, that is, sections of the code where garbage collection *cannot* take place. As long as *any* thread is inside a protected section, garbage collection is disabled. Thus, if some thread is blocked by a large memory allocation request (§2.2), then this thread must wait until the GC has been allowed to run, which itself cannot take place until every thread is outside a protected section.

A protected section is explicitly delimited by two special instructions, enter and exit, which mark the beginning and end of the section. A single well-balanced construct "protected $\{t\}$" would be insufficiently flexible, because a protected section typically has one entry point and multiple exit points. This is illustrated by the example of Treiber's stack (Figure 2).

Protected sections are subject to two restrictions. First, they cannot be nested. Second, a protected section must not contain a memory allocation instruction, a "fork" instruction,[1] a polling point (§2.4), or a function call.[2] These restrictions ensure that a protected section cannot contain a blocking instruction and can be exited in a bounded number of steps. The syntax of LambdaFit does not enforce these restrictions; however, violating them causes a runtime error, and is statically forbidden by our program logic.

[1] In our operational semantics, "fork" does not allocate any memory in the heap. We could technically allow "fork" inside a protected section without breaking any of our results. In the real world, though, "fork" is likely to allocate memory. Because we forbid memory allocation inside a protected section, it seems natural to disallow "fork" inside protected sections as well.
[2] Because loops are encoded as recursive functions, forbidding function calls inside protected sections also forbids loops inside protected sections.

Decorating a program with protected sections reduces the set of its possible behaviors: indeed, as long as one thread is inside a protected section, garbage collection cannot take place, so any thread that is in need of a large allocation must wait. Therefore, decorating a program with protected sections can only reduce its worst-case heap space complexity. This phenomenon is illustrated by the example of Treiber's stack (§3).

2.4 Polling Points

The combination of blocking memory allocations (§2.2) and protected sections (§2.3) potentially creates deadlocks, endangering *liveness*: that is, for some programs, there exist adversarial schedules where a large memory allocation request is blocked forever because the GC can never run. For example, imagine that thread A is blocked by a large memory allocation request, while threads B and C both are in an infinite loop whose body contains a protected section. Then, the scheduler can interleave threads B and C in such a way that at all times one of them is inside a protected section, thereby forever disabling garbage collection and blocking thread A. We wish to forbid this scenario and to formally establish a liveness guarantee of the form: *always, eventually, every thread can make progress* (Theorem 7.2).

To this end, we equip LambdaFit with *polling points*. A polling point is a synchronization instruction, a form of barrier. A thread may proceed past a polling point only if no large memory allocation request is currently outstanding. In other words, if any thread is currently blocked by a large memory allocation request, then no thread can move past a polling point. A polling point must not appear inside a protected section.

By inserting sufficiently many polling points into a program, one can ensure that every memory allocation request is eventually satisfied. Indeed, as soon as one thread is blocked on a large memory allocation request, every thread must eventually reach a polling point or a large memory allocation request, where it, too, becomes blocked. At this point, since neither polling points nor memory allocation instructions can appear inside a protected section, every thread must be outside a protected section. Thus, garbage collection can, and must, take place. If enough space becomes available—which our program logic statically guarantees!—then all outstanding memory allocation requests can be satisfied.

In the scenario outlined above, provided a polling point is inserted the loops of both thread B and thread C, these two threads must eventually reach a polling point, where they become blocked. The only permitted step is then a garbage collection step, which is expected to free up enough memory to satisfy thread A's large allocation request. Consequently, all three threads become unblocked.

In principle, polling points could be manually inserted by the programmer, but that would be tedious. In practice, we expect a compiler to automatically insert polling points where needed. In §7.2, we prove that a particular polling point insertion strategy, inspired by that of the OCaml 5 compiler, does indeed insert enough polling points to guarantee liveness.

2.5 A Concurrent Separation Logic for Heap Space

This paper presents IrisFit, a concurrent Separation Logic for LambdaFit. IrisFit shares many features with pre-existing Separation Logics. The behavior of a program fragment is described by a *triple*, an assertion whose parameters include a precondition (an assertion that describes the initial state), the program fragment of interest, and a postcondition (an assertion that describes the final state). In IrisFit, a triple also includes a thread identifier, as the logic assigns a unique name to each thread. A rich vocabulary of logical connectives, including *points-to* assertions, *separating conjunction*, and many more, is used to construct assertions, which encode both *knowledge* of the current state and *permission to update* this state in certain ways.

What sets IrisFit apart from traditional Separation Logics? IrisFit borrows ideas from previous Separation Logics equipped with support for reasoning about heap space in the presence of garbage collection [Madiot and Pottier 2022; Moine et al. 2023] and scales them up to a concurrent setting. *Space credits* keep track of available space and serve as permissions to allocate memory. Furthermore, several kinds of assertions record which memory locations are reachable and in what way they can be reached. *Pointed-by-heap* assertions [Madiot and Pottier 2022] keep track of predecessors of each location in the heap. *Pointed-by-thread* assertions (new in this paper) keep track of the threads in which each location is a root. Like previous logics [Madiot and Pottier 2022; Moine et al. 2023], IrisFit features a *ghost deallocation rule*. Because the programming language does not have an explicit memory deallocation instruction, it is up to the user of the logic to decide where to apply this rule. This rule requires proof that the memory block of interest is unreachable. This proof takes the form of pointed-by-heap and pointed-by-thread assertions, which are consumed; space credits are produced in their stead. A novelty of this paper is that logical deallocation *does not require or consume the points-to assertion.*

A crucial novel aspect of IrisFit is its ability to take advantage of protected sections while reasoning. Indeed, IrisFit offers a relaxed way of keeping track of roots inside protected sections. Ordinarily, pointed-by-thread assertions record which locations are roots, and as long as a location is a root, this location cannot be logically deallocated. Inside a protected section, however, an exception to this regime is made: the logic keeps track of a set of *temporary* roots. The user can turn an ordinary root into a temporary root (and vice-versa). The logic requires that, by the time the protected section ends, no temporary roots remain. Thus, by that time, every temporary root must no longer be a root (or must have been turned back into an ordinary root). Crucially, inside a protected section, the condition under which logical deallocation is permitted is: *if a location ℓ is not an ordinary root in any thread, and if ℓ has no live heap predecessors, then it can be logically deallocated.* In other words, even though physical garbage collection is disabled inside protected sections, logical deallocation remains permitted, and is oblivious to the existence of temporary roots.[3] Finally, perhaps surprisingly, because the points-to assertion survives logical deallocation and enables read and write access, *a temporary root that has already been logically deallocated can still be accessed* before the protected section ends. This pattern appears while verifying lock-free data structures (§10.5).

2.6 Closures

To model the space complexity of programs that involve closures [Landin 1964; Appel 1992], we must somehow reflect the fact that a closure is a heap-allocated object. It has an address, a size, and may hold pointers to other objects. Thus, a closure has both direct and indirect impacts on space complexity: it occupies some space; and, by pointing to other objects, it keeps these objects live (reachable), preventing the GC from reclaiming the space that they occupy.

Thus, we cannot use the standard small-step, substitution-based semantics of the λ-calculus, where a λ-abstraction is a value that does not have an address or a size. Instead, two approaches come to mind. One approach is to view a λ-abstraction as a primitive expression (not a value) whose evaluation causes the allocation of a closure. Another approach is to adopt a restricted calculus that offers only closed functions (as opposed to λ-abstractions with free variables) and to *define* closure construction and closure invocation as *macros*, or canned sequences of instructions, on top of this calculus. As shown by Paraskevopoulou and Appel [2019], these two approaches yield the

[3] Because the GC cannot run while any thread is inside a protected section, it cannot observe the existence of a temporary root. Therefore, there is no reason why the existence of a temporary root should prevent logical deallocation.

```
let create () = ref nil

let rec push s v =
  let h = !s in
  let h' = new_cell () in
  set_data h' v;
  set_tail h' h;
  if compare_and_swap s h h'
  then ()
  else push s v

let rec pop s =
  let h = !s in
  if is_nil h
  then pop s
  else
    let h' = tail h in
    if compare_and_swap s h h'
    then data h
    else pop s
```

Fig. 1. An unsafe-for-space implementation of Treiber's stack

same space cost model. Furthermore, provided suitable syntax is chosen, the end user does not see the difference: it is just a matter of presentation in the metatheory.

We choose the second approach, because we find it simpler. In so doing, we follow Gómez-Londoño et al. [2020], who define the CakeML cost model at the level of DataLang, the language that serves as the target of closure conversion.

Thus, we equip LambdaFit with *closed functions*, which we also refer to as *code pointers*. We write $\mu_{\mathrm{ptr}} f.\, \lambda\vec{x}.\, t$ for a (recursive, multi-argument) closed function, and write $(v\ \vec{u})_{\mathrm{ptr}}$ for the invocation of the code pointer v with arguments $\vec{u}$. LambdaFit does not have primitive closures. This allows us to present a program logic for LambdaFit and to establish the soundness of this logic without worrying about closures. Once this is done, we define *closure construction* $\mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t$ and *closure invocation* $(\ell\ \vec{u})_{\mathrm{clo}}$ as macros, and we extend our program logic with high-level reasoning rules for closures (§8). This allows end users to reason about these macros without expanding them and without even knowing how they are defined. In summary, LambdaFit can macro-express closures, and our logic allows reasoning about closures in the same way as if they were primitive constructs.

Our construction of closures as macros is the same as in our previous paper [Moine et al. 2023]. Our treatment of closures in the logic, however, has been generalized to multiple threads and simplified by describing closures via persistent predicates (§8).

3 WHY TREIBER'S STACK NEEDS PROTECTED SECTIONS

To motivate the interest of protected sections for establishing space bounds, we use the example of Treiber's stack, a lock-free, linearizable stack [Treiber 1986]. We first present a naive implementation of this data structure without protected sections (§3.1). We point out that this implementation has an unsatisfying worst-case heap space complexity: there are scenarios where a successful pop operation does not allow any memory cell to be freed (§3.2). All memory *can* eventually be recovered, but this requires waiting until all threads have completed their interaction with the stack. This situation is unpleasant: pop cannot be given a simple logical specification of the form "a successful pop frees up one list cell worth of heap space". We show that, by annotating the code with protected sections, one can eliminate these undesirable scenarios and obtain the desired specification (§3.3). Near the end of this paper (§10.5), we present the details of how we formally establish this specification in IrisFit.

3.1 Naive Implementation of Treiber's Stack

Treiber's stack is implemented as a mutable reference to an immutable linked list, whose head corresponds to the top of the stack. Pseudo-code is presented in Figure 1.

The function call `create()` creates a new stack, represented as a fresh reference to an empty list `nil`. The `nil` value takes up no heap space: it is in fact an integer value.

The functions push and pop make crucial use of the atomic *compare-and-swap* (CAS) instruction. Each of them is implemented as a "CAS loop": it prepares an operation and attempts to atomically commit this operation using a CAS instruction. If the CAS succeeds, the function returns; otherwise, the loop continues with another attempt. Here, each loop is encoded as a tail-recursive function.

The function push s v inserts a new element v in a stack s. First, s is dereferenced (line 4) so as to obtain the address h of the head of the linked list. Then, a new list cell h' is allocated (line 5). The "data" and "tail" fields are initialized with v (line 6) and h (line 7). Then, a CAS instruction attempts to update the content of s from h to h' (line 8). If this attempt is successful, push returns (line 9); otherwise, it means that a concurrent push or pop has succeeded. In this case, another attempt is made (line 10).

The function pop s extracts the top element of the stack s. First, the head h of the linked list is read (line 12). If the list is empty, pop makes another attempt (line 14), waiting for the stack to become nonempty. Otherwise, the "tail" field of the cell h is read so as to obtain the address h' of the next list cell (line 16). Then, a CAS instruction attempts to update the content of s from h to h' (line 17). If this attempt is successful, pop reads the "data" field of the cell h and returns its value (line 18); otherwise, it means that a concurrent push or pop has succeeded. In this case, another attempt is made (line 19).

Treiber's stack is *linearizable* [Herlihy and Wing 1990], in the sense that push and pop atomically take effect at a certain point between the function call and return.

3.2 Space Consumption of Treiber's Stack without Protected Sections

What is the space consumption of push and pop? Let us write W for the number of memory words occupied by one list cell. A successful push operation consumes W memory words, as it allocates one single list cell. Symmetrically, a successful pop operation should intuitively free up W memory words. Indeed, the list cell being extracted from the list becomes unused, so one might hope that the GC could reclaim it.

However, this intuition is false: when pop returns, although the extracted list cell is indeed unused, it is not necessarily unreachable. Indeed, the extracted list cell might still be a root of other threads that are still in the process of executing a push or pop operation (which is about to fail) on the exact same cell. This issue leads to a problematic worst-case space complexity. Indeed, a thread that holds a list cell as a root causes all descendants of this cell to remain reachable.

A Problematic Scenario and a Solution. Here is a problematic scenario where a cell extracted by a successful pop remains reachable by other threads, preventing its immediate reclamation. Suppose that the stack s consists of a single list cell whose address is ℓ. Suppose that thread A attempts to push a new value onto s, while thread B attempts to pop a value off s. Thread A starts making progress while thread B is asleep. Thread A begins to execute push. At line 4, its local variable h is bound to the address ℓ. At line 5, it allocates a new list cell at address ℓ'; its local variable h' is bound to ℓ'. At line 7, the "tail" field of the new cell is set to ℓ. Then, suppose thread A falls asleep. thread B wakes up and successfully pops one value off the stack. The reference s now stores the value nil. The cell ℓ has been extracted by pop and is no longer logically part of the stack. The cell ℓ' has not yet been inserted by push and is not logically part of the stack.

Because the cell ℓ has been extracted by a pop operation that has successfully completed, one might expect this cell to be now unreachable. However, this is not the case. Thread A has fallen asleep between lines 7 and 8. At this point, the local variables h and h' are still needed in the future: they occur on line 8. Therefore, the locations ℓ and ℓ' are *roots* in thread A. Besides, even if ℓ was not a root, it would still be reachable via the root ℓ', since the "tail" field of the cell ℓ' contains the pointer ℓ. This is problematic: a cell that has been extracted by pop is still reachable after pop has

```
1   let create () = ref nil
2
3   let rec push s v =
4     let h' = new_cell () in
5     set_data h' v;
6     enter; let h = !s in
7     set_tail h' h;
8     if compare_and_swap s h h'
9     then exit
10    else (exit; push s v)
11  let rec pop s =
12  enter; let h = !s in
13  if is_nil h
14  then (exit; pop s)
15  else
16    let h' = tail h in
17    if compare_and_swap s h h'
18    then (let v = data h in exit; v)
19    else (exit; pop s)
```

Fig. 2. A safe-for-space version of Treiber's stack. Protected section entry and exit points are highlighted.

returned. So, *if the GC is invoked at this point, it cannot collect this cell.* Therefore, it is impossible to claim (and to prove) that pop frees up W words of memory!

How can this problem be addressed? A possible approach is to somehow forbid this undesirable behavior. For example, forbidding thread A from falling asleep at this particular point, between lines 7 and 8, might come to mind, but does not seem practical. Instead, we remark that *blocking garbage collection while thread A is asleep at this point* solves the problem, too. If some other thread signals that it needs memory, then, instead of immediately invoking the GC, we suggest to first wait until thread A wakes up, executes the CAS instruction at line 8, and reaches line 10. Recall the scenario that we are considering: thread B has successfully executed pop after the location ℓ was read from s by thread A at line 4. Therefore, the CAS instruction in thread A must fail, and thread A must reach line 10. By this time, the variables h and h' are no longer needed, so the locations ℓ and ℓ' are no longer roots. Moreover, ℓ' does not appear in the heap at all, and ℓ can be reached only via ℓ': therefore, both ℓ and ℓ' are unreachable. If the GC is now allowed to run, then it can reclaim these cells. In this approach, one *can* hope to prove that "pop frees up W words of memory", in the sense that "once pop has returned, as soon as garbage collection is allowed to take place, W words of memory will be freed up".

3.3 Space Consumption of Treiber's Stack with Protected Sections

We introduce protected sections in Treiber's stack to prevent garbage collection between the moment a thread reads the address of the head cells and the moment the CAS operation is executed.

The modified pseudo-code that we propose appears in Figure 2. With respect to the original code in Figure 1, two main changes are made. First, protected sections, delimited by enter and exit instructions, are inserted into push and pop. Second, the allocation of a new list cell in push must be anticipated (moved higher up in the code), because memory allocations are forbidden inside protected sections (§2.3). The protected sections in Figure 2 are placed in such a way that, outside these sections, no list cell that is part of the data structure is a root. Therefore, when garbage collection takes place, necessarily at the time when no thread is inside a protected section, it is the case that no internal list cell is a root. This guarantee is strong enough to allow us to prove that "pop frees up W words of memory". Intuitively, the list cell addresses that are read inside protected sections can be registered in our logic as temporary roots, allowing for their logical deallocation after a successful pop operation. More details about this statement and about its proof are given in (§10.5).

4 SYNTAX AND SEMANTICS OF LAMBDAFIT

In this section, we formally present the syntax of LambdaFit (§4.1) and its small-step reduction relations (§4.2).

$$\begin{array}{lll}
\text{Primitives} & \odot ::= \&\& \mid \| \mid + \mid - \mid \times \mid \div \mid = & \\[1ex]
\text{Values} & v, w ::= () \mid b \in \{\mathsf{false}, \mathsf{true}\} \mid z \in \mathbb{Z} \mid \ell \in \mathcal{L} \mid \mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t & \text{where } \mathit{fv}(t) \subseteq \{f\} \cup \vec{x}
\end{array}$$

$$\begin{array}{llllll}
\text{Terms} & t, u ::= & v & \textit{value} & t[t] & \textit{heap load} \\
 & & x & \textit{variable} & t[t] \leftarrow t & \textit{heap store} \\
 & & \mathsf{let}\, x = t \,\mathsf{in}\, t & \textit{sequencing} & \mathsf{fork}\, t & \textit{thread creation} \\
 & & \mathsf{if}\, t \,\mathsf{then}\, t \,\mathsf{else}\, t & \textit{conditional} & \mathsf{CAS}\, t[t]\, t\, t & \textit{compare-and-swap} \\
 & & (t\ \vec{u})_{\mathsf{ptr}} & \textit{code pointer invocation} & \mathsf{enter} & \textit{entering a protected section} \\
 & & t \odot t & \textit{primitive operation} & \mathsf{exit} & \textit{exiting a protected section} \\
 & & \mathsf{alloc}\ t & \textit{heap allocation} & \mathsf{poll} & \textit{polling point}
\end{array}$$

$$\begin{array}{lllllllll}
\text{Contexts} & K ::= & \mathsf{let}\, x = \square \,\mathsf{in}\, t & \mid & \mathsf{if}\, \square \,\mathsf{then}\, t \,\mathsf{else}\, t & \mid & \square \odot t & \mid & v \odot \square \\
 & & \mathsf{alloc}\ \square & \mid & \square[t] & \mid & v[\square] & \mid & \square[t] \leftarrow t \\
 & & v[\square] \leftarrow t & \mid & v[v] \leftarrow \square & \mid & (\square\ \vec{u})_{\mathsf{ptr}} & \mid & (v\ (\vec{v} \mathbin{+\!\!+} \square \mathbin{+\!\!+} \vec{u}))_{\mathsf{ptr}} \\
 & & \mathsf{CAS}\, \square[t]\, t\, t & \mid & \mathsf{CAS}\, v[\square]\, t\, t & \mid & \mathsf{CAS}\, v[v]\, \square\, t & \mid & \mathsf{CAS}\, v[v]\, v\, \square
\end{array}$$

Fig. 3. LambdaFit: syntax

4.1 Syntax

The syntax of LambdaFit appears in Figure 3. A *value* v is a piece of data that fits in one word of memory. A value can be the unit value $()$, a Boolean value b, an integer value z, a memory location ℓ (drawn from an infinite set $\mathcal{L}$), or a code pointer $\mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t$. Such a code pointer is a closed, recursive, multi-argument function. The side condition $\mathit{fv}(t) \subseteq \{f\} \cup \vec{x}$ ensures that the function is closed: that is, the only variables that may appear in the body of the function are f (a self-reference, allowing the function to invoke itself) and $\vec{x}$ (the formal parameters).

The syntax of terms (also known as expressions) includes a number of standard sequential constructs, such as sequencing, conditionals, code pointer invocations, and primitive operations. The heap allocation expression alloc n allocates a fresh memory block of size n and returns its address. The field at offset i in the memory block at address x is read by the "load" expression $x[i]$ and written by the "store" expression $x[i] \leftarrow y$.

Two standard concurrency-related constructs are "fork" and CAS. The expression fork t spawns a new thread whose code is t. The compare-and-swap expression CAS $\ell[i]\ v\ v'$ atomically loads a value from block ℓ at offset i, compares this value with v, and, in case they are equal, overwrites this value with v'. Its Boolean result indicates whether the write took place.

The instructions enter and exit mark the beginning and end of a protected section (§2.3). The poll instruction is a polling point (§2.4).

4.2 Semantics

We now define the operational semantics of LambdaFit. We begin with our model of memory, that is, our view of the heap as a collection of memory blocks, and our notion of heap size (§4.2.1). We define thread pools and configurations (§4.2.2). Then, we introduce a series of reduction relations which, together, form the dynamic semantics of LambdaFit. The *head reduction* relation (§4.2.3) describes one elementary step of computation by one thread. The *step* relation (§4.2.4) allows head reduction to take place under an evaluation context. It represents one step of computation by one thread. The *garbage collection* relation (§4.2.5) describes the effect of the GC on the heap. The *action* relation (§4.2.6) and the *main reduction* relation (§4.2.8) describe the evolution of a complete system.

$$\textsc{HeadLetVal}\qquad \text{let}\; x = v \;\text{in}\; t \,/\, g \,/\, \sigma \xrightarrow{\text{head}} [v/x]t \,/\, g \,/\, \sigma \,/\, \varepsilon$$

$$\textsc{HeadCall}\qquad \frac{v = \mu_{\text{ptr}} f.\,\lambda\vec{x}.\,t \qquad |\vec{x}| = |\vec{w}|}{(v\;\vec{w})_{\text{ptr}} \,/\, \text{Out} \,/\, \sigma \xrightarrow{\text{head}} [v/f][\vec{w}/\vec{x}]t \,/\, \text{Out} \,/\, \sigma \,/\, \varepsilon}$$

$$\textsc{HeadIfTrue}\qquad \begin{array}{l} \quad\text{if true then}\; t_1 \;\text{else}\; t_2 \,/\, g \,/\, \sigma \\ \xrightarrow{\text{head}} t_1 \,/\, g \,/\, \sigma \,/\, \varepsilon \end{array}$$

$$\textsc{HeadIfFalse}\qquad \begin{array}{l} \quad\text{if false then}\; t_1 \;\text{else}\; t_2 \,/\, g \,/\, \sigma \\ \xrightarrow{\text{head}} t_2 \,/\, g \,/\, \sigma \,/\, \varepsilon \end{array}$$

$$\textsc{HeadEnter}\qquad \begin{array}{l} \quad\text{enter} \,/\, \text{Out} \,/\, \sigma \\ \xrightarrow{\text{head}} () \,/\, \text{In} \,/\, \sigma \,/\, \varepsilon \end{array}$$

$$\textsc{HeadExit}\qquad \begin{array}{l} \quad\text{exit} \,/\, \text{In} \,/\, \sigma \\ \xrightarrow{\text{head}} () \,/\, \text{Out} \,/\, \sigma \,/\, \varepsilon \end{array}$$

$$\textsc{HeadPrim}\qquad \frac{v_1 \odot v_2 \xrightarrow{\text{pure}} v}{v_1 \odot v_2 \,/\, g \,/\, \sigma \xrightarrow{\text{head}} v \,/\, g \,/\, \sigma \,/\, \varepsilon}$$

$$\textsc{HeadAlloc}\qquad \frac{\ell \notin dom(\sigma) \qquad 0 < n \qquad \sigma' = [\ell := ()^n]\sigma}{\text{alloc}\; n \,/\, \text{Out} \,/\, \sigma \xrightarrow{\text{head}} \ell \,/\, \text{Out} \,/\, \sigma' \,/\, \varepsilon}$$

$$\textsc{HeadLoad}\qquad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\ell[i] \,/\, g \,/\, \sigma \xrightarrow{\text{head}} v \,/\, g \,/\, \sigma \,/\, \varepsilon}$$

$$\textsc{HeadStore}\qquad \frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \sigma' = [\ell := [i := v]\vec{w}]\sigma}{\ell[i] \leftarrow v \,/\, g \,/\, \sigma \xrightarrow{\text{head}} () \,/\, g \,/\, \sigma' \,/\, \varepsilon}$$

$$\textsc{HeadCASFailure}\qquad \frac{\begin{array}{c}\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \\ \vec{w}(i) \neq v\end{array}}{\text{CAS}\; \ell[i]\; v\; v' \,/\, g \,/\, \sigma \xrightarrow{\text{head}} \text{false} \,/\, g \,/\, \sigma \,/\, \varepsilon}$$

$$\textsc{HeadCASSuccess}\qquad \frac{\begin{array}{c}\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \\ \vec{w}(i) = v \qquad \sigma' = [\ell := [i := v']\vec{w}]\sigma\end{array}}{\text{CAS}\; \ell[i]\; v\; v' \,/\, g \,/\, \sigma \xrightarrow{\text{head}} \text{true} \,/\, g \,/\, \sigma' \,/\, \varepsilon}$$

$$\textsc{HeadPoll}\qquad \text{poll} \,/\, \text{Out} \,/\, \sigma \xrightarrow{\text{head}} () \,/\, \text{Out} \,/\, \sigma \,/\, \varepsilon$$

$$\textsc{HeadFork}\qquad \text{fork}\; t \,/\, \text{Out} \,/\, \sigma \xrightarrow{\text{head}} () \,/\, \text{Out} \,/\, \sigma \,/\, t$$

Fig. 4. The head reduction relation

$$\textsc{StepHead}\qquad \frac{t \,/\, g \,/\, \sigma \xrightarrow{\text{head}} t' \,/\, g' \,/\, \sigma' \,/\, t^?}{t \,/\, g \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^?}$$

$$\textsc{StepCtx}\qquad \frac{t \,/\, g \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^?}{K[t] \,/\, g \,/\, \sigma \xrightarrow{\text{step}} K[t'] \,/\, g' \,/\, \sigma' \,/\, t^?}$$

Fig. 5. The step relation

$$\textsc{Edge}\qquad \frac{\sigma(\ell) = \vec{w} \qquad \vec{w}(i) = \ell'}{\ell \rightsquigarrow_\sigma \ell'}$$

$$\textsc{GC}\qquad \frac{dom(\sigma') = dom(\sigma) \qquad \begin{array}{l} \forall \ell.\; \ell \in dom(\sigma) \implies \\ \left\{\begin{array}{l} \quad \sigma'(\ell) = \sigma(\ell) \\ \vee\; \sigma'(\ell) = \lightning \;\wedge\; \neg\,(\exists r \in R,\;\; r \rightsquigarrow^*_\sigma \ell) \end{array}\right. \end{array}}{R \vdash \sigma \xrightarrow{\text{gc}} \sigma'}$$

Fig. 6. The garbage collection relation

There, each step is either a garbage collection step or a step of one thread. The main reduction relation is obtained from the action relation by restricting it to a subset of *enabled* actions (§4.2.7).

4.2.1 Memory Blocks, Stores, and Heap Size. A *memory block* is either a tuple of values, written $\vec{v}$, or a special deallocated block, written $\lightning$. A *store* σ (or *heap*) is a finite map of locations to memory blocks. We write $\emptyset$ for the empty store.

Our semantics does not recycle memory locations. When a heap block at address ℓ is reclaimed by the GC, the store is updated with a mapping of ℓ to $\lightning$. The address ℓ continues to exist and is never re-used. Naturally, in an implementation, memory locations would be recycled. However, we work at a higher level of abstraction. The reasoning rules of our program logic guarantee that a memory allocation always produces a fresh address.

We assume that the space usage (in words) of a block of n fields is $\mathit{size}(n)$, where size is a mathematical function of $\mathbb{N}$ to $\mathbb{N}$. If, for instance, every memory block is preceded by a one-word header, then the function size would be defined by $\mathit{size}(n) = n + 1$. LambdaFit and IrisFit are independent of the definition of size. For our case studies (§10), we chose $\mathit{size}(n) = n$. We write $\mathit{size}(\vec{v})$ as a shorthand for $\mathit{size}(n)$, where n is the length of the list $\vec{v}$. By convention, we let $\mathit{size}(\lightning)$ be 0. This reflects the fact that a deallocated block occupies no space.

We define the size of a store σ as the sum of the sizes of its blocks. Thus, we do not measure the physical size of the heap, that is, how much memory has been borrowed from the operating system. Instead, we measure the total size of the memory blocks that are currently allocated. We ignore fragmentation.

4.2.2 Thread Pools and Configurations. A *thread* t is just a term. A thread's *status* g is either In or Out. The status records whether the thread is currently inside or outside a protected section. A *thread pool* θ is a list of pairs (t, g) of a thread t and its status g. A *thread identifier* π is an integer index into a thread pool.

A *configuration* c is a pair (θ, σ) of a thread pool θ and a store σ. The *initial configuration* for a program t consists of a thread pool that contains just the thread (t, Out) and the empty store $\emptyset$. We write $\mathit{init}(t)$ for this initial configuration. We define the heap size of a configuration as the size of its store: $\mathit{size}((\theta, \sigma)) = \mathit{size}(\sigma)$.

4.2.3 The Head Reduction Relation. The *head reduction* relation $t \,/\, g \,/\, \sigma \xrightarrow{\text{head}} t' \,/\, g' \,/\, \sigma' \,/\, t^?$ describes an evolution of the term t with status g and store σ to a term t' with status g' and store σ', optionally forking off a new thread $t^?$. The metavariable $t^?$ denotes an optional term: it is either a term t or ε, which means that no thread was forked off.

The head reduction relation describes how an instruction is executed under the assumption that this instruction is *enabled*, that is, not blocked. The definition of enabled instructions, which describes under what conditions an instruction is blocked, is given later on (§4.2.7).

The head reduction relation is defined by the rules in Figure 4.

HeadLetVal, HeadIfTrue, HeadIfFalse, HeadPrim are standard.

HeadLoad, HeadStore, HeadCASSuccess and HeadCASFailure, which describe memory accesses, are also standard. These rules require that the memory location ℓ be valid: this is expressed by the premise $\sigma(\ell) = \vec{w}$. Furthermore, they require the integer value i to be a valid index into the memory block at address ℓ: this is expressed by the premise $0 \leq i < |\vec{w}|$. We write $\vec{w}(i)$ for the i-th value in the sequence $\vec{w}$, and $[i{:=}v]\vec{w}$ for the sequence obtained by updating the sequence $\vec{w}$ at index i with the value v. We write $[\ell{:=}\vec{w}]\sigma$ for the store obtained by updating the store σ at address ℓ with the block $\vec{w}$. Hence, $[\ell{:=}[i{:=}v]\vec{w}]\sigma$ describes an update of the i-th field of the block at location ℓ.

HeadEnter and HeadExit cause the thread to change its status from Out to In and vice-versa. By design, no reduction rule describes the effect of enter when the thread's status is In or the effect of exit when the thread's status is Out. Such a situation is considered a runtime error: the thread is *stuck*.

HeadCall, HeadAlloc, HeadFork, and HeadPoll require the thread's status to be Out. Thus, inside a protected section, a function call, a memory allocation request, a "fork" instruction, or a polling point causes a runtime error. Aside from this, HeadCall and HeadFork are standard.

$$\textsc{ActionThread}\quad \frac{\theta(\pi) = (t, g) \qquad t \,/\, g \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^? \qquad \theta' = [\pi := (t', g')]\theta +\!\!+ [(t^?, \mathrm{Out})]}{(\theta, \sigma) \xrightarrow{\text{action}}_{\pi} (\theta', \sigma')}$$

$$\textsc{ActionGC}\quad \frac{locs(\theta) \vdash \sigma \xrightarrow{\text{gc}} \sigma' \qquad \sigma \neq \sigma'}{(\theta, \sigma) \xrightarrow{\text{action}}_{\text{gc}} (\theta, \sigma')}$$

Fig. 7. The action relation

HeadAlloc allocates a block of n fields at a fresh memory location and initializes each field with a unit value. We write $()^n$ for a sequence of n unit values. HeadPoll indicates that a polling point is a no-operation: poll acts as a form of barrier (§4.2.7), and is otherwise effectless.

4.2.4 The Step Relation. The *step* relation has the same shape as the head reduction relation (§4.2.3). It takes the form $t \,/\, g \,/\, \sigma \xrightarrow{\text{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^?$. It is inductively defined by the rules StepHead and StepCtx in Figure 5. These rules allow one head reduction step under a stack of evaluation contexts. An *evaluation context* K is a term with a hole written □ at depth exactly 1. The syntax of evaluation contexts, presented in Figure 3, dictates a left-to-right, call-by-value evaluation strategy. We write $K[t]$ for the term obtained by filling the hole of the evaluation context K with the term t.

4.2.5 The Garbage Collection Relation. Several concepts related with garbage collection are defined in Figure 6. The *edge* relation $\ell \rightsquigarrow_\sigma \ell'$, defined by the rule Edge, means that the block at location ℓ contains a pointer to location ℓ'.[4] When this relation holds, we say that ℓ is a *predecessor* of ℓ'. The *reachability* relation $\ell \rightsquigarrow^*_\sigma \ell'$ is the reflexive-transitive closure of the edge relation.

The *garbage collection* relation $R \vdash \sigma \xrightarrow{\text{gc}} \sigma'$, defined by the rule GC, describes the effect of the GC. This relation means that a garbage collection phase can transform the store σ into a store σ', while respecting the set of roots R, a set of memory locations. This relation is non-deterministic: the GC may reclaim any unreachable memory block, but need not reclaim every such block. According to the first premise of the rule GC, the stores σ and σ' have the same domain: garbage collection does not create or destroy any memory locations. According to the second premise, at each memory location ℓ, either nothing happens ($\sigma'(\ell) = \sigma(\ell)$) or a memory block becomes deallocated ($\sigma'(\ell) = \lightning$). The second case is permitted only if ℓ is not reachable from any of the roots in the set R.

4.2.6 The Action Relation. The relations defined so far describes how a thread makes a step (§4.2.4) and how the GC makes a step (§4.2.5). We now define a relation that interleaves these two kinds of steps. It is a labeled transition relation: each step is labeled with an *action* a, which is either a thread identifier π or the fixed token "gc". The *action* relation $c \xrightarrow{\text{action}}_a c'$ relates two configurations c and c' and is labeled with an *action*. It is defined by the two rules in Figure 7. ActionThread allows a step by one thread whose identifier is π. This thread evolves from (t, g) to (t', g'): the thread pool is updated accordingly. The heap, which is shared between all threads, evolves from σ to σ'. A new thread $t^?$ possibly appears: if so, the thread pool is extended with the new entry $(t^?, \mathrm{Out})$. ActionGC describes a garbage collection step. The roots provided to the GC are $locs(\theta)$, that is, the locations that occur in the thread pool: this is the FVR (§2.1). The side condition $\sigma \neq \sigma'$ prevents the GC from stuttering. We would otherwise not be able to prove that every thread is eventually able to make progress (Theorem 7.2).

4.2.7 Enabled Actions. Two LambdaFit instructions possibly have a blocking behavior: a large memory allocation instruction is blocking (§2.2); if a large memory allocation request is outstanding, then a polling point is blocking (§2.4). Furthermore, while any thread is inside a protected section,

[4] A value either *is* a location or contains no location at all. Thus, in Edge, we write just $\vec{w}(i) = \ell'$ instead of the seemingly more general condition $\ell' \in locs(\vec{w}(i))$.

$$\textsc{IsAllocHead}\quad \mathsf{IsAlloc}\ n\ (\mathsf{alloc}\ n) \qquad \textsc{IsAllocCtx}\quad \frac{\mathsf{IsAlloc}\ n\ t}{\mathsf{IsAlloc}\ n\ (K[t])} \qquad \textsc{IsPollHead}\quad \mathsf{IsPoll}\ \mathsf{poll} \qquad \textsc{IsPollCtx}\quad \frac{\mathsf{IsPoll}\ t}{\mathsf{IsPoll}\ (K[t])}$$

$$\textsc{AllocFits}\quad \frac{\forall n.\ \mathsf{IsAlloc}\ n\ t \implies size(\sigma) + n \leq S}{\mathsf{AllocFits}\ \sigma\ t} \qquad \textsc{EveryAllocFits}\quad \frac{\forall t\, g.\ (t, g) \in \theta \implies \mathsf{AllocFits}\ \sigma\ t}{\mathsf{EveryAllocFits}\ (\theta, \sigma)}$$

$$\textsc{EnabledThread}\quad \frac{\begin{array}{c} c = (\theta, \sigma) \qquad \theta(\pi) = (t, g) \\ \mathsf{AllocFits}\ \sigma\ t \\ \mathsf{IsPoll}\ t \implies \mathsf{EveryAllocFits}\ c \end{array}}{\mathsf{Enabled}\ c\ \pi} \qquad \textsc{AllOutside}\quad \frac{\forall t\, g.\ \ (t, g) \in \theta \implies g = \mathsf{Out}}{\mathsf{AllOutside}\ (\theta, \sigma)} \qquad \textsc{EnabledGC}\quad \frac{\mathsf{AllOutside}\ c}{\mathsf{Enabled}\ c\ \mathsf{gc}}$$

Fig. 8. Enabled actions (and auxiliary predicates)

$$\textsc{EnabledAction}\quad \frac{\mathsf{Enabled}\ c\ a \qquad c \xrightarrow{\text{action}}_a c'}{c \xrightarrow{\text{enabled action}}_a c'} \qquad \textsc{Main}\quad \frac{c \xrightarrow{\text{enabled action}}_a c'}{c \xrightarrow{\text{main}} c'}$$

Fig. 9. The main reduction relation

garbage collection is disabled (§2.3). To reflect these aspects, we now wish to define under what conditions an action is *enabled* (allowed to proceed) or *disabled* (blocked).

The distinction between *small* and *large* memory allocation requests depends on the maximum heap size S (§2.2): Therefore, the notion of enabled action depends on the parameter S, and so does the main reduction relation, which is defined in the next subsection (§4.2.8).

To define enabled actions, a few auxiliary predicates are needed. They appear in Figure 8.

The proposition IsAlloc $n\ t$ means that the next instruction of the thread t is "alloc n". In other words, this thread is now requesting a new memory block of n fields. Similarly, the proposition IsPoll t means that the next instruction of the thread t is "poll".

The proposition AllocFits $t\ \sigma$ means that, if the next instruction in thread t is an allocation request, then it is a small one: that is, there is currently enough free space in the store σ to satisfy it. When this is the case, we say that *thread t fits*. The proposition EveryAllocFits c means that, in the configuration c, every thread fits.

The proposition Enabled $c\ a$ means that, in the configuration c, action a is enabled. It is defined by the rules EnabledThread and EnabledGC in Figure 8. For a thread π to be enabled, it must be the case that (1) thread π fits and (2) if thread π is at a polling point then every thread fits. For garbage collection to be enabled, it must be the case that every thread is currently outside a protected section.

The following simple lemma states that if every thread fits then every action is enabled. It is used in the proof of our liveness theorem (§7.2).

Lemma 4.1 (All Enabled). *If EveryAllocFits c holds, then, for every thread identifier π that is valid with respect to the configuration c, Enabled $c\ \pi$ holds.*

4.2.8 The Main Reduction Relation. The auxiliary relation $c \xrightarrow{\text{enabled action}}_a c'$, defined in Figure 9, is the restriction of the action relation to enabled actions. The *main reduction* relation $c \xrightarrow{\text{main}} c'$ is obtained from this auxiliary relation by abstracting away the action a. Thus, a step in the main reduction relation corresponds to an enabled action by some thread or by the GC.

By design of our semantics, the maximum heap size S is never exceeded. This is an immediate consequence of the fact that large memory allocation requests are blocked.

Theorem 4.2 (Heap Size). *If* $size(c) \leq S$ *and* $c \xrightarrow{\text{main}} c'$ *then* $size(c') \leq S$.

This simple theorem is not used anywhere; it serves to document the design of the semantics.

5 PROGRAM LOGIC: ASSERTIONS

This section offers an overview of the various kinds of assertions that play a role in IrisFit. We introduce the syntax of each assertion, its intuitive meaning, and the ghost reasoning rules that help understand this meaning, such as splitting and joining rules. We informally explain the life cycle of each assertion: where it typically appears, where it is exploited, and where it is consumed. A presentation of the reasoning rules for terms is deferred to the following section (§6).

We begin with a presentation of triples (§5.1) and ghost updates (§5.2). Then, we briefly present the standard points-to assertion (§5.3), the novel "*sizeof*" assertion (§5.4), and space credits (§5.5). We then devote our attention to the assertions that record reachability or unreachability information, namely the pointed-by-heap assertion (§5.6), the novel pointed-by-thread assertion (§5.7), the novel "*inside*" and "*outside*" assertions (§5.8), and deallocation witnesses (§5.9). Finally, we explain liveness-based cancellable invariants (§5.10), a useful idiom that expresses that a certain invariant holds as long as a certain location is live.

IrisFit is a variant of the Iris program logic [Jung et al. 2018, §6–7], and is built on top of the Iris base logic [Jung et al. 2018, §5]. We reuse a large amount of Iris's standard notation. In particular, we write Φ for assertions, $\ulcorner P \urcorner$ for a pure assertion, $\Phi * \Phi'$ for a separating conjunction, and $\Phi \mathrel{-\!\!*} \Phi'$ for a separating implication. We express the logical equivalence of two assertions as $\Phi \equiv \Phi'$. A postcondition Ψ is a function of a value to an assertion: in other words, it is the form $\lambda v.\, \Phi$.

5.1 Triples

A triple takes the form $\{\Phi\}\ \pi\colon t\ \{\Psi\}$. Its intuitive meaning is that if the store satisfies the assertion Φ then it is safe for thread π to execute the term t; furthermore, if and when this computation terminates and produces a value v, then the store satisfies the assertion $\Psi\ v$.

Even though the main reduction relation (§4.2.8) is parameterized with a maximum heap size S, the meaning of triples is independent of S. Indeed, triples are internally defined in terms of the *oblivious* reduction relation (§7.3), which does not depend on S. Therefore, none of the reasoning rules mentions S. Our program logic is compositional: each program component can be verified in isolation and without knowledge of S.

Formally, a triple is also parameterized by a *mask* [Jung et al. 2018, §2.2]. Masks prevent the user from opening an invariant twice. As our treatment of invariants and masks is standard, we omit masks everywhere. The interested reader is referred to our mechanization [Moine 2024].

We write $\{\Phi\}\ \pi\colon t\ \{\lambda\ell.\, \Phi'\}$, where the metavariable ℓ denotes a memory location, as syntactic sugar for $\{\Phi\}\ \pi\colon t\ \{\lambda v.\ \exists\ell.\ \ulcorner v = \ell \urcorner * \Phi'\}$. We adopt the convention that multi-line assertions are implicitly joined by a separating conjunction.

5.2 Ghost Updates

Iris features *ghost state* and *ghost updates* [Jung et al. 2018, §5.4]. A ghost update is written $\Phi \Rrightarrow \Phi'$. It is an assertion, which means that (up to an update of the ghost state) the assertion Φ can be transformed into Φ'.

$$\textsc{Consequence}\qquad\frac{\Phi \;{}^{\pi}{\Rrightarrow}^{\mathit{locs}(t)}\; \Phi' \qquad \{\Phi'\}\;\pi\colon t\;\{\Psi'\} \qquad \forall v.\; \Psi'\,v \;{}^{\pi}{\Rrightarrow}^{\mathit{locs}(v)}\; \Psi\,v}{\{\Phi\}\;\pi\colon t\;\{\Psi\}}$$

$$\textsc{Frame}\qquad\frac{\{\Phi\}\;\pi\colon t\;\{\Psi\}}{\{\Phi \;*\; \Phi'\}\;\pi\colon t\;\{\lambda v.\,\Psi\,v \;*\; \Phi'\}}$$

Fig. 10. Structural reasoning rules

$$\begin{array}{rclr}
 & \ell \mapsto_p \vec{w} & \twoheadrightarrow \quad \ell \mapsto_p \vec{w} \;*\; \mathit{sizeof}\;\ell\;(\mathit{size}(\vec{w})) & \textsc{SizeOfPointsTo} \\
\mathit{sizeof}\;\ell\;n \;*\; & \mathit{sizeof}\;\ell\;m & \twoheadrightarrow \quad \ulcorner n = m \urcorner & \textsc{SizeOfConfront} \\
 & \mathit{sizeof}\;\ell\;n \text{ is persistent} & & \textsc{SizeOfPersist}
\end{array}$$

Fig. 11. Reasoning rules of the "*sizeof*" assertion

In IrisFit, it is sometimes necessary for a ghost update to refer to "the identifier of the current thread" or to "the roots of the current thread". For this purpose, we introduce a *custom ghost update*, written $\Phi \;{}^{\pi}{\Rrightarrow}^{V}\; \Phi'$, whose extra parameters are a thread identifier π and a set of memory locations V. It is strictly more powerful than a standard ghost update: the law $(\Phi \Rrightarrow \Phi') \twoheadrightarrow (\Phi \;{}^{\pi}{\Rrightarrow}^{V}\; \Phi')$ is valid.

Custom ghost updates are exploited in the Consequence rule, which appears in Figure 10. This rule allows strengthening the precondition and weakening the postcondition of a triple. Updating the precondition requires a custom ghost update where the parameter V is instantiated with $\mathit{locs}(t)$. Indeed, this set represents the roots at the point where this update takes place. Updating the postcondition requires a custom ghost update where V instantiated with $\mathit{locs}(v)$, where v denotes the value of the term t. Indeed, these are the roots at the point where that update takes place.

When a custom ghost update is independent of the parameters π and V, we omit them: we write $\Phi \Rrightarrow \Phi'$ for $\forall \pi\, V.\; \Phi \;{}^{\pi}{\Rrightarrow}^{V}\; \Phi'$. Examples of custom ghost updates appear in Figures 14, 15, and 16 and are discussed in the following sections.

The Frame rule, also shown in Figure 10, retains its standard form.

5.3 Points-to Assertions

IrisFit features standard points-to assertions of the form $\ell \mapsto_p \vec{w}$, where p is either a fraction in the semi-open interval $(0, 1]$ or the *discarded fraction* $\square$ [Vindum and Birkedal 2021]. In the latter case, the points-to assertion is persistent.

Rules. Points-to assertions can be split and joined in the usual way, and a points-to assertion that carries a fraction p can be permanently transformed into one that carries the discarded fraction $\square$. We do not show these standard rules.

Life cycle. A points-to assertion appears when a memory block is allocated. It is required (and possibly updated) when this block is accessed by a load, store, or CAS instruction (§6.2). It is *not* required or consumed when this block is logically deallocated (§6.1). This is an original feature of IrisFit.

5.4 Sizeof Assertions

The assertion $\mathit{sizeof}\;\ell\;n$ means that there is or there used to be a block of size n at address ℓ. It is persistent: indeed, once the size of a block has been fixed, it can never be changed.

Rules. Two reasoning rules allow introducing and exploiting "*sizeof*" assertions (Figure 11). SizeOfPointsTo creates a "*sizeof*" assertion out of a points-to assertion. SizeOfConfront states that two "*sizeof*" assertions for the same address must agree on the size of the block at this address.

$$\begin{array}{rcll} \ulcorner \mathrm{True} \urcorner & \Rrightarrow & \Diamond 0 & \textsc{ZeroSC} \\ \Diamond(n_1 + n_2) & \equiv & \Diamond n_1 \ * \ \Diamond n_2 & \textsc{SplitJoinSC} \end{array}$$

Fig. 12. Reasoning rules for space credits

Life cycle. The "*sizeof*" assertion is produced by SizeOfPointsTo. It is consulted by the logical deallocation rules (§6.1, §6.6) to determine the number of space credits that must be produced.

5.5 Space Credits

To reason about free space, we use *space credits* [Madiot and Pottier 2022; Moine et al. 2023]. The assertion $\Diamond n$ denotes the unique ownership of n space credits. It can be understood as a permission to allocate n words of memory. At a lower level of understanding, this assertion means that n memory words *are currently free or can be freed* by the GC *once it is given a chance to run.* This interpretation of space credits is the same as the earlier papers cited above; however, in these previous papers, garbage collection was allowed to take place at any time, whereas in the present paper, garbage collection is enabled only when all threads are outside protected sections.

Following Moine et al. [2023], space credits are measured using non-negative *rational* numbers. Of course, a physical word of memory cannot be split, so the total number of space credits in existence is a natural number; so are the numbers involved in the reasoning rules for memory allocation and deallocation. Still, rational numbers appear essential in certain amortized complexity analyses, as illustrated by the example of chunked stacks [Moine et al. 2023]. Rational credits also appear in amortized *time* complexity analyses [Charguéraud and Pottier 2019; Mével et al. 2019].

Rules. Figure 12 presents two basic reasoning rules about space credits. ZeroSC asserts that zero credits can be forged out of thin air. SplitJoinSC asserts that space credits can be split and joined.

Life cycle. Space credits are consumed by memory allocation (§6.2) and produced by logical deallocation (§6.1). Because there is no way of creating space credits out of nothing, a program or program component is usually verified under the assumption that a number of space credits are provided. This is apparent in the statement of our safety theorem (§7.1). This theorem states that, if a program is verified under the precondition $\Diamond S$, then setting the maximum heap size to S allows this program to be safely executed.

5.6 Pointed-By-Heap Assertions

Our *pointed-by-heap* assertions are the "pointed-by" assertions of our earlier paper [Moine et al. 2023]. The longer name "pointed-by-heap" avoids confusion with our novel "pointed-by-thread" assertions (§5.7). To make this paper self-contained, we recall what form these assertions take, what they mean, and what purpose they serve.

A *pointed-by-heap* assertion for the location ℓ' keeps track of a multiset L of predecessors of ℓ' (§4.2.5). It takes the form $\ell' \hookleftarrow_q L$, where L is a signed multiset of locations and q is a possibly-null fraction, that is, a rational number in the closed interval $[0; 1]$.

Signed multisets. Signed multisets [Hailperin 1986], also known as *generalized sets* [Whitney 1933; Blizard 1990] or *hybrid sets* [Loeb 1992], are a generalization of multisets: they allow an element to have *negative* multiplicity. A signed multiset is a total function of elements to $\mathbb{Z}$. The disjoint union operation $\uplus$ is the pointwise addition of multiplicities. We write $+x$ for a positive occurrence of x and $-x$ for a negative occurrence of x. For example, $\{+x; +x\} \uplus \{-x\}$ is $\{+x\}$. We write $\mathrm{NoNegative}(L)$ when no element has negative multiplicity in L. Symmetrically, we write $\mathrm{NoPositive}(L)$ when no element has positive multiplicity in L.

$$\begin{array}{rcll}
(\ell \mathrel{\leftarrow\!\!\!\dashv}_{q_1} L_1 \;*\; \ell \mathrel{\leftarrow\!\!\!\dashv}_{q_2} L_2) & \mathrel{-\!\!*} & \ell \mathrel{\leftarrow\!\!\!\dashv}_{q_1+q_2} (L_1 \uplus L_2) & & \textsc{JoinPBHeap} \\
\ell \mathrel{\leftarrow\!\!\!\dashv}_{q_1+q_2} (L_1 \uplus L_2) & \mathrel{-\!\!*} & (\ell \mathrel{\leftarrow\!\!\!\dashv}_{q_1} L_1 \;*\; \ell \mathrel{\leftarrow\!\!\!\dashv}_{q_2} L_2) & \text{if } \begin{cases} q_1 = 0 \Rightarrow \mathsf{NoPositive}(L_1) \\ q_2 = 0 \Rightarrow \mathsf{NoPositive}(L_2) \end{cases} & \textsc{SplitPBHeap} \\
\ell \mathrel{\leftarrow\!\!\!\dashv}_{q} L & \mathrel{-\!\!*} & \ell \mathrel{\leftarrow\!\!\!\dashv}_{q} (L \uplus \{+\ell'\}) & \text{if } q > 0 & \textsc{CovPBHeap}
\end{array}$$

Fig. 13. Reasoning rules for the pointed-by-heap assertion

Possibly-Null Fractions. In traditional Separation Logics with fractional permissions [Boyland 2003; Bornat et al. 2005], a fraction is a rational number in the semi-open interval $(0, 1]$. If there exists a share that carries the fraction 1, then no other shares can separately exist. With *possibly-null fractions*, the fraction 0 is allowed, so a full pointed-by-heap assertion $\ell' \mathrel{\leftarrow\!\!\!\dashv}_1 L$ does *not* exclude the existence of a separate pointed-by-heap assertion with fraction zero, say $\ell' \mathrel{\leftarrow\!\!\!\dashv}_0 L'$.

Nevertheless, we enforce the following *null-fraction invariant*: in a pointed-by-heap assertion $\ell' \mathrel{\leftarrow\!\!\!\dashv}_q L$, *if the fraction q is* 0*, then no location can have positive multiplicity in* L; or, in short, $q = 0$ implies $\mathsf{NoPositive}(L)$.

Signed multisets and possibly-null fractions allow us to use the assertion $\ell' \mathrel{\leftarrow\!\!\!\dashv}_0 \{-\ell\}$ as *a permission to remove one occurrence of ℓ from the predecessors of ℓ'*. This lets us formulate the reasoning rule for store instructions (§6.2) in a simpler way than would otherwise be possible.

Over-Approximation of Live Predecessors. We say that a location ℓ is *dead* if it has been allocated and logically deallocated already (§5.9). We say that it is *live* if it has been allocated but not logically deallocated yet.

The true purpose of pointed-by-heap assertions is to keep track of *live* predecessors. A dead predecessor is irrelevant: increasing its multiplicity in a multiset of predecessors is sound; decreasing it is sound, too. As far as live predecessors are concerned, only over-approximation is permitted. Increasing the multiplicity of a live predecessor is sound; decreasing it is not.

In light of this, and in light of the null-fraction invariant, a *full* pointed-by-heap assertion $\ell' \mathrel{\leftarrow\!\!\!\dashv}_1 L$, where the fraction is 1, guarantees that the multiset L contains *all live predecessors* of the location ℓ'. In particular, the assertion $\ell' \mathrel{\leftarrow\!\!\!\dashv}_1 \emptyset$ guarantees that ℓ' has *no live predecessors*. Such full knowledge of the live predecessors is required by the logical deallocation rule (§6.1, §6.6).

Rules. Pointed-by-heap assertions obey the splitting, joining, and weakening rules in Figure 13. JoinPBHeap joins two pointed-by-heap assertions by adding the fractions q_1 and q_2 and by adding the signed multisets L_1 and L_2. In the reverse direction, SplitPBHeap splits a pointed-by-heap assertion. Its side condition ensures that the null-fraction invariant is preserved. CovPBHeap asserts that a pointed-by-heap assertion (whose fraction is nonzero) is covariant in its multiset: that is, over-approximating the multiset of predecessors is sound. It is a direct consequence of SplitPBHeap, instantiated with $q_2 \triangleq 0$ and $L_2 \triangleq \{-\ell'\}$. In the reverse direction, the rule CleanPBHeap, which is discussed later on (§5.9), allows removing a dead predecessor from a multiset of predecessors.

Life cycle. A full pointed-by-heap assertion for the location ℓ appears when this location is allocated. Fractional pointed-by-heap assertions are required, updated, and produced by store instructions. For example, consider a store instruction that updates the field $\ell[i]$ and overwrites the value ℓ'_1 with the value ℓ'_2. The reasoning rule for this instruction (§6.2) requires a pointed-by-heap assertion $\ell'_2 \mathrel{\leftarrow\!\!\!\dashv}_q \emptyset$, which it transforms into $\ell'_2 \mathrel{\leftarrow\!\!\!\dashv}_q \{+\ell\}$. Furthermore, the pointed-by-heap assertion $\ell'_1 \mathrel{\leftarrow\!\!\!\dashv}_0 \{-\ell\}$ is produced. A full pointed-by-heap assertion for the location ℓ is consumed when ℓ is logically deallocated.

$$\begin{array}{rcll} \ell \Mapsfrom_{p_1+p_2} (\Pi_1 \cup \Pi_2) & \equiv & (\ell \Mapsfrom_{p_1} \Pi_1 \quad * \quad \ell \Mapsfrom_{p_2} \Pi_2) & \textsc{FracPBThread} \\ \ell \Mapsfrom_p \Pi_1 & \mathrel{-\!\!*} & \ell \Mapsfrom_p (\Pi_1 \cup \Pi_2) & \textsc{CovPBThread} \\ \ulcorner \ell \notin V \urcorner \quad * \quad \ell \Mapsfrom_p \{\pi\} & {}^{\pi}\!\Rrightarrow^{V} & \ell \Mapsfrom_p \emptyset & \textsc{TrimPBThread} \end{array}$$

Fig. 14. Reasoning rules for the pointed-by-thread assertion

Notation. We define a generalized pointed-by-heap assertion $v \mapsfrom_q L$ whose first argument is a value, as opposed to a memory location. If v is a location ℓ', then this assertion is defined as $\ell' \mapsfrom_q L$. Otherwise, it is defined as $\ulcorner \text{True} \urcorner$. Furthermore, we write $v \overset{>0}{\mapsfrom}_q L$ for the assertion $\ulcorner q > 0 \urcorner * v \mapsfrom_q L$. This notation is used in the reasoning rule Store (§6.2), among other places.

5.7 Pointed-By-Thread Assertions

The pointed-by-heap assertions presented in the previous section record *which heap blocks* contain pointers to a location ℓ. This information is useful but is not sufficient for our purposes. The logic must also record *which threads* have access to ℓ, that is, in which threads ℓ is a root. For this purpose, we introduce two distinct yet cooperating mechanisms. The first mechanism, presented here, is the pointed-by-thread assertion. The second mechanism, presented next (§5.8), is the "*inside*" assertion. When the fact that ℓ is a root in thread π is recorded by a pointed-by-thread assertion, we say that ℓ is an *ordinary root* in thread π; when this fact is recorded by an "*inside*" assertion, we say that ℓ is a *temporary root* in thread π. The motivation for this distinction has been presented earlier (§3, §2.3).

A *pointed-by-thread* assertion takes the form $\ell \Mapsfrom_p \Pi$, where p is a fraction in the semi-open interval $(0; 1]$ and Π is a set of thread identifiers. These assertions intuitively generalize the *Stackable* assertions of our earlier paper [Moine et al. 2023] to a multi-threaded setting.

A *full* pointed-by-thread assertion $\ell \Mapsfrom_1 \Pi$, where the fraction is 1, guarantees that Π is the set of *all* threads in which ℓ is an ordinary root. Such full knowledge is required by the logical deallocation rule (§6.1, §6.6).

Rules. Figure 14 presents the splitting, joining, weakening, and trimming rules associated with the pointed-by-thread assertion. FracPBThread allows splitting and joining pointed-by-thread assertions. CovPBThread asserts that a pointed-by-thread assertion is covariant in the set Π: that is, over-approximating Π is sound. TrimPBThread allows *trimming* a pointed-by-thread assertion, that is, removing the thread identifier π from a pointed-by-thread assertion for the location ℓ, provided it is evident that ℓ is no longer a root in thread π. This rule is expressed as a custom ghost update: it transforms $\ell \Mapsfrom_p \{\pi\}$ into $\ell \Mapsfrom_p \emptyset$, provided ℓ is not a member of the set V, which denotes the set of roots of the thread π (recall §5.2). The condition $\ell \notin V$ means indeed that ℓ is not a root in thread π.

A curious reader may wonder whether and why TrimPBThread remains sound in combination with the Bind rule. Indeed, Bind lets the user focus on a subterm, therefore implies that the set V is a strict *subset* of the set of all roots of the current thread. This aspect is explained later on (§6.4).

Life cycle. A full pointed-by-thread assertion $\ell \Mapsfrom_1 \{\pi\}$ appears when a location ℓ is allocated by a thread π. A fractional pointed-by-thread assertion is ordinarily required and updated by load instructions: when a thread π obtains the location ℓ as the result of a load instruction, an assertion $\ell \Mapsfrom_p \emptyset$ is updated to $\ell \Mapsfrom_p \{\pi\}$. If the thread π is currently outside a protected section, such an update is mandatory. If the thread π is currently inside a protected section, then it can be avoided by recording ℓ as a temporary root (§6.3). Once ℓ is no longer a root in any thread, TrimPBThread can be used to obtain $\ell \Mapsfrom_1 \emptyset$, which is consumed by the logical deallocation of ℓ.

$$
\begin{array}{rcll}
\mathit{inside}\,\pi\,T \;*\; \mathit{outside}\,\pi & -\!\!* & \ulcorner\mathrm{False}\urcorner & \textsc{InsideNotOutside} \\
\mathit{inside}\,\pi\,T \;*\; \ell \Mapsfrom_p \{\pi\} & \Rrightarrow & \mathit{inside}\,\pi\,(T \cup \{\ell\}) \;*\; \ell \Mapsfrom_p \emptyset & \textsc{AddTemporary} \\
\mathit{inside}\,\pi\,T \;*\; \ell \Mapsfrom_p \emptyset & \Rrightarrow & \mathit{inside}\,\pi\,(T \setminus \{\ell\}) \;*\; \ell \Mapsfrom_p \{\pi\} & \textsc{RemTemporary} \\
\mathit{inside}\,\pi\,T & {}^{\pi}\Rrightarrow^{V} & \mathit{inside}\,\pi\,(T \cap V) & \textsc{TrimInside}
\end{array}
$$

Fig. 15. Reasoning rules for "*inside*" and "*outside*" assertions

Notation. We define a generalized pointed-by-thread assertion $v \Mapsfrom_p \Pi$, whose first argument is a value, as opposed to a memory location. If v is a location ℓ, then this assertion is defined as $\ell \Mapsfrom_p \Pi$. Otherwise, it is defined as $\ulcorner\mathrm{True}\urcorner$. Besides, we write an iterated conjunction of pointed-by-thread assertions under the form $M \Mapsfrom \Pi$, where M is a finite map of memory locations to fractions and Π is a set of thread identifiers. It is defined by $M \Mapsfrom \Pi \;\triangleq\; \mathop{\Huge *}_{(\ell,p)\in M} (\ell \Mapsfrom_p \Pi)$.

5.8 Inside and Outside Assertions

The assertion $\mathit{outside}\,\pi$ means that the thread π is currently outside a protected section. The assertion $\mathit{inside}\,\pi\,T$ means that thread π is currently inside a protected section and that the set of its temporary roots (§2.5) is T. The set T is a set of memory locations.

Rules. Figure 15 presents a number of reasoning rules related to "*inside*" and "*outside*" assertions. InsideNotOutside states that a thread cannot be both inside and outside a protected section. AddTemporary converts an ordinary root to a temporary root. The pointed-by-thread assertion $\ell \Mapsfrom_p \{\pi\}$ is transformed to $\ell \Mapsfrom_p \emptyset$; meanwhile, ℓ is added to the set of temporary roots carried by the "*inside*" assertion. In the reverse direction, RemTemporary converts a temporary root to an ordinary root. TrimInside trims the set of temporary roots by removing any locations that are no longer roots in the current thread. It is analogous to TrimPBThread.

Life cycle. The assertion $\mathit{outside}\,\pi$ appears when thread π is created and is consumed when this thread terminates. This will be visible in the statement of Theorem 7.1, which describes the creation and termination of the main thread, and in the reasoning rule for "fork" instructions (§6.2). The assertion $\mathit{outside}\,\pi$ is required and preserved by the instructions that must not appear inside a protected section, namely memory allocations, function calls, "fork" instructions, and polling points. Entering a protected section transforms $\mathit{outside}\,\pi$ into $\mathit{inside}\,\pi\,\emptyset$; exiting a protected section causes the reverse transformation.

5.9 Deallocation Witnesses

The persistent assertion $\dagger\,\ell$ is a *deallocation witness* for the location ℓ. This assertion guarantees that ℓ has been logically deallocated, that is, ℓ is dead.

The fact that ℓ is dead implies that ℓ cannot be reached from an ordinary root. However, this does not imply that ℓ is unreachable: indeed, it could still be reachable via a temporary root.

The assertion $\dagger\,\ell$ can be read as a permission to remove ℓ from the multiset of predecessors carried by a pointed-by-heap assertion. Indeed, the purpose of pointed-by-heap assertions is to keep track of live predecessors (§5.6).

A non-persistent deallocation witness $x \not\mapsto$ appears in Incorrectness Separation Logic [Raad et al. 2020]. Persistent deallocation witnesses appear in Madiot and Pottier's work [2022] and in our earlier paper [Moine et al. 2023]. These two papers do not have protected sections, therefore have no distinction between ordinary and temporary roots. There, a dead location is unreachable.

Rules. Figure 16 presents reasoning rules for deallocation witnesses. CleanPBHeap requires a deallocation witness for ℓ and produces $\ell' \mapsfrom_0 \{-\ell\}$, allowing ℓ to be removed from the predecessors

$$\begin{array}{rcll}
\dagger\,\ell & \Rrightarrow & \ell' \mapsfrom_0 \{-\ell\} & \textsc{CleanPBHeap} \\
\dagger\,\ell \;\ast\; \ell \mapsfrom^{>0}_q L & \Rrightarrow & \ulcorner\text{False}\urcorner & \textsc{DeadPBHeap} \\
\dagger\,\ell \;\ast\; \ell \Mapsfrom_p \Pi & \Rrightarrow & \ulcorner\text{False}\urcorner & \textsc{DeadPBThread} \\
\ulcorner \ell \in V \urcorner \;\ast\; \dagger\,\ell \;\ast\; \mathit{outside}\,\pi & {}^{\pi}\!\!\Rrightarrow^{V} & \ulcorner\text{False}\urcorner & \textsc{NoDanglingRootOut} \\
\ulcorner \ell \in (V \setminus T) \urcorner \;\ast\; \dagger\,\ell \;\ast\; \mathit{inside}\,\pi\,T & {}^{\pi}\!\!\Rrightarrow^{V} & \ulcorner\text{False}\urcorner & \textsc{NoDanglingRootIn} \\
\multicolumn{3}{c}{\dagger\ell \text{ is persistent}} & \textsc{DeadPersist}
\end{array}$$

Fig. 16. Reasoning rules for deallocation witnesses

of an arbitrary location ℓ'. DeadPBHeap and DeadPBThread reflect the fact that logical deallocation consumes full pointed-by-heap and pointed-by-thread assertions. Therefore, the assertions $\dagger\,\ell$ and $\ell \mapsfrom_q L$ cannot coexist, except in the special case where q is zero, and the assertions $\dagger\,\ell$ and $\ell \Mapsfrom_p \Pi$ cannot coexist. However, in contrast with our earlier work [Madiot and Pottier 2022; Moine et al. 2023], our deallocation witness *is* compatible with the points-to assertion. Indeed, our logical deallocation rule does not consume the points-to assertion. NoDanglingRootOut and NoDanglingRootIn both state that it is impossible for a dead location to be an ordinary root. A dead location can, however, be a temporary root: indeed, our logical deallocation rule allows deallocating a temporary root (§6.1).

5.10 Liveness-Based Cancellable Invariants

An Iris *invariant* [Jung et al. 2018, §2.2] is written in the form $\boxed{\Phi}$.[5] It is a persistent assertion, whose meaning is that the assertion Φ in the rectangular box holds at all times. The assertion Φ itself is usually not persistent. An invariant can be temporarily *accessed* so as to gain access to the assertion Φ.

A *cancellable invariant* [Jung et al. 2018, §7.1.3] is an invariant that comes with a teardown mechanism, allowing the user to recover ownership of the assertion Φ once the invariant is canceled. This is a one-shot mechanism: once a cancellable invariant is torn down, it cannot be restored. Naturally, accessing a cancellable invariant requires proving that this invariant has not been torn down already.

In IrisFit, a form of *liveness-based cancellable invariants* (LCIs, for short) naturally arises. An LCI is tied to a memory location ℓ, and remains in force as long as this location is live. When the location ℓ is logically deallocated, all LCIs associated with ℓ are implicitly torn down. Therefore, to access an LCI associated with the location ℓ, one must prove that this location is still live: that is, one must prove that $\dagger\,\ell$ implies $\ulcorner\text{False}\urcorner$. This can be done using any of the rules DeadPBHeap, DeadPBThread, NoDanglingRootOut, and NoDanglingRootIn in Figure 16. When the location ℓ is logically deallocated, the assertion Φ can be recovered at the same time. We have used LCIs to reason about closures (§8.4) and about Treiber's stack (§10.5).

The implementation of LCIs is simple. A liveness-based cancellable invariant tied to the location ℓ, whose content is the assertion Φ, is just $\boxed{\dagger\,\ell \vee \Phi}$, that is, a plain Iris invariant whose content is the disjunction $\dagger\,\ell \vee \Phi$. By proving that $\dagger\,\ell$ is contradictory, the user excludes the left-hand disjunct, therefore obtains access to Φ. In particular, when one is about to logically deallocate ℓ, the assertion $\ell \Mapsfrom \emptyset$ is at hand, so $\dagger\,\ell$ is excluded. One can therefore open the invariant, extract Φ, deallocate ℓ, and close the invariant by supplying $\dagger\,\ell$, keeping Φ. (This is a somewhat unusual variation on the "golden idol" technique [Kaiser et al. 2017], with the persistent assertion $\dagger\,\ell$ in the role of the "bag of sand".)

[5] Formally, an invariant also carries a *namespace*, a technicality that prevents the user from accessing the invariant twice and obtaining two copies of Φ at the same time. For simplicity, we hide namespaces in this paper.

6 PROGRAM LOGIC: REASONING RULES

In this section, we present the reasoning rules of IrisFit. Because most of our design is guided by the desire for a flexible logical deallocation rule, we begin with a presentation of this rule, in the simplified case where a single memory location is deallocated (§6.1). Then, we present the reasoning rules for terms (§6.2), devoting special attention to protected sections (§6.3) and to the Bind rule, whose form is non-standard (§6.4). The standard statement of the Bind rule can be recovered when the user enters a restricted mode where certain rules are disabled (§6.5). Finally, we present the general form of the logical deallocation rule, which can deallocate cycles (§6.6).

6.1 Logical Deallocation

As in the previous papers by Madiot and Pottier [2022] and Moine et al. [2023], a key aspect of IrisFit is to provide a *logical deallocation* rule. This rule produces space credits: by logically deallocating a memory block, the user recovers the space credits that were consumed when this block was allocated. It can be applied to a memory location ℓ as soon as one is able to prove that this memory location is eligible for collection *during the next garbage collection phase.*

As in the previous work cited above, *if ℓ is unreachable* then it can be logically deallocated. Furthermore, what is new in this paper, *if ℓ is reachable only via temporary roots* (that is, via roots that will disappear by the time all protected sections are exited), then it can also be logically deallocated.

This reasoning rule may seem surprising, as it involves a form of anticipation: it exploits the fact that ℓ will be eligible for collection *once all protected sections have been exited*, yet it produces space credits *immediately*, at the point where the rule is applied. Intuitively, this is safe because a space credit serves to justify an allocation and (by design of our operational semantics) a large allocation request blocks until all protected sections have been exited. Hence, by the time extra free space is needed, any location that has been logically deallocated is effectively unreachable.

In §6.6, we present the general form of the logical deallocation rule, which can deallocate multiple memory locations at once, even if they form a cycle. Here, we present FreeOne, a simplified rule that is also useful in practice and that deallocates a single location ℓ:

$$\mathit{sizeof}\ \ell\ n \;\ast\; \ell \mapsfrom_1 \emptyset \;\ast\; \ell \Mapsfrom_1 \emptyset \quad\Rrightarrow\quad \Diamond \mathit{size}(n) \;\ast\; \dagger\,\ell \qquad\qquad \text{FreeOne}$$

FreeOne is expressed as a ghost update. It consumes three assertions: the "*sizeof*" assertion $\mathit{sizeof}\ \ell\ n$, the pointed-by-heap assertion $\ell \mapsfrom_1 \emptyset$, and the pointed-by-thread assertion $\ell \Mapsfrom_1 \emptyset$. The assertion $\mathit{sizeof}\ \ell\ n$ indicates that the memory block at address ℓ has size n. The assertion $\ell \mapsfrom_1 \emptyset$ guarantees that ℓ has no predecessor in the heap, that is, no memory block contains the pointer ℓ. The assertion $\ell \Mapsfrom_1 \emptyset$ guarantees that ℓ is not an ordinary root of any thread: that is, if ℓ is a root at all in a thread π, then it must be a temporary root for this thread (§2.5, §5.8). Together, the last two assertions imply that ℓ will be eligible for collection in the next garbage collection phase.

On the right-hand side of the ghost update, FreeOne produces two assertions, namely the recovered space credits $\Diamond n$ and the deallocation witness $\dagger\,\ell$. As noted earlier (§5.9), the latter assertion is a permission to remove ℓ from the predecessor multisets of other locations. Thus, by iterated application of FreeOne, acyclic chains of unreachable blocks can be logically deallocated.

FreeOne can be applied to a reachable location if this location is a temporary root inside a protected section. Our logic thereby allows such a location to be read or written *post mortem*, after it has been logically deallocated. This is made possible by the fact that the points-to assertion survives logical deallocation. This pattern appears, for example, in the verification of Treiber's stack (§10.5).

Our logical deallocation rule differs from the one proposed by Moine et al. [2023]. Indeed, while their rule consumes a points-to assertion for the location ℓ, ours does not. The points-to assertion is

$$\textsc{IfTrue}\quad \frac{\{\Phi\}\ \pi\colon t_1\ \{\Psi\}}{\{\Phi\}\ \pi\colon \text{if true then } t_1 \text{ else } t_2\ \{\Psi\}} \qquad \textsc{IfFalse}\quad \frac{\{\Phi\}\ \pi\colon t_2\ \{\Psi\}}{\{\Phi\}\ \pi\colon \text{if false then } t_1 \text{ else } t_2\ \{\Psi\}} \qquad \textsc{LetVal}\quad \frac{\{\Phi\}\ \pi\colon [v/x]t\ \{\Psi\}}{\{\Phi\}\ \pi\colon \text{let } x = v \text{ in } t\ \{\Psi\}}$$

$$\textsc{Prim}\quad \frac{v_1 \odot v_2 \xrightarrow{\text{pure}} w}{\{\ulcorner\text{True}\urcorner\}\ \pi\colon v_1 \odot v_2\ \{\lambda v.\ \ulcorner v = w\urcorner\}} \qquad \textsc{CallPtr}\quad \frac{\begin{matrix} v = \mu_{\text{ptr}} f.\, \lambda\vec{x}.\, t \qquad |\vec{x}| = |\vec{w}| \\ \{\mathit{outside}\,\pi\ *\ \Phi\}\ \pi\colon [v/f][\vec{w}/\vec{x}]t\ \{\Psi\}\end{matrix}}{\{\mathit{outside}\,\pi\ *\ \Phi\}\ \pi\colon (v\ \vec{w})_{\text{ptr}}\ \{\Psi\}}$$

$$\textsc{Val}\quad \{\ulcorner\text{True}\urcorner\}\ \pi\colon v\ \{\lambda v'.\ \ulcorner v' = v\urcorner\}$$

$$\textsc{Poll}\quad \{\mathit{outside}\,\pi\}\ \pi\colon \text{poll}\ \{\lambda().\ \mathit{outside}\,\pi\}$$

$$\textsc{Alloc}\quad \frac{0 < n}{\left\{\begin{matrix}\Diamond n \\ \mathit{outside}\,\pi\end{matrix}\right\}\ \pi\colon \text{alloc } \mathit{size}(n)\ \left\{\lambda\ell.\ \begin{matrix}\ell \mapsto_1 ()^n \\ \ell \mapsfrom_1 \emptyset \\ \ell \Mapsfrom_1 \{\pi\} \\ \mathit{outside}\,\pi\end{matrix}\right\}}$$

$$\textsc{Load}\quad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix}\ell \mapsto_p \vec{w} \\ v \Mapsfrom_{p'} \emptyset\end{matrix}\right\}\ \pi\colon \ell[i]\ \left\{\lambda v'.\ \begin{matrix}\ulcorner v' = v\urcorner \\ \ell \mapsto_p \vec{w} \\ v \Mapsfrom_{p'} \{\pi\}\end{matrix}\right\}}$$

$$\textsc{Store}\quad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix}\ell \mapsto_1 \vec{w} \\ v' \mapsfrom_q^{>0} \emptyset\end{matrix}\right\}\ \pi\colon \ell[i] \leftarrow v'\ \left\{\lambda().\ \begin{matrix}\ell \mapsto_1 [i{:=}v']\vec{w} \\ v' \mapsfrom_q^{>0} \{+\ell\} \\ v \mapsfrom_0 \{-\ell\}\end{matrix}\right\}}$$

$$\textsc{CASSuccess}\quad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix}\ell \mapsto_1 \vec{w} \\ v' \mapsfrom_q^{>0} \emptyset\end{matrix}\right\}\ \pi\colon \text{CAS}\ \ell[i]\ v\ v'\ \left\{\lambda b.\ \begin{matrix}\ulcorner b = \text{true}\urcorner \\ \ell \mapsto_1 [i{:=}v']\vec{w} \\ v' \mapsfrom_q^{>0} \{+\ell\} \\ v \mapsfrom_0 \{-\ell\}\end{matrix}\right\}}$$

$$\textsc{CASFailure}\quad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) \neq v}{\left\{\ell \mapsto_p \vec{w}\right\}\ \pi\colon \text{CAS}\ \ell[i]\ v\ v'\ \left\{\lambda b.\ \begin{matrix}\ulcorner b = \text{false}\urcorner \\ \ell \mapsto_p \vec{w}\end{matrix}\right\}}$$

$$\textsc{Fork}\quad \frac{\begin{matrix} \mathit{dom}(M) = \mathit{locs}(t) \\ (\forall\pi'.\ \{\mathit{outside}\,\pi'\ *\ M \Mapsfrom \{\pi'\}\ *\ \Phi\}\ \pi'\colon t\ \{\lambda().\ \mathit{outside}\,\pi'\})\end{matrix}}{\{\mathit{outside}\,\pi\ *\ M \Mapsfrom \{\pi\}\ *\ \Phi\}\ \pi\colon \text{fork}\ t\ \{\lambda().\ \mathit{outside}\,\pi\}}$$

Fig. 17. Syntax-directed reasoning rules, without protected-section-specific rules and without Bind

not needed to guarantee that the location is unreachable, nor is it needed to prevent a location from being deallocated twice. The size of the deallocated block is obtained in this paper from the "*sizeof*" assertion, whereas in the previous paper this assertion did not exist, so the size was obtained from a points-to assertion.

6.2 Reasoning Rules for Terms

Figure 17 presents most of the syntax-directed reasoning rules of IrisFit, except for the rules that are specific to protected sections and the Bind rule, which are presented later on (§6.3, §6.4). In every rule, the thread identifier π represents the current thread, that is, the thread about which one is reasoning (§5.1).

IfTrue, IfFalse, LetVal, Prim and Val are standard rules.

CallPtr governs calls to (recursive, closed) functions, also known in this paper as code pointers. Its only unusual aspect is the presence of the assertion *outside* π, which ensures that the current thread is currently outside a protected section. The presence of this assertion forbids function calls inside protected sections.

Similarly, Poll forbids polling points inside a protected section. Outside of this aspect, a polling point is a no-operation.

Alloc exhibits three differences with the allocation rule of Separation Logic. First, it requires and consumes $\mathit{size}(n)$ space credits, so as to pay for the space occupied by the new block. Second, the presence of the assertion $\mathit{outside}\,\pi$ forbids allocation inside a protected section. Third, in addition to a points-to assertion for the new block, allocation produces pointed-by-heap and pointed-by-thread assertions. These assertions indicate that there is initially no pointer from the heap to the new block, and that this new block is a root for the current thread (and only for this thread).

As in standard Separation Logic, Load requires a (fractional) points-to assertion for the memory location ℓ that is accessed. Furthermore, it requires a pointed-by-thread assertion $v \Mapsfrom_p \emptyset$ for the value v that is read from memory. This assertion is updated to $v \Mapsfrom_p \{\pi\}$, reflecting the fact that the value v becomes a root for the current thread.

As in standard Separation Logic, Store requires a full points-to assertion $\ell \mapsto_1 \vec{v}$ and produces an updated assertion $\ell \mapsto_1 [i{:=}v']\vec{v}$. Furthermore, it performs bookkeeping of predecessor multisets, so as to reflect the fact that the value v that was stored in the field $\ell[i]$ is overwritten with the value v'. First, to reflect the *creation* of an edge from ℓ to the value v', an assertion of the form $v' \mapsfrom_q \emptyset$ is changed to $v' \mapsfrom_q \{+\ell\}$. Here, because ℓ has positive multiplicity in $\{+\ell\}$, the null-fraction invariant requires that q be positive; it cannot be 0. Second, to reflect the *deletion* of an edge from ℓ to the value v, the assertion $v \mapsfrom_0 \{-\ell\}$ appears in the postcondition. As explained earlier (§5.6), this assertion is a permission to remove one occurrence of ℓ from a multiset of predecessors of v.

CASSuccess is similar to Store, but returns the Boolean value true rather than the unit value. Because a failed CAS does not modify the heap or create a new root, CASFailure is standard.

Fork reasons about the operation of spawning a new thread whose code is the term t. This operation must take place outside a protected section. Its impact on roots is as follows. Suppose, for a moment, that fork t is the last instruction in the parent thread. Then, the locations that occur in the term t cease to be roots of the parent thread π and become roots of the child thread π'. The reasoning rule reflects this intuition by updating a group of pointed-by-thread assertions. The iterated pointed-by-thread assertion $M \Mapsfrom \{\pi\}$ is taken away from the parent thread, and the updated assertion $M \Mapsfrom \{\pi'\}$ is transmitted to the child thread. M is a map of locations to fractions, whose domain is the set $\mathit{locs}(t)$. This is a form of *trimming*, similar in effect to the rules TrimPBThread and TrimInside.

If fork t is *not* the last instruction in the parent thread, then the user must use the reasoning rules Bind and Fork in combination. The interaction between the Bind rule and the "trimming" rules is discussed later on (§6.4, §6.5).

Still looking at Fork, an arbitrary assertion Φ is transmitted from the parent thread to the child thread. The assertion $\mathit{outside}\,\pi'$ is made available in the child thread, reflecting the fact that a new thread initially runs outside a protected section. The child thread t must be verified with the nontrivial postcondition $\mathit{outside}\,\pi'$, thereby disallowing a thread to terminate while inside a protected section.

In our Coq formalization, the postconditions of many reasoning rules contain a *later credit* [Spies et al. 2022]. Later credits play a role in eliminating the "later" modality. They are orthogonal to the main concern of this paper, namely the analysis of space complexity, so we hide them in the presentation of our reasoning rules. We do explain how later credits are used in our case study of the async-finish library (§10.4).

$$\textsc{Enter}\qquad \{\mathit{outside}\,\pi\}\ \pi\colon \mathsf{enter}\ \{\lambda().\ \mathit{inside}\,\pi\,\emptyset\}$$

$$\textsc{Exit}\qquad \{\mathit{inside}\,\pi\,\emptyset\}\ \pi\colon \mathsf{exit}\ \{\lambda().\ \mathit{outside}\,\pi\}$$

$$\textsc{LoadInside}\qquad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix}\ell \mapsto_p \vec{w}\\ \mathit{inside}\,\pi\,T\end{matrix}\right\}\ \pi\colon \ell[i]\ \left\{\lambda v'.\ \begin{matrix}\ulcorner v' = v \urcorner\ *\ \ell \mapsto_p \vec{w}\\ \mathit{inside}\,\pi\,(\mathit{locs}(v) \cup T)\end{matrix}\right\}}$$

$$\textsc{StoreDead}\qquad \frac{0 \le i < |\vec{w}|}{\left\{\begin{matrix}\ell \mapsto_1 \vec{w}\\ \dagger\,\ell\end{matrix}\right\}\ \pi\colon \ell[i] \leftarrow v'\ \left\{\lambda().\ \ell \mapsto_1 [i{:=}v']\vec{w}\right\}}$$

$$\textsc{CASSuccessDead}\qquad \frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix}\ell \mapsto_1 \vec{w}\\ \dagger\,\ell\end{matrix}\right\}\ \pi\colon \mathsf{CAS}\ \ell[i]\ v\ v'\ \left\{\lambda b.\ \begin{matrix}\ulcorner b = \mathsf{true} \urcorner\\ \ell \mapsto_1 [i{:=}v']\vec{w}\end{matrix}\right\}}$$

Fig. 18. Reasoning rules: protected-section-specific rules

6.3 Reasoning about Protected Sections

Within a protected section, the reasoning rules presented in the previous section (§6.2) can still be used, except for CallPtr, Alloc, and Poll, which require the assertion *outside* π. In addition, a number of reasoning rules, shown in Figure 18, specifically concern protected sections.

Enter allows entering a protected section. This rule transforms the assertion *outside* π into the assertion *inside* $\pi\ \emptyset$, thereby witnessing that the current thread is now inside a protected section and has no temporary roots.

Conversely, Exit allows exiting a protected section. By consuming the assertion *inside* $\pi\ \emptyset$, this rule requires the user to prove that the current thread has no remaining temporary roots.

LoadInside allows reading a value v from a location ℓ in the heap. The locations that appear in the value v become temporary roots of the current thread: the assertion *inside* $\pi\ T$ is updated to *inside* $\pi\ (T \cup \mathit{locs}(v))$. In contrast with Load, no pointed-by-thread assertion is required or updated. In fact, the location ℓ or some locations in the set $\mathit{locs}(v)$ might be logically deallocated already.

StoreDead allows writing a logically deallocated block. The rule requires and updates a points-to assertion. A deallocation witness $\dagger\,\ell$ is also required. Compared with Store, no pointed-by-heap assertion is required or updated. Indeed, there is no need to do so. Pointed-by-heap assertions keep track of which blocks are reachable via ordinary roots; but, because the block at address ℓ is logically deallocated, it is not reachable via ordinary roots. This is reminiscent of CleanPBHeap.

Although StoreDead does not require an "*inside*" assertion, it can be used only inside a protected section. Indeed, the rule applies to a store instruction $\ell[i] \leftarrow v'$, where the address ℓ occurs. This means that ℓ is a root, yet ℓ is also logically deallocated. This is possible only if the current thread is currently inside a protected section. Indeed, outside a protected section, a logically deallocated location cannot be a root: the rule NoDanglingRootOut says so (§5.9).

CASSuccessDead is analogous to StoreDead. It concerns a successful CAS instruction on a logically deallocated location. Because a failed CAS does not write anything, the rule CASFailure can be applied to a logically deallocated location without change.

6.4 Reasoning under Evaluation Contexts

A proof in Separation Logic is traditionally carried out under an unknown context. That is, one reasons about a term t without knowing in what evaluation context K this term is placed. There are specific points in the proof where this unknown context grows and shrinks. As an archetypical example, consider the sequencing construct $\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$. To reason about this construct, one first

Bind

$$\frac{dom(M) = locs(K) \qquad \{\Phi\}\ \pi\colon t\ \{\Psi'\} \qquad \forall v.\ \{M \mathrel{\Leftarrow\!\!\!\mid} \{\pi\} \ \ast\ \Psi'\ v\}\ \pi\colon K[v]\ \{\Psi\}}{\{M \mathrel{\Leftarrow\!\!\!\mid} \{\pi\} \ \ast\ \Phi\}\ \pi\colon K[t]\ \{\Psi\}}$$

Fig. 19. Reasoning rules: the Bind rule

focuses on the term t_1, thereby temporarily forgetting the frame $\mathsf{let}\ x = \square\ \mathsf{in}\ t_2$, which is pushed onto the unknown context. After the verification of t_1 is completed, this focusing step is reversed: the frame $\mathsf{let}\ x = \square\ \mathsf{in}\ t_2$ is popped and one continues with the verification of t_2. These focusing and defocusing steps are described by the "Bind" rule [Jung et al. 2018, §6.2].

In our setting, however, a complication arises. An evaluation context contains memory locations. When one applies the Bind rule, so as to temporarily forget about this evaluation context, one must still somehow record that these locations are roots. We use pointed-by-thread assertions for this purpose.

Suppose we wish to decompose the sequence $\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$ into a subterm t_1 and an evaluation context $\mathsf{let}\ x = \square\ \mathsf{in}\ t_2$. For simplicity, let us further assume that $locs(t_2)$ is a singleton set $\{\ell\}$. This implies that, while t_1 is being executed, the location ℓ is a root. In this specific case, our Bind rule takes the following form:

Particular Case of Bind

$$\frac{locs(t_2) = \{\ell\} \qquad \{\Phi\}\ \pi\colon t_1\ \{\Psi'\} \qquad \forall v.\ \{\ \ell \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\} \ \ \ast\ \Psi'\ v\}\ \pi\colon [v/x]t_2\ \{\Psi\}}{\{\ \ell \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\} \ \ \ast\ \Phi\}\ \pi\colon \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\ \{\Psi\}}$$

What is unusual, compared with the standard Bind rule of Separation Logic, is that the fractional pointed-by-thread assertion $\ell \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\}$ is required in the beginning, taken away from the user while focusing on the term t_1, and given back to the user once she is done reasoning about t_1 and ready to reason about t_2. In other words, this assertion is *forcibly framed out* while reasoning about t_1.

The assertion $\ell \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\}$ records that ℓ is a root in thread π. By taking it away from the user and by giving it back once she is done reasoning about t_1, we ensure that the information that "ℓ is a root in thread π" is carried up to this point and cannot be prematurely destroyed.

What could go wrong if we did not do this? Then, the user would be allowed to keep the *full* pointed-by-thread assertion $\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \{\pi\}$ while reasoning about t_1. Technically, the user would do so by instantiating Φ with $\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \{\pi\}$ in the Bind rule. Then, the user would focus on establishing the first premise, $\{\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \{\pi\}\}\ \pi\colon t_1\ \{\Psi'\}$. Now suppose $\ell \notin locs(t_1)$, that is, ℓ does not occur in t_1. Then, the user could apply TrimPBThread to transform the assertion $\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \{\pi\}$ into $\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \emptyset$. Oops! The assertion $\ell \mathrel{\Leftarrow\!\!\!\mid}_1 \emptyset$ means that ℓ *is not* a root. Yet ℓ really *is* still a root, as it occurs in the evaluation context that has been abstracted away, namely $\mathsf{let}\ x = \square\ \mathsf{in}\ t_2$.

Besides TrimPBThread, two reasoning rules, namely Fork and TrimInside, involve a form of "trimming" of sets of thread identifiers. The soundness of these rules relies on the fact that Bind forcibly frames out fractional pointed-by-thread assertions.

The general form of our Bind rule, shown in Figure 19, extends this idea to an arbitrary evaluation context K, in which an arbitrary number of locations may occur. Then, for every location in $locs(K)$, a fractional pointed-by-thread assertion is forcibly framed out.

6.5 Locally Trading Trimming for a Simpler and More Powerful Bind Rule

Forcing pointed-by-thread assertions to be framed out at each application of Bind is cumbersome, and can be restrictive, as there are situations where no pointed-by-thread assertion is at hand. (An example appears later on in this section.) Fortunately, such forced framing is unnecessary if the

$$\textsc{SwitchMode}\quad \dfrac{\{\Phi\}\ \text{✖}/\pi\colon t\ \{\Psi\}}{\{\Phi\}\ m/\pi\colon t\ \{\Psi\}} \qquad\qquad \textsc{BindNoTrim}\quad \dfrac{\{\Phi\}\ \text{✖}/\pi\colon t\ \{\Psi'\} \qquad \forall v.\,\{\Psi'\,v\}\ m/\pi\colon K[v]\ \{\Psi\}}{\{\Phi\}\ m/\pi\colon K[t]\ \{\Psi\}}$$

Fig. 20. Reasoning rules: additional mode-specific rules

user promises not to exploit any of the trimming rules TrimPBThread, Fork and TrimInside. Thus, we introduce a mode that the user may choose to enter at any time, in which the trimming rules are disabled and, in exchange, a simpler, more powerful Bind rule is made available.

We parameterize IrisFit triples with a *mode* m, which is either the normal mode ✂ or the “no trim” mode ✖. Thus, in general, our triples have the form $\{\Phi\}\ m/\pi\colon t\ \{\Psi\}$, and our custom ghost update has the form $\Phi\ {}^{\pi}\!\!\Rrightarrow^{V}_{m}\ \Phi'$. All of the reasoning rules presented so far are polymorphic in the mode, except for the trimming rules TrimPBThread, Fork, and TrimInside, which are disabled in “no trim” mode. The public specification of a function is always stated in the normal mode. The “no trim” mode is intended for local use, inside the body of a function. It is an adaptation of Moine et al.’s “NoFree” mode [2023].

Figure 20 presents two new reasoning rules, SwitchMode and BindNoTrim, which allow entering “no trim” mode and taking advantage of it.

When read from bottom to top, SwitchMode lets the user locally enter “no trim” mode, whenever she so wishes, in a subproof. When read from to top to bottom, this rule asserts that if a triple holds in “no trim” mode then it also holds in normal mode. Indeed, every reasoning rule that is available in “no trim” mode is available in normal mode as well.

BindNoTrim is the standard Bind rule of Separation Logic, but imposes a switch to “no trim” mode ✖ in its left-hand premise. Thus, unlike our Bind rule, it does *not* force pointed-by-thread assertions to be framed out. Because of this, it must disable the trimming rules while the user reasons about the subterm t.

We remark that, inside a protected section, one can switch to “no trim” mode without loss of expressive power. Indeed, there, the trimming rules are never needed. Fork is forbidden inside protected sections; the effect of TrimPBThread can be simulated by AddTemporary; and all uses of TrimInside can be postponed until the protected section is about to be exited.

At a high level, BindNoTrim is needed for reasoning about code that, within a protected section, reads or writes in a location after it has been logically deallocated. Indeed, in this case, Bind can be too restrictive. To illustrate this case, consider the following code, where we assume that the location r is not accessible via the heap and is not known to any thread other than the current thread:

$$\mathsf{enter}\,;\ (\mathsf{let}\,x = t\,\mathsf{in}\,x + r[0])\,;\ \mathsf{exit}$$

Just after entering the protected section, the user may wish to logically deallocate r, in order to recover the corresponding space credits without waiting for the end of the protected section. In this case, just after entering the protected section, she would use AddTemporary to obtain a pointed-by-thread assertion $r \Lleftarrow \emptyset$, then use FreeOne to logically deallocate r, consuming this pointed-by-thread assertion. Thereafter, the user may wish to decompose the let construct. Yet, the Bind rule cannot be used, as it would require a (fractional) pointed-by-thread assertion for r, which no longer exists, because the fraction 1 was consumed by FreeOne. Fortunately, BindNoTrim is applicable.

$$\ulcorner \mathrm{True} \urcorner \mathrel{-\!\!*} \emptyset \;☁^{0}\; \emptyset \qquad \textsc{CloudEmpty}$$

$$\begin{array}{r} P \;☁^{n}\; D \;*\; \mathit{sizeof}\;\ell\;m \\ \ell \Mapsfrom_1 \emptyset \;*\; \ell \mapsfrom_1 L \;*\; \ulcorner \mathrm{NoNegative}(L) \urcorner \end{array} \mathrel{-\!\!*} (P \cup L) \;☁^{(n+m)}\; (D \cup \{\ell\}) \qquad \textsc{CloudAdd}$$

$$\ulcorner P \subseteq D \urcorner \;*\; P \;☁^{n}\; D \Rrightarrow \Diamond n \;*\; \mathop{\circledast}_{\ell \in D} \dagger \ell \qquad \textsc{CloudFree}$$

Fig. 21. Reasoning rules: logical deallocation

6.6 Logical Deallocation of Cycles

Figure 21 presents our rules for deallocating an unreachable heap *fragment*, as opposed to a single location. This fragment may contain an arbitrary number of heap blocks, which may point to each other in arbitrary ways. In particular, these pointers may form one or more cycles.

These rules make use of the "cloud" assertion $P \;☁^{n}\; D$, whose parameters P (for "predecessors") and D (for "domain") are sets of locations, and whose parameter n is a natural integer. This assertion means that the memory blocks at locations D have total size n, that the locations D are not roots in any thread, and that these locations can be reached only via the locations P. We refer to P also as the *entry points* of the cloud.

If $P \subseteq D$ holds, then the locations in the set D are reachable only via D itself. In other words, the set D is closed under predecessors. This means that the locations in the set D are in fact *unreachable*, and can safely be logically deallocated. This explains the side condition $P \subseteq D$ in the logical deallocation rule CloudFree. We do not require $P \subseteq D$ to hold at all times: while constructing large "cloud" assertions out of smaller "cloud" assertions, one must allow the sets P and D to be unrelated.

Figure 21 presents two cloud construction rules as well as the logical deallocation rule, which consumes a cloud.

Out of nothing, CloudEmpty creates an empty cloud $\emptyset \;☁^{0}\; \emptyset$.

CloudAdd adds the memory block at location ℓ to an existing cloud $P \;☁^{n}\; D$. This consumes the full pointed-by-thread assertion $\ell \Mapsfrom_1 \emptyset$, which guarantees that ℓ is not a root in any thread, and the full pointed-by-heap assertion $\ell \mapsfrom_1 L$, which guarantees that L contains all of the predecessors of the location ℓ in the heap. A "*sizeof*" assertion determines the size m of the memory block at address ℓ. CloudAdd produces an extended cloud, where L is added to the cloud's entry points, m is added to the cloud's size, and ℓ is added to the cloud's domain.

CloudFree logically deallocates a cloud that is closed under predecessors, that is, a cloud such that $P \subseteq D$ holds. The "cloud" assertion is consumed. In exchange for it, the rule produces n space credits, where n is the size of the cloud. Furthermore, it produces a deallocation witness for every location in the cloud.

The rule FreeOne that was presented earlier (§6.1) is easily derived from the rules in Figure 21.

7 SAFETY AND LIVENESS

In this section, we state a safety theorem and a liveness theorem about programs that have been verified using IrisFit.

The *safety* theorem (§7.1) guarantees that no thread crashes. More precisely, it states that if a thread is enabled (in the sense of §4.2.7), then this thread is not stuck: either it has reached a value or it can make a step.

The *liveness* theorem (§7.2) guarantees that no thread can be blocked forever. More precisely, under the assumption that there is a polling point in front of every function call, we prove that

$$\textsc{NotStuckVal}\ \frac{\theta(\pi) = (v, \mathsf{Out})}{\mathsf{NotStuck}\ (\theta, \sigma)\ \pi} \qquad \textsc{NotStuckStep}\ \frac{c \xrightarrow{\text{enabled action}}_{\pi} c'}{\mathsf{NotStuck}\ c\ \pi} \qquad \textsc{Safe}\ \frac{\forall \pi.\ \mathsf{Enabled}\ c\ \pi \implies \mathsf{NotStuck}\ c\ \pi}{\mathsf{Safe}\ c} \qquad \textsc{Always}\ \frac{\forall c'.\ c \xrightarrow{\text{main}}{}^{*}\ c' \implies P\ c'}{\mathsf{Always}\ P\ c}$$

Fig. 22. Predicates used in the statement of the safety theorem

every thread is eventually enabled. Furthermore, we prove that inserting a polling point in front of every function call preserves safety. Thus, after a source program without polling points has been verified with IrisFit, one can let a compiler automatically insert polling points, and obtain both safety and liveness for this instrumented program.

Our safety and liveness theorems both follow from a *core soundness* theorem (§7.3). This theorem spells out the guarantee that is offered by IrisFit when a LambdaFit program is executed under a simplified *oblivious semantics* that has neither garbage collection nor blocking instructions.

7.1 Safety

A concurrent Separation Logic typically comes with a safety guarantee, formulated in the form: "*no thread can crash*". A slightly more precise statement is: "*always, every thread is not stuck*". In other words, in every reachable configuration of the system, every thread either has terminated or is able to make a reduction step. A thread that has not reached a value and is unable to make a step is *stuck*: by convention, this is considered an undesirable situation, akin to a crash.

In our setting, however, this statement must be amended, because LambdaFit has blocking instructions. A blocking instruction is sometimes *disabled* (§4.2.7), therefore unable to make a step; yet, this situation is not considered a crash.

Our amended safety guarantee is qualified as follows: "always, every thread *that is enabled* is not stuck". A thread that is not enabled is considered blocked: this is a normal situation.

Figure 22 defines a few auxiliary predicates that appear in the statement of the safety theorem. The proposition NotStuck $c\ \pi$ means that, in the configuration c, the thread identified by π is not stuck. It is defined by two rules. NotStuckVal states that if a thread has reached a value and is outside a protected section, then it is not stuck. (Terminating inside a protected section is forbidden.) NotStuckStep states that if a thread is enabled and can take a step, then it is not stuck. The proposition Safe c, defined by the rule Safe, means that no enabled thread in the configuration c is stuck. The proposition Always $P\ c$, defined by the rule Always, means that every configuration that is reachable from the configuration c satisfies the predicate P.

The safety theorem (Theorem 7.1) can be read as follows. Suppose that the program t has been verified using IrisFit, with an arbitrary identifier π for the main thread, under the precondition $\Diamond S * \mathit{outside}\,\pi$ and the postcondition $\mathit{outside}\,\pi$. The precondition provides S space credits and guarantees that the main thread initially runs outside a protected section. The postcondition forbids termination inside a protected section. Then, the initial configuration $\mathit{init}(t)$ is *always safe*.

Theorem 7.1 (Safety). *Assume that, for every thread identifier π, the following triple holds:*

$$\{\Diamond S * \mathit{outside}\,\pi\}\ \pi\colon t\ \{\lambda\_.\ \mathit{outside}\,\pi\}$$

Then Always Safe $(\mathit{init}(t))$ holds.

The natural number S in this statement is the maximum heap size that appears as a parameter in the definition of the semantics of LambdaFit (§4.2.8). Although the soundness theorems mention S, the meaning of a triple is independent of S, therefore the reasoning rules are independent of S as

$$\textsc{HoldsNow}\ \frac{P\ c}{\mathsf{AfterAtMost}\ n\ P\ c} \qquad \textsc{HoldsAfter}\ \frac{\begin{array}{c} n > 0 \qquad (\exists c'.\ c \xrightarrow{\mathsf{main}} c') \\ (\forall c'.\ c \xrightarrow{\mathsf{main}} c' \implies \mathsf{AfterAtMost}\ (n-1)\ P\ c') \end{array}}{\mathsf{AfterAtMost}\ n\ P\ c} \qquad \textsc{Eventually}\ \frac{\mathsf{AfterAtMost}\ n\ P\ c}{\mathsf{Eventually}\ P\ c}$$

Fig. 23. Predicates used in the statement of the liveness theorem

well. Therefore, one can verify a program component without fixing the value of S. A concrete value of S must be chosen and fixed only when Theorem 7.1 is applied to a complete (closed) program.

7.2 Liveness

The safety theorem guarantees that no thread can crash, but allows a thread to become blocked. Therefore, a liveness guarantee is also desirable: one would like to be assured that *always, every thread is eventually enabled.* In other words, there is no execution scenario where certain threads remain blocked forever, in the sense that they never become enabled after some point.

In fact, we are able to offer a stronger guarantee: we prove that *always, eventually, every allocation fits.* In other words, in every execution scenario, infinitely often, the system reaches a point where no allocation request is blocked due to a lack of memory. This property is indeed stronger, because it captures the fact that, at a certain point, *every* thread is enabled at once. (By Lemma 4.1, the property *always, eventually, every allocation fits* implies that *always, eventually, all threads are enabled* at the same time; which, in turn, implies that *always, every thread is eventually enabled.*)

However, our liveness guarantee is subject to a condition: the program must contain *enough polling points.* To see why this is necessary, imagine a program where thread A is blocked on a large allocation request and thread B is running in an infinite loop, without allocating memory or encountering a polling point. Then, there exists a scenario where thread B runs forever, the GC is never invoked, and thread A never becomes unblocked. Thus, the desired liveness property does not hold. However, suppose that a polling point is inserted in the loop: thread B is not allowed to proceed past this polling point. Then, in every scenario, a garbage collection step eventually takes place, at which time both thread A and thread B become unblocked.

How can one tell whether a program has enough polling points? Or, in other words, where polling points must be inserted so that the program has enough polling points? We propose a simple approach, which is to *insert a polling point in front of every function call.*[6] This ensures that every thread must reach a polling point in a bounded number of steps. Up to an administrative side condition,[7] we prove that this polling point insertion strategy preserves safety and ensures liveness. We refer to this polling point insertion strategy as *addpp*. Thus, if t is a term, then $addpp(t)$ is the term obtained by inserting a polling point in front of every function call in the term t.

[6] LambdaFit does not have loops: instead, loops must be simulated via tail-recursive functions. Thus, inserting a polling point in front of every function call effectively implies inserting a polling point inside every loop as well. Incidentally, because function calls are forbidden inside protected sections, a polling point is never inserted into a protected section, satisfying our restriction that polling points in protected sections are forbidden. Our polling point insertion strategy is loosely inspired by the (undocumented) polling point insertion strategy of the OCaml compiler. The OCaml compiler inserts a polling point at the beginning of every function (except possibly small leaf functions), inside every loop, and views memory allocation instructions as polling points.

[7] Prior to inserting polling points, we require the program to be in administrative normal form (ANF). That is, in every function call, we require the function itself and the actual arguments to be variables or values, as opposed to arbitrary expressions. This guarantees that the polling point that is inserted in front of the function call is executed *after* the actual arguments have been computed and *just before* the function is invoked.

Figure 23 introduces a few auxiliary predicates that appear in the statement of the liveness theorem. The proposition AfterAtMost $n\ P\ c$ means that, out of the configuration c, *every* execution path reaches a configuration that satisfies P in at most n steps. The proposition AfterAtMost $n\ P\ c$ is inductively defined by the rules HoldsNow and HoldsAfter. HoldsAfter guarantees not only that the predicate continues to hold after any possible step, but also that there exists such a step. The proposition Eventually $P\ c$ means that in a bounded number of steps, out of the configuration c, *every* execution path reaches a configuration that satisfies P. It is defined by the rule Eventually, via an existential quantification over n. (The explicit depth bound n provides a stronger guarantee than just the plain inductive. Indeed, AfterAtMost is *infinitely branching* due to the non-determinism of allocation, and one cannot extract a depth bound from an infinitely branching inductive [Bertot and Castéran 2004].)

Our final theorem states that if the program t has been verified using IrisFit, under the exact same conditions as in Theorem 7.1, then the program $addpp(t)$, in which enough polling points have been inserted, is safe and live.

Theorem 7.2 (Combined Safety and Liveness after Polling Point Insertion). *Suppose that the term t is in administrative normal form. Assume that, for every thread identifier π, the following triple holds:*

$$\{\Diamond S * \mathit{outside}\,\pi\}\ \pi\colon t\ \{\lambda\_.\ \mathit{outside}\,\pi\}$$

Let t' stand for the term $addpp(t)$. Then, both of the following propositions hold:

(1) *Always Safe* $(init(t'))$
(2) *Always* $(\mathit{Eventually\,EveryAllocFits})$ $(init(t'))$.

This statement reflects how we envision the practical use of IrisFit. We expect the user to verify a program t in which polling points have not yet been inserted. Thus, the user need not know where polling points will be placed; in fact, the user need not be aware of polling points at all. As explained earlier, the uninstrumented verified program t enjoys safety but not liveness. Nevertheless, the theorem guarantees that, once enough polling points have been inserted, the program becomes safe and live. Although Theorem 7.2 fixes a specific polling point insertion strategy, namely *addpp*, we do in fact support other strategies. Our mechanization [Moine 2024] includes a more general liveness theorem that leaves up to the user the burden of proving that there is "enough" polling points—meaning that, if one thread ever requests space, then eventually, either the program crashes or this request is satisfied. We prove that this hypothesis holds true for *addpp*.

7.3 Core Soundness

A provocative yet fundamental remark is that IrisFit has nothing to do with garbage collection. Indeed, its deallocation rule is purely logical. More generally, its reasoning rules are independent of *when* garbage collection takes place, or *whether* it takes place at all. In reality, IrisFit is concerned with the *live heap space* of a program, that is, the sum of the sizes of the reachable blocks.

Our earlier results, namely Theorems 7.1 and 7.2, follow from a *core soundness* result, which is expressed with respect to the *oblivious semantics*, an alternative semantics in which no garbage collection takes place and no instructions are blocking (§2.2). This core soundness theorem states that IrisFit offers safety and maximum live heap space guarantees.

This oblivious semantics takes the form of an *oblivious reduction* relation $c \xrightarrow{\text{oblivious}} c'$, defined by the rule Oblivious in Figure 24. This relation simply allows one action by an arbitrary thread π. This action need not be enabled: in this semantics, no instructions are blocking. Garbage collection steps are not permitted: in this semantics, there is no need for garbage collection. This relation does *not* depend on a parameter S.

$$\textsc{Oblivious}\ \dfrac{c \xrightarrow{\text{action}}_{\pi} c'}{c \xrightarrow{\text{oblivious}} c'} \qquad \textsc{NotStuckObliviousVal}\ \dfrac{\theta(\pi) = (v, \mathsf{Out})}{\mathsf{NotStuckOblivious}\ (\theta, \sigma)\ \pi} \qquad \textsc{NotStuckObliviousStep}\ \dfrac{c \xrightarrow{\text{action}}_{\pi} c'}{\mathsf{NotStuckOblivious}\ c\ \pi}$$

Fig. 24. The oblivious reduction relation and associated predicates

The transitive closure of the oblivious reduction relation interleaves the actions of all threads in arbitrary ways.

In this setting, we must redefine what it means for a thread to be "not stuck". The proposition NotStuckOblivious $c\ \pi$, also defined in Figure 24, serves this purpose. A thread is not stuck if either it has reached a value outside a protected section, or it can make a step.

Let us write $\mathit{livespace}(R, \sigma)$ for the total size of the fragment of the store σ that is reachable from the roots R. Let us write $\mathit{livespace}(c)$ for the live heap space of the configuration c. It is defined by $\mathit{livespace}((\theta, \sigma)) = \mathit{livespace}(\mathit{locs}(\theta), \sigma)$.

Our core soundness theorem states that in every configuration that is reachable (with respect to the oblivious semantics), the following two properties hold. First, no thread is stuck. Furthermore, if every thread is currently outside a protected section, then the live heap size is at most S, where S is the number of space credits that was granted when the program was statically verified.

Theorem 7.3 (Core Soundness). *Assume that, for every thread identifier π, this triple holds:*

$$\{\Diamond S * \mathit{outside}\,\pi\}\ \pi\colon t\ \{\lambda\_.\ \mathit{outside}\,\pi\}$$

Then, for every configuration c such that $\mathit{init}(t) \xrightarrow{\text{oblivious}}{}^{*} c$,

(1) *for every identifier π of a thread in c, the property NotStuckOblivious $c\ \pi$ holds;*
(2) *AllOutside c implies $\mathit{livespace}(c) \leq S$.*

This statement may seem surprisingly weak, as it offers no guarantee about $\mathit{livespace}(c)$ at a time where AllOutside c does not hold, that is, at a time where at least one thread is inside a protected section. Moreover, this statement offers a safety guarantee; it does not offer any liveness guarantee. Nevertheless, this core soundness theorem is sufficiently strong to derive Theorems 7.1 and 7.2, which express the purpose of our logic in a different manner.

Our internal definition of IrisFit triples [Moine 2024] is relative to the oblivious semantics. The proof of Theorem 7.3, as well as the proofs of our reasoning rules, involve the oblivious semantics only. Thus, in many of our proofs, there is no need for us to reason about garbage collection or about the distinction between enabled and disabled reduction steps.

8 CLOSURES

As explained earlier (§2.6), LambdaFit does not have primitive closures. Instead, we define *closure construction* $\mu_{\mathsf{clo}} f.\,\lambda\vec{x}.\,t$ and *closure invocation* $(\ell\ \vec{u})_{\mathsf{clo}}$ as macros, which expand to sequences of primitive LambdaFit instructions. These macros implement *flat closures* [Appel 1992, Chapter 10]. That is, a closure is represented as a record whose fields store a code pointer (at offset 0) and a series of values (at offset 1 and beyond). The implementation of these macros (§8.2) is the same as in our earlier paper [Moine et al. 2023]. Our reasoning rules for closure construction, invocation, and deallocation are improved versions of the rules presented in our earlier paper [Moine et al. 2023]. The main improvement is that the assertions that describe closures are now *persistent*. From an end user's point of view, this makes closures much easier to work with. Internally, this is made possible by using *liveness-based cancellable invariants* (§5.9).

```
Closure construction:
$\mu_{\mathsf{clo}} f.\, \lambda\vec{x}.\, t \triangleq$
    let $f$ = alloc $(n+1)$ in
    $f[0] \leftarrow codeclo(f, \vec{x}, t)$;
    $f[i+1] \leftarrow y_i$;   # for each $i$ in $[0, n)$
    $f$
Closure invocation:
$(v\ \vec{w})_{\mathsf{clo}} \triangleq$
    $(v[0]\ (v :: \vec{w}))_{\mathsf{ptr}}$
```

```
Closure code pointer:
$codeclo(f, \vec{x}, t) \triangleq$
    $\mu_{\mathsf{ptr}} \_.\, \lambda(f :: \vec{x})$.
        let $y_i = f[i+1]$ in   # for each $i$ in $[0, n)$
        $t$

Side condition:
$fvclo(f, \vec{x}, t) = [y_0; \ldots; y_{n-1}]$
```

Fig. 25. Closures: macros for closure construction and invocation

Our reasoning rules for closures are abstract and do not reveal *how* closures are implemented. They reveal only how much space a closure occupies and which pointers it keeps live. A user can apply these rules without knowing how closures are internally represented.

Our construction of the reasoning rules for closures is in two layers. First, we introduce a low-level assertion $Closure\, E\, f\, \vec{x}\, t\, \ell$, which asserts that, at location ℓ in the heap, one finds a closure that behaves like the function $\mu f.\lambda\vec{x}.\, t$ under the environment E. Crucially, in this assertion, the term $\mu f.\lambda\vec{x}.\, t$ *can* have free variables, whose values are given by E. This assertion does not reveal how a closure is represented in memory, but does reveal its code. We give an overview of this low-level API (§8.3), then describe some details of its implementation (§8.4). Second, we define a high-level assertion $Spec\, n\, E\, P\, \ell$, which describes the behavior of a closure in a more abstract way. It asserts that, at location ℓ, one finds a closure that corresponds to a n-ary function, whose behavior is described by the predicate P, and whose environment is E. The exact type and meaning of P are explained later on; roughly speaking, it is a Hoare triple. Although the environment E does not participate in the description of the behavior of the closure, it remains needed in order to reason about the pointers that it contains and about the size of the closure block. We give an overview of this high-level API (§8.5), then describe its implementation (§8.6). Only the high-level layer is exposed to the end user; the low-level layer remains internal.

8.1 Environments

We write $fvclo(f, \vec{x}, t)$ for a list of the free variables of the function $\mu f.\lambda\vec{x}.\, t$, that is, for a list of the variables in the set $fv(t) \setminus \{f, \vec{x}\}$. The order in which the variables occur in this list does not matter, but is fixed: this is reflected in the fact that *fvclo* is a function of f, $\vec{x}$, and t.

An environment E is a list of pairs (v, q) of a value v and a nonzero fraction q. This fraction is used in a pointed-by-heap assertion, as follows: we write $E \hookleftarrow L$ for the conjunction $\mathop{\Large *}_{(v,q)\in E}\, v \hookleftarrow_q L$. The assertion $E \hookleftarrow L$ can be understood as a collective fractional pointed-by-heap assertion that covers every memory location that occurs in the environment E.

The length and order of the list E are intended to match the length and order of the list $fvclo(f, \vec{x}, t)$. An environment E is not a runtime object: it is a mathematical object that we use as a parameter of the predicates *Closure* and *Spec*.

8.2 Closure Implementation

The definitions of the closure macros $\mu_{\mathsf{clo}} f.\, \lambda\vec{x}.\, t$ and of $(\ell\ \vec{v})_{\mathsf{clo}}$ appear in Figure 25. Both macros generate LambdaFit syntax: that is, the result of their expansion is a LambdaFit expression. We write $t_1 \,;\, t_2$ is as sugar for let $x = t_1$ in t_2 where $x \notin fv(t_2)$.

The code produced by the macro $\mu_{\mathsf{clo}} f.\, \lambda\vec{x}.\, t$ allocates a block of size $n+1$, stores a code pointer in the first field, stores the values currently bound to the variables $y_0, \ldots, y_{n-1}$ in the remaining

MkClo

$$\frac{\vec{y} = \mathit{fvclo}(f, \vec{x}, t) \qquad E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad |\vec{v}| = |\vec{y}| \qquad f \notin \vec{x}}{\left\{\begin{matrix}\Diamond(\mathit{size}(1 + |E|)) \;\ast\; \mathit{outside}\,\pi \\ E \mapsfrom \emptyset\end{matrix}\right\} \;\pi\colon [\vec{v}/\vec{y}]\;(\mu_{\mathrm{clo}} f.\,\lambda\vec{x}.\,t)\; \left\{\lambda\ell.\begin{matrix}\mathit{outside}\,\pi \;\ast\; \mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell \\ \ell \Mapsfrom \{\pi\} \;\ast\; \ell \mapsfrom \emptyset\end{matrix}\right\}}$$

CallClo

$$\frac{\begin{matrix}\vec{y} = \mathit{fvclo}(f, \vec{x}, t) \qquad E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad |\vec{x}| = |\vec{w}| \\ \mathit{locs}(\vec{v}) = \mathit{dom}(M) \qquad \{\mathit{outside}\,\pi \;\ast\; M \Mapsfrom \{\pi\} \;\ast\; \Phi\}\;\pi\colon [\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t\;\{\Psi\}\end{matrix}}{\{\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell \;\ast\; \mathit{outside}\,\pi \;\ast\; M \Mapsfrom \{\pi\} \;\ast\; \Phi\}\;\pi\colon (\ell\;\vec{w})_{\mathrm{clo}}\;\{\Psi\}}$$

$$\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell \;\ast\; \ell \mapsfrom \emptyset \;\ast\; \ell \Mapsfrom \emptyset \quad\Rrightarrow\quad \Diamond(\mathit{size}(1 + |E|)) \;\ast\; \dagger\ell \;\ast\; E \mapsfrom \emptyset \qquad \textsc{CloFree}$$

$$\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell \text{ is persistent} \qquad \textsc{CloPersist}$$

Fig. 26. Closures: low-level API

fields, and returns the address of this block. The variables $y_0, \ldots, y_{n-1}$ are the free variables of the function $\mu f.\lambda\vec{x}.\,t$, that is, $\mathit{fvclo}(f, \vec{x}, t)$.

The code pointer is produced by the auxiliary macro $\mathit{codeclo}(f, \vec{x}, t)$. It is a closed function whose parameters are f (the closure itself) followed with $\vec{x}$. This function loads the values stored in the closure and binds them to the variables $y_0, \ldots, y_{n-1}$ before executing the body t.

The code produced by the closure invocation macro $(v\;\vec{v})_{\mathrm{clo}}$ first fetches the code pointer that is stored in the first field of the closure, then invokes this code pointer, passing it the closure v itself as well as the actual arguments $\vec{v}$.

8.3 Low-Level Closure API

Our low-level reasoning rules for closures, shown in Figure 26, involve the predicate *Closure*, which describes the layout of a closure in memory. Its definition is given in the next section (§8.4).

The rule MkClo specifies a closure construction operation. The term, written $[\vec{v}/\vec{y}]\;\mu_{\mathrm{clo}} f.\,\lambda\vec{x}.\,t$, is the application of the substitution $[\vec{v}/\vec{y}]$ to the closure construction macro $\mu_{\mathrm{clo}} f.\,\lambda\vec{x}.\,t$. In this substitution, the variables $\vec{y}$ are the free variables of the function $\mu f.\lambda\vec{x}.\,t$. The reason why we must be prepared to reason about a term of this form is that the premise of LetVal gives rise to substitutions which (after being propagated down) become blocked in front of the *opaque* macro $\mu_{\mathrm{clo}} f.\,\lambda\vec{x}.\,t$. The values $\vec{v}$ that appear in this substitution are the values "captured" by the closure, that is, the values that are stored in the closure when it is constructed.

In the second premise of MkClo, an environment E is built by pairing up the values $\vec{v}$ with nonzero fractions $\vec{q}$. Then, according to the precondition in MkClo, closure construction consumes $E \mapsfrom \emptyset$. In other words, for each memory location that occurs in E, it consumes a fractional pointed-by-heap assertion. This records the fact that there exists a pointer from the closure to each such memory location.

According to the precondition in MkClo, closure construction consumes $\mathit{size}(1 + |E|)$ space credits, reflecting the space needed to store a code pointer and the values $\vec{v}$ in a flat closure.

Because closure construction involves an allocation, MkClo requires the thread π to be outside a protected section.

According to the postcondition in MkClo, closure construction produces a memory location ℓ. Pointed-by-heap and pointed-by-thread assertions for this memory location are produced, indicating that this memory location is fresh. Furthermore, the assertion $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$, which guarantees that there is a well-formed closure at address ℓ, is also produced. In this paper, in contrast with

$$\begin{aligned} \mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell \quad &\triangleq \quad \ulcorner f \notin \vec{x} \;\wedge\; |E| = |\mathit{fvclo}(f, \vec{x}, t)| \urcorner \;\ast \\ &\qquad \ell \mapsto_{\square} (\mathit{codeclo}(f, \vec{x}, t) :: \mathit{map}\, \mathit{fst}\, E) \;\ast \\ &\qquad \boxed{\dagger\ell \;\vee\; E \mapsfrom \{+\ell\}} \end{aligned}$$

Fig. 27. Definition of the predicate *Closure*

our earlier work [Moine et al. 2023], this assertion is *persistent* [Jung et al. 2018, §2.3]. This means that the knowledge that there is a closure at address ℓ can be shared without any restriction. The pointed-by-heap and pointed-by-thread assertions $\ell \Mapsfrom \{\pi\} \ast \ell \mapsfrom \emptyset$ are *not* persistent. Indeed, these assertions allow deallocating the closure, and our program logic ensures that every object is deallocated at most once.

The rule CallClo closely resembles the rule CallPtr for primitive function calls (Figure 17). One difference is that CallClo requires the assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$, which describes the closure. Another difference is that, whereas a primitive function $\mu_{\mathrm{ptr}} f.\, \lambda\vec{x}.\, t$ must be closed, a general function can have a nonempty list of free variables $\vec{y}$, an alias for $\mathit{fvclo}(f, \vec{x}, t)$. In the last premise of CallClo, which requires reasoning about the function's body, the variables $\vec{y}$ are replaced with the values $\vec{v}$ captured at closure construction time, which are recorded in the environment E.

The precondition of CallClo requires a pointed-by-thread assertion $M \Mapsfrom \{\pi\}$, where the domain of the map M includes all of the locations that appear in $\vec{v}$, that is, all of the locations that appear in the closure's environment. This assertion is not consumed: it appears again in the precondition of the triple that forms the last premise of CallClo. In other words, it is transmitted from the caller to the callee. The presence of this assertion is imposed to us by the fact that, when the closure is invoked, these values are read from memory: the load instructions that appear in the definition of $\mathit{codeclo}(f, \vec{x}, t)$ in Figure 25 require pointed-by-thread assertions for the values that are read. If desired, the pointed-by-thread assertion $M \Mapsfrom \{\pi\}$ can be transmitted back from the callee to the caller via a suitable instantiation of the postcondition Ψ. Alternatively, it may be consumed by the callee to justify a logical deallocation operation.

Together, the rules MkClo and CallClo express the correctness of our closure construction and invocation macros. They guarantee that a closure at address ℓ constructed by $[\vec{v}/\vec{y}]\ \mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t$, when invoked with actual arguments $\vec{w}$, behaves like the term $[\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t$. This is the operational behavior that is expected of a closure.

CloFree logically deallocates a closure. It resembles FreeOne, but, instead of a "*sizeof*" assertion, requires the abstract assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$. Like FreeOne, it produces space credits and a deallocation witness for the closure. Furthermore, CloFree lets the user recover the pointed-by-heap assertion $E \mapsfrom \emptyset$, thereby undoing the effect of MkClo.

8.4 Low-Level Closure API: Implementation Details

Figure 27 presents the internal definition of the assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$. It records two pure facts: the name f is disjoint from the parameters $\vec{x}$ and the length of the environment E matches the number of free variables of the closure.

Then, a points-to assertion states that the location ℓ points to a block of size $1 + |E|$, whose first field contains the code of the closure, $\mathit{codeclo}(f, \vec{x}, t)$, and whose remaining fields contain the values recorded in the environment E. Because this points-to assertion carries a *discarded fraction* $\square$ [Vindum and Birkedal 2021], it is a *persistent points-to* assertion. This reflects the fact that the closure is immutable.

The last component in this definition is a liveness-based cancellable invariant (§5.10): a persistent assertion that we can tear down and regain full ownership when we deallocate ℓ.

$$\textsc{MkSpec}\quad\frac{\vec{y} = \mathit{fvclo}(f,\vec{x},t) \qquad E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad |\vec{v}| = |\vec{y}| \qquad f \notin \vec{x} \qquad n = |\vec{x}| \\ \forall \vec{w}.\;\; \Box\big(\mathit{Spec}\,n\,E\,P\,\ell \mathrel{-\!\!*} P\,\ell\,\vec{w}\,([\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t)\big)}{\left\{\begin{matrix}\Diamond(\mathit{size}(1+|E|)) \;\;*\;\; \mathit{outside}\,\pi \\ E \mapsfrom \emptyset\end{matrix}\right\}\; \pi\colon [\vec{v}/\vec{y}]\;(\mu_{\mathrm{clo}} f.\,\lambda\vec{x}.\,t)\; \left\{\lambda\ell.\begin{matrix}\mathit{outside}\,\pi \;\;*\;\; \mathit{Spec}\,n\,E\,P\,\ell \\ \ell \Mapsfrom \{\pi\} \;\;*\;\; \ell \mapsfrom \emptyset\end{matrix}\right\}}$$

$$\textsc{CallSpec}\quad\frac{E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad \mathit{dom}(M) = \mathit{locs}(\vec{v}) \qquad |\vec{w}| = n \\ (\,\forall u.\; P\,\ell\,\vec{w}\,u \mathrel{-\!\!*} \{\mathit{outside}\,\pi \;\;*\;\; M \Mapsfrom \{\pi\} \;\;*\;\; \Phi\}\; \pi\colon u\; \{\Psi\}\,)}{\{\mathit{Spec}\,n\,E\,P\,\ell \;\;*\;\; \mathit{outside}\,\pi \;\;*\;\; M \Mapsfrom \{\pi\} \;\;*\;\; \Phi\}\; \pi\colon (\ell\;\vec{w})_{\mathrm{clo}}\; \{\Psi\}}$$

$$\Box\,\big(\forall \vec{w}\,t.\; P_1\,\ell\,\vec{w}\,t \mathrel{-\!\!*} P_2\,\ell\,\vec{w}\,t\big) \;\;*\;\; \mathit{Spec}\,n\,E\,P_1\,\ell \quad \mathrel{-\!\!*} \quad \mathit{Spec}\,n\,E\,P_2\,\ell \qquad \textsc{SpecWeak}$$

$$\mathit{Spec}\,n\,E\,P\,\ell \;\;*\;\; \ell \mapsfrom \emptyset \;\;*\;\; \ell \Mapsfrom \emptyset \quad \Rrightarrow \quad \Diamond(\mathit{size}(1+|E|)) \;\;*\;\; \dagger\,\ell \;\;*\;\; E \mapsfrom \emptyset \qquad \textsc{SpecFree}$$

$$\mathit{Spec}\,n\,E\,P\,\ell \text{ is persistent} \qquad \textsc{SpecPersist}$$

Fig. 28. Closures: high-level API

Because every assertion involved in its definition is persistent, the assertion $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ is itself persistent.

The liveness-based cancellable invariant contains the pointed-by-heap assertion $E \mapsfrom \{+\ell\}$, which means that every memory location in E is pointed to by the closure. In the proof of the reasoning rule CloFree, we tear down the liveness-based cancellable invariant, and gain back the assertion $E \mapsfrom \{+\ell\}$. Because ℓ is now dead, we use the CleanPBHeap rule to change $E \mapsfrom \{+\ell\}$ into $E \mapsfrom \emptyset$. This explains how, in the proof of CloFree, we are able to produce the assertion $E \mapsfrom \emptyset$.

8.5 High-Level Closure API

The user of a program logic is ultimately interested in the specification of a function, not in the details of its implementation. Yet, the predicate $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ reveals the code of the closure. As a result, a user naturally wishes to hide this information via an existential quantification over this code. This pattern is common enough and technical enough that we offer a higher-level API where this existential quantification is built in. To this end, we introduce the assertion $\mathit{Spec}\,n\,E\,P\,\ell$ (defined further on in §8.6), where n is the arity of the function, E is the environment of the closure, P describes the behavior of the closure, and ℓ is the location of the closure in memory.

Like the *Closure* predicate (§8.3, §8.4), and unlike the *Spec* predicate presented in our previous paper [Moine et al. 2023], the predicate *Spec* is persistent. This enables a better separation of concerns between the persistent assertion $\mathit{Spec}\,n\,E\,P\,\ell$, which views the closure as an eternal service provider, and the affine assertion $\ell \Mapsfrom \{\pi\} * \ell \mapsfrom \emptyset$, which views it as an object in memory, allowing it to participate in the object graph and (someday) to be logically deallocated.

Figure 28 presents the reasoning rules associated with the *Spec* predicate. Let us first examine the rule CallSpec. In many ways, this rule is the same as the low-level rule CallClo. The main difference is that, to prove that the call $(\ell\;\vec{w})_{\mathrm{clo}}$ admits the postcondition Ψ, the user must check that the entailment $\forall u.\; P\,\ell\,\vec{w}\,u \mathrel{-\!\!*} \{\mathit{outside}\,\pi * M \Mapsfrom \{\pi\} * \Phi\}\; \pi\colon u\; \{\Psi\}$ holds. Intuitively, u denotes the instantiated function body that was visible in CallClo; however, this function body is now abstracted away by the universal quantification over u. The predicate P represents the specification of the function, and is typically instantiated with a triple. For example, in the specification of a closure of arity 1 whose effect is to increment a reference r that it receives as an argument, the predicate P takes the form: $\lambda\ell\,\vec{w}\,u.\; \forall\,r\,n.\; \ulcorner\vec{w} = [r]\urcorner \mathrel{-\!\!*} \{r \mapsto [n]\}\; \pi\colon u\; \{\lambda().\; r \mapsto [n+1]\}$. In short,

$$
\begin{array}{l}
\mathit{Spec}\, n\, E\, P\, \ell \;\triangleq \\
\quad \exists\, f\, \vec{x}\, t\, P'. \\
\quad\quad \ulcorner |\vec{x}| = n \urcorner \;\;\ast\;\; \mathit{Closure}\, E\, f\, \vec{x}\, t \;\;\ast \\
\quad\quad \text{let } \vec{v} = \mathit{map}\, \mathit{fst}\, E \text{ in} \\
\quad\quad \text{let } \vec{y} = \mathit{fvclo}(f, \vec{x}, t) \text{ in} \\
\quad\quad \text{let } \mathit{body}\, \vec{w} = [\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t \text{ in} \\
\quad\quad \triangleright \square (\forall \vec{w}.\, \mathit{Spec}\, n\, E\, P'\, \ell \;\;-\!\!\ast\;\; P'\, \ell\, \vec{w}\, (\mathit{body}\, \vec{w})) \;\;\ast \\
\quad\quad \triangleright \square (\forall \vec{w}\, u.\, P'\, \ell\, \vec{w}\, u \;\;-\!\!\ast\;\; P\, \ell\, \vec{w}\, u)
\end{array}
$$

Fig. 29. Definition of the predicate *Spec*

the user must prove an entailment stating that the specification needed by the caller follows from the specification P.

Let us now consider the rule MkSpec. It is again quite similar to the low-level rule MkClo. The premise on the second line ensures that P is a valid description of the behavior of the function body, whose concrete form $[\vec{v}/\vec{y}][\vec{w}/\vec{x}]t$ is visible. In comparison with the low-level API (§8.3), the work of reasoning about the function body is shifted from the closure invocation site to the closure construction site. Moreover, while establishing $P\, \ell\, \vec{w}\, ([\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t)$, the user is allowed to assume $\mathit{Spec}\, n\, E\, P\, \ell$: this allows verifying recursive calls.

The rule SpecWeak is a consequence rule: it allows weakening the assertion $\mathit{Spec}\, n\, E\, P_1\, \ell$ into $\mathit{Spec}\, n\, E\, P_2\, \ell$, under the hypothesis that P_1 is stronger than P_2.

The rule SpecFree is similar to the rule CloFree.

8.6 High-Level Closure API: Implementation Details

Figure 29 presents the definition of the assertion $\mathit{Spec}\, n\, E\, P\, \ell$. This is a guarded recursive definition: *Spec* appears (under a "later" modality) in its own definition. The definition is existentially quantified over the code of the closure, represented by f, $\vec{x}$, and t. It is also existentially quantified over a predicate P' that is required to be stronger than P. This lets us establish SpecWeak.

9 TRIPLES WITH SOUVENIR

In this section, we introduce *triples with souvenir*, a syntactic sugar that allows for simpler reasoning rules—in particular, a simpler Bind rule—while reasoning about code that lies outside a protected section. We first present the reasoning rules of triples with souvenir (§9.1), then cover how they are defined (§9.2).

9.1 Those Who Cannot Remember the Past Are Condemned to Repeat It

IrisFit, as presented until this point, can be cumbersome to use, for two unrelated reasons.

One reason is that the user must give up pointed-by-thread assertions at each application of Bind, even in the common case where such a fraction has been framed already at a previous application of Bind, which encloses the current application. This obligation to split off and give up pointed-by-thread assertions becomes especially heavy when a variable x denotes a location and has a long *live range*, that is, when this location remains a root throughout a long sequence of instructions. In such a situation, at each point in the sequence, the user is required to split off and give up a fractional pointed-by-thread assertion for x.[8]

[8] The problem is partly mitigated by the "no trim" mode ✖ (§6.5). However, this mode is designed for very local use, and cannot be exploited if trimming is needed.

BindWithSouvenir

$$\frac{dom(M) = locs(K) \setminus R \qquad [R \cup locs(K)]\,\{\Phi\}\,\pi\colon t\,\{\Psi'\} \qquad \forall v.\ [R]\,\{M \Leftarrow\!\!| \{\pi\} \ast \Psi'\,v\}\,\pi\colon K[v]\,\{\Psi\}}{[R]\,\{M \Leftarrow\!\!| \{\pi\} \ast \Phi\}\,\pi\colon K[t]\,\{\Psi\}}$$

AddSouvenir

$$\frac{[\{\ell\} \cup R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}{[R]\,\{\ell \Leftarrow\!\!|_p \{\pi\} \ast \Phi\}\,\pi\colon t\,\{\lambda v.\,\ell \Leftarrow\!\!|_p \{\pi\} \ast \Psi\,v\}}$$

ForgetSouvenir

$$\frac{R' \subseteq R \qquad [R']\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}{[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}$$

Fig. 30. Key reasoning rules for triples with souvenir

A second reason is that, typically, the large majority of instructions are placed outside protected sections. Yet, the user must provide the assertion *outside* π at each application of the *outside rules* Alloc, CallPtr, Fork, Poll, MkSpec, and CallSpec. This is not difficult, but the presence of this assertion creates visual clutter in pre- and postconditions.

To alleviate both problems at once, we follow Moine et al. [2023] and introduce *triples with souvenir.* A triple with souvenir takes the form $[R]\,\{\Phi\}\,\pi : t\,\{\Psi\}$, where R is a set of locations for which the user has already given up a pointed-by-thread assertion. Recording this *souvenir* (or remembrance) relieves the user from the obligation of giving up another pointed-by-thread assertion at future applications of the Bind rule. Furthermore, a triple with souvenir implicitly carries an *outside* π assertion: this allows for more concise statements of the outside rules.

For each reasoning rule in Figure 17, we provide a new rule (not shown) that operates on triples with souvenir and that is polymorphic in R. This is done simply by inserting $[R]$ in front every triple that appears in the rule. We do not provide new reasoning rules for protected sections, as triples with souvenir are applicable only outside protected sections.

The new reasoning rules that make use of souvenirs appear in Figure 30. BindWithSouvenir is what we aimed for: it is our motivation for introducing triples with souvenir. It closely resembles Bind, but does not require the user to give up pointed-by-thread assertions for the locations that are already part of the souvenir R. The first premise requires the domain of M (a map of locations to nonzero fractions) to cover all roots of the evaluation context K, except those that are already in the souvenir R. In other words, *if a location already appears in R then there is no need to again split off and give up a pointed-by-thread assertion for this location.* Furthermore, BindWithSouvenir augments the current souvenir by changing R to $R \cup locs(K)$ in its second premise. Thus, nested applications of this rule do not require repeatedly and redundantly giving up pointed-by-thread assertions. The rule AddSouvenir extends the current souvenir with a location ℓ. This requires framing out (temporarily giving up) a pointed-by-thread assertion for ℓ. The rule ForgetSouvenir shrinks the current souvenir.

By exploiting triples with souvenir, each of the outside rules mentioned above can be given a more concise statement. For example, the reasoning rule Poll can be more concisely formulated as PollWithSouvenir:

Poll

$$\{\mathit{outside}\,\pi\}\ \pi\colon \mathsf{poll}\ \{\lambda().\ \mathit{outside}\,\pi\}$$

PollWithSouvenir

$$[R]\{\ulcorner\mathrm{True}\urcorner\}\ \pi\colon \mathsf{poll}\ \{\lambda().\ \ulcorner\mathrm{True}\urcorner\}$$

9.2 Internals of Souvenirs

The definition of triples with souvenir appears in Figure 31. A triple with souvenir $[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}$ is expressed as an ordinary triple where the assertions *outside* π and $M \Leftarrow\!\!| \{\pi\}$ are framed out. That is, these assertions appear in the pre- and postcondition, so they are required and preserved, but they are not made available to a user who views a triple with souvenir as an abstract assertion.

$$\begin{array}{l}[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\} \;\triangleq\\ \qquad \forall M.\quad R = \mathit{dom}(M) \implies\\ \qquad\qquad \{\Phi \;*\; \mathit{outside}\,\pi \;*\; M \Lleftarrow \{\pi\}\}\;\pi\colon t\;\{\lambda v.\, \Psi\, v \;*\; \mathit{outside}\,\pi \;*\; M \Lleftarrow \{\pi\}\}\end{array}$$

Fig. 31. Definition of triples with souvenir

The domain of the map M is the set R: this ensures that, for every location in this set, a fractional pointed-by-thread assertion is indeed framed out.

A triple with souvenir describes a piece of code whose execution begins and ends outside a protected section: it cannot be used to describe a code fragment that lies inside a protected section. To establish a triple with souvenir about a whole protected section, the user must unfold the definition of triples with souvenir and drop down to the level of standard triples. Then, all of the reasoning rules for standard triples are applicable.

In our mechanization [Moine 2024], we use a more general triple that allows both "no trim" mode (§6.5) without a souvenir and normal mode with a souvenir. This general triple always frames out an "*outside*" assertion. In our case studies, this is the triple that we use most of the time.

10 CASE STUDIES

We now showcase the expressiveness of IrisFit via a series of representative case studies. We first present *logically atomic triples* [da Rocha Pinto et al. 2014; Jung et al. 2015], a standard way of specifying operations on concurrent data structures. We begin our case studies with an encoding of the fetch-and-add operation in LambdaFit, which makes use of protected sections (§10.2). Then, we present an implementation of a concurrent counter object, implemented as a pair of closures that share an internal reference (§10.3). We continue with a library for async/finish parallelism, which exploits our implementation of fetch-and-add (§10.4). We conclude this section by presenting our version of Treiber's stack (§10.5), which exploits protected sections, along the lines sketched earlier (§3). For each case study, we present the code, the specification, and some insights into the proof. For establishing concrete heap bounds, we pose in this section that a block of n fields is represented by n memory words, that is, we pose $\mathit{size}(n) = n$. Another practical choice such as $\mathit{size}(n) = n + 1$ would only affect the constant values that appear behind diamond symbols in specifications.

Our mechanization [Moine 2024] contains additional case studies that we do not cover here. They include sequential examples (a sequential algorithm written in continuation-passing style; a sequential cyclic list; three distinct implementations of sequential stacks) and concurrent examples (a spin lock; Michael and Scott's lock-free queue, with protected sections).

10.1 Atomic triples

Our specifications for fetch-and-add (§10.2) and for Treiber's stack (§10.5) involve *logically atomic triples*, also known simply as *atomic triples* [da Rocha Pinto et al. 2014; Jung et al. 2015]. In our work, an atomic triple takes the form:

$$[R]\left\langle \frac{\Phi_{private}}{\forall \vec{x}.\ \Phi_{public}} \right\rangle \pi\colon t \left\langle \frac{\lambda v.\ \Phi'_{private}}{\Phi'_{public}} \right\rangle$$

The parameter R between square brackets is a souvenir (§9). We construct our atomic triples on top of our triples with souvenir (§9) in the same way that atomic triples are usually constructed on top of ordinary triples. Intuitively, atomic triples with a souvenir $[R]$ are atomic triples whose

$$\begin{array}{l}
\mathsf{faa} \triangleq \mu_{\mathsf{ptr}} f.\, \lambda[l, i, n]. \\
\quad \mathsf{let}\; m = l[i]\; \mathsf{in} \\
\quad \mathsf{enter}\,;\; \mathsf{if}\; \mathsf{CAS}\; l[i]\; m\; (m + n) \\
\quad \mathsf{then}\; (\mathsf{exit}\,;\; m) \\
\quad \mathsf{else}\; (\mathsf{exit}\,;\; (f\;\; [l, i, n])_{\mathsf{ptr}})
\end{array}$$

FAA

$$[\emptyset] \left\langle \frac{\ell \Lleftarrow_p \{\pi\}}{\forall \vec{v}\, m.\ \ulcorner \vec{v}(i) = m \urcorner \ \ast\ \ell \mapsto \vec{v}} \right\rangle \pi \colon (\mathsf{faa}\ [\ell, i, n])_{\mathsf{ptr}} \left\langle \frac{\lambda m'. \qquad\qquad \ulcorner m' = m \urcorner}{\ell \mapsto ([i := (m + n)]\vec{v}) \ \ast\ \ell \Lleftarrow_p \emptyset} \right\rangle$$

Fig. 32. Code and specification of fetch-and-add

private pre- and postconditions are extended with pointed-by-thread assertions covering R and with an "*outside*" assertion.

The private precondition $\Phi_{private}$ and the private postcondition $\lambda v.\ \Phi'_{private}$ play the same role as the precondition and postcondition of a standard triple. The private precondition is given up by thread π when the execution of the term t begins; the private postcondition is gained by thread π when the execution of the term t ends. They are *private* in the sense that they are invisible to other threads.

The unique feature of atomic triples is the presence of a public precondition Φ_{public} and of a public postcondition Φ'_{public}. An atomic triple guarantees that the public precondition Φ_{public} continuously holds until a certain point in time, the *linearization point* [Birkedal et al. 2021], where it is atomically transformed into the public postcondition Φ'_{public}. Somewhat more accurately, an atomic triple involves a quantification over a list of variables $\vec{x}$, whose scope includes Φ_{public}, $\Phi'_{private}$, and Φ'_{public}. The existentially quantified public precondition $\exists\vec{x}.\ \Phi_{public}$ continuously holds until the linearization point is reached. There, a specific instantiation of the variables $\vec{x}$ becomes fixed. For this specific choice of $\vec{x}$, the public precondition is transformed into the public postcondition Φ'_{public}, and the value v that is eventually returned satisfies $\Phi'_{private}$.

10.2 Fetch-and-Add

The "fetch-and-add" (FAA) operation atomically increments the content of an integer reference, and returns the previous content of the reference. Although this operation is commonly provided in hardware, implementing it in LambdaFit is a fairly instructive exercise. Indeed, this code and its proof offer a typical example of the use of protected sections.

Code. In our setting, FAA takes three parameters: an address l, an offset i, and the desired increment n, an integer value. We encode FAA as a tail-recursive function whose body contains a CAS instruction enclosed in a protected sections. The code is shown in Figure 32. The recursive function is named f; its parameters are l, i and n. Initially, the content of the memory at address l and offset i is loaded into the variable m. Then, a protected section is entered, and a CAS instruction attempts to update the content of the memory from m to $m + n$. In case of success, the protected section is exited and the value m is returned. In case of failure, the protected section is also exited, and a recursive call is performed, so as to try again.

Thanks to the protected section, as soon as the CAS instruction succeeds, the memory location l can be considered as a temporary root, as opposed to an ordinary root. Indeed, as soon as CAS succeeds, it is known that the first branch of the conditional construct will be taken, so the protected section will be exited via the first exit instruction, where l is no longer a root.

Without a protected section, at the program point that follows CAS and precedes the separation of the two branches, l would still be considered a root (that is to say, an ordinary root), because it occurs inside the "else" branch, and according to the FVR (§2.1), every location that occurs in the code that lies ahead is a root.

Specification. Our specification of FAA appears in Figure 32. The private precondition consumes a pointed-by-thread assertion for the location ℓ, carrying some fraction p and the current thread identifier π. The public precondition requires that ℓ point to a block $\vec{v}$ and that the value stored at offset i in this block be m. The public postcondition asserts that FAA atomically updates m into $m + n$. Crucially, it also produces an updated pointed-by-thread assertion for ℓ, carrying the same fraction p and an *empty* set of thread identifiers. This means that as soon as the linearization point is reached, ℓ is not a root in thread π any more. This turns out to be crucial while reasoning about our async/finish library (§10.4). The private postcondition asserts that the result of FAA is m.

Proof insights. Here is how we use the reasoning rules of protected sections (Figure 18) while verifying that FAA obeys its specification. Upon entering the protected section, we use Enter and transform the assertion $\mathit{outside}\,\pi$ into the assertion $\mathit{inside}\,\pi\,\emptyset$. Then, we face the CAS instruction, a possible linearization point. We open the public precondition, and gain the points-to assertion for ℓ. By case analysis on the value that is currently stored at address l and offset i, we consider the case where CAS succeeds and the case where it fails. Let us focus on the case where it succeeds. We use CASSuccess, which updates the points-to assertion, and effectively execute the linearization point. At this point, the atomic triple requires us to prove that the public postcondition holds. Using AddTemporary, we make ℓ a temporary root: this changes the assertions $\ell \Lleftarrow_p \{\pi\}$ and $\mathit{inside}\,\pi\,\emptyset$ into $\ell \Lleftarrow_p \emptyset$ and $\mathit{inside}\,\pi\,\{\ell\}$. By giving up the points-to and pointed-by-thread assertions, we fulfill the public postcondition. Then, we use IfTrue and enter the first branch of the "if" statement. There, TrimInside lets us change the assertion $\mathit{inside}\,\pi\,\{\ell\}$ to $\mathit{inside}\,\pi\,\emptyset$. This allows us to exit the protected section using Exit. We finish the proof with Val.

10.3 A Concurrent Counter Object

Our next example is a concurrent monotonic "counter" object, whose internal state is stored in a mutable reference, and whose access is mediated by a pair of closures: a closure i *increments* the counter; a closure g *gets* its current value. This is an example of a procedural abstraction [Reynolds 1975], also known as an *object*: indeed, "an object is a value exporting a procedural interface to data or behavior" [Cook 2009]. Crucially, a counter can be used concurrently by several threads.

Code. The top of Figure 33 presents the code that we verify. The function call $(\mathsf{ref}\ [x])_{\mathsf{ptr}}$ allocates a mutable reference, that is, a block of size 1. The function call $(\mathsf{pair}\ [x, y])_{\mathsf{ptr}}$ allocates a mutable pair, that is, a block of size 2. The function call $(\mathsf{ignore}\ [x])_{\mathsf{ptr}}$ ignores its argument and returns the unit value. The function call $(\mathsf{create}\ [])_{\mathsf{ptr}}$ returns a fresh "counter", that is, a pair of two closures i and g. Both closures point to an internal reference r, which is initialized to the value 0. The closure i uses our fetch-and-add function (§10.2) and ignores its result.

Specifications. Figure 33 presents the specification of our concurrent counter. It is inspired by a specification that appears in lecture notes [Birkedal and Bizjak 2023]. It relies on an abstract assertion $\mathit{counter}\ i\ g\ p\ n$ where i is the location of the "increment" closure, g is the location of the "get" closure, $p \in (0; 1]$ is a fraction that represents a *share* of the ownership of the counter, and n, a natural number, represents a *past contribution* to the current value of the counter. If p is 1 then the contribution n is in fact the current value of the counter.

The equivalence axiom in Figure 33 shows that "*counter*" assertions can be split and joined; both the fraction and the contribution are then split or joined by addition. This allows a counter to be

$$
\begin{array}{l}
\text{ref} \triangleq \mu_{\text{ptr}\_}.\,\lambda[x].\\
\quad \text{let } r = \text{alloc } 1 \text{ in}\\
\quad r[0] \leftarrow x\,;\, r\\
\text{pair} \triangleq \mu_{\text{ptr}\_}.\,\lambda[x,y].\\
\quad \text{let } r = \text{alloc } 2 \text{ in}\\
\quad r[0] \leftarrow x\,;\, r[1] \leftarrow y\,;\, r
\end{array}
$$

$$
\begin{array}{l}
\text{ignore} \triangleq \mu_{\text{ptr}\_}.\,\lambda[x].\,()\\
\text{create} \triangleq \mu_{\text{ptr}\_}.\,\lambda[].\\
\quad \text{let } r = (\text{ref } [0])_{\text{ptr}} \text{ in}\\
\quad \text{let } i = \mu_{\text{clo}\_}.\,\lambda\_.\,(\text{ignore } [(\text{faa } [r,0,1])_{\text{ptr}}])_{\text{ptr}} \text{ in}\\
\quad \text{let } g = \mu_{\text{clo}\_}.\,\lambda\_.\,r[0] \text{ in}\\
\quad (\text{pair } [i,g])_{\text{ptr}}
\end{array}
$$

$$(\mathit{counter}\; i\; g\; (p_1 + p_2)\; (n_1 + n_2)) \quad \equiv \quad (\mathit{counter}\; i\; g\; p_1\; n_1 \;*\; \mathit{counter}\; i\; g\; p_2\; n_2)$$

$$[\emptyset]\;\left\{\Diamond 7\right\}\;\pi\colon (\text{create } [])_{\text{ptr}}\;\left\{\lambda\ell.\,\exists i\, g.\; \begin{array}{c} \ell \mapsto [i; g] \;*\; \mathit{counter}\; i\; g\; 1\; 0\\ \ell \Mapsfrom \{\pi\} \;*\; \ell \mapsfrom \emptyset\\ i \Mapsfrom \emptyset \;*\; i \mapsfrom \{+\ell\}\\ g \Mapsfrom \emptyset \;*\; g \mapsfrom \{+\ell\} \end{array}\right\}$$

$$[\emptyset]\;\left\{\mathit{counter}\; i\; g\; p\; n\right\}\quad \pi\colon (i\; [])_{\text{clo}}\quad \left\{\lambda().\; \mathit{counter}\; i\; g\; p\; (n+1)\right\}$$

$$[\emptyset]\;\left\{\mathit{counter}\; i\; g\; p\; n\right\}\quad \pi\colon (g\; [])_{\text{clo}}\quad \left\{\lambda m.\; \begin{array}{c} \ulcorner n \leq m \;\wedge\; (p = 1 \implies n = m)\urcorner\\ \mathit{counter}\; i\; g\; p\; n \end{array}\right\}$$

$$\left(\begin{array}{c} \mathit{counter}\; i\; g\; 1\; n\\ i \Mapsfrom \emptyset \;*\; i \mapsfrom \emptyset\\ g \Mapsfrom \emptyset \;*\; g \mapsfrom \emptyset \end{array}\right) \quad \Rrightarrow \quad \left(\Diamond 5\right)$$

Fig. 33. Code and specification of a concurrent monotonic counter

used in a concurrent setting: the user can split the "*counter*" predicate into several parts and give a part to each participating thread. In the end, the user can gather all parts, draw conclusions about the final value of the counter, and logically deallocate the counter.

The specification of $(\text{create } [])_{\text{ptr}}$ states that this call consumes 7 space credits (1 credit for the shared reference, 2 credits for each closure, and 2 credits for the pair). It returns a pair ℓ of two locations i and g such that $\mathit{counter}\; i\; g\; 1\; 0$ holds. This assertion captures the full ownership of the counter, and specifies that its current value is 0.

Figure 33 also shows the specifications of calls to i and g. Both calls require an assertion of the form $\mathit{counter}\; i\; g\; p\; n$. The postcondition of a call to the "increment" closure contains an updated assertion $\mathit{counter}\; i\; g\; p\; (n+1)$. The postcondition of a call to the "get" closure contains an unmodified "*counter*" assertion. Furthermore, it guarantees that the natural number m that is returned by this call is no less than the past contribution n and, in the case where p is 1, is equal to the past contribution.

Last, Figure 33 shows the reasoning rule for deallocating a counter. This rule requires full ownership of the counter as well as pointed-by-heap and pointed-by-thread assertions for the closures i and g, with fraction 1 and empty sets—this witnesses that both closures are unreachable. In exchange, the rule produces 5 spaces credits. The 2 credits corresponding to the pair produced by create can be recovered independently.

Proof insights. The proof that the counter obeys its specification uses ghost state in a standard way [Birkedal and Bizjak 2023, §8.7]. The internal definition of the abstract predicate "*counter*" involves an existential quantification over the shared location r: indeed, this location does not appear in the specification. The assertion $\mathit{counter}\; i\; g\; p\; n$ contains a pointed-by-thread assertion for the location r with fraction p. Moreover, the assertion $\mathit{counter}\; i\; g\; p\; n$ contains *Spec* assertions (§8.5) for the closures i and g. The environments that appear in these *Spec* assertions map r to the

$$
\begin{array}{l}
\mathsf{create} \triangleq \mu_{\mathsf{ptr}}\_.\,\lambda[].\\
\quad (\mathsf{ref}\ [0])_{\mathsf{ptr}}\\
\mathsf{async} \triangleq \mu_{\mathsf{ptr}}\_.\,\lambda[l,f].\\
\quad (\mathsf{faa}\ [l,0,1])_{\mathsf{ptr}}\,;\\
\quad \mathsf{fork}\,((f\ [])_{\mathsf{clo}}\,;\,(\mathsf{ignore}\ [(\mathsf{faa}\ [l,0,-1])_{\mathsf{ptr}}])_{\mathsf{ptr}})
\end{array}
\qquad
\begin{array}{l}
\mathsf{finish} \triangleq \mu_{\mathsf{ptr}} f.\,\lambda[l].\\
\quad \mathsf{if}\ l[0] = 0\\
\quad \mathsf{then}\ ()\\
\quad \mathsf{else}\ (f\ [l])_{\mathsf{ptr}}
\end{array}
$$

AFCreate

$$[\emptyset]\{\Diamond 1\}\ \pi\colon (\mathsf{create}\ [])_{\mathsf{ptr}}\ \{\lambda\ell.\ \mathsf{AF}\,\ell \ \ast\ \ell \Mapsfrom_{\frac{1}{2}} \{\pi\} \ \ast\ \ell \mapsfrom_1 \emptyset\}$$

AFAsync

$$\frac{\forall\pi'.\quad [\{\ell\}]\{f \Mapsfrom_p \{\pi'\} \ \ast\ \Phi\}\ \pi'\colon (f\ [])_{\mathsf{clo}}\ \{\lambda().\ \Psi\}}{[\{\ell\}]\{\mathsf{AF}\,\ell \ \ast\ f \Mapsfrom_p \{\pi\} \ \ast\ \Phi\}\ \pi\colon (\mathsf{async}\ [\ell,f])_{\mathsf{ptr}}\ \{\lambda().\ \mathit{spawned}\,\ell\,\Psi\}}$$

AFFinish

$$[\emptyset]\{\mathsf{AF}\,\ell \ \ast\ \ell \Mapsfrom_{\frac{1}{2}} \{\pi\}\}\ \pi\colon (\mathsf{finish}\ [\ell])_{\mathsf{ptr}}\ \{\lambda().\ \mathit{finished}\,\ell\}$$

FinishedSpawned

$$\mathit{finished}\,\ell \ \ast\ \mathit{spawned}\,\ell\,\Psi \quad \Rrightarrow \quad \Psi$$

FinishedFree

$$\mathit{finished}\,\ell \ \ast\ \ell \mapsfrom_1 \emptyset \quad \boldsymbol{\Rrightarrow} \quad \Diamond 1$$

AFPersistent

$\mathsf{AF}\,\ell$ is persistent

FinishedPersistent

$\mathit{finished}\,\ell$ is persistent

Fig. 34. Code and specification of an async/finish library

fraction $\frac{1}{2}$, which means that each closure owns one half of the pointed-by-heap assertion for the location r.

10.4 An Async/Finish Library

The async/finish paradigm was introduced in X10 [Charles et al. 2005], as a generalization of the spawn/sync mechanism of Cilk [Blumofe et al. 1996], spawn/sync itself being a generalization of the binary fork/join paradigm. The async/finish paradigm allows spawning an arbitrary number of tasks before waiting at a common join point. More precisely, the construct "async" allows spawning new tasks, whereas "finish" performs synchronization: it blocks until all previously spawned tasks terminate. In this section, we show how to encode these two constructs in LambdaFit using a shared mutable reference that is updated using a fetch-and-add operation (§10.2). We then provide specifications in IrisFit, and show that the space credits associated to the shared reference can be recovered as soon as "finish" returns.[9] A strength of our specification is that it allows for *nested* spawns: a spawned task can itself spawn tasks.

Code. The code of our async/finish library is presented in the top part of Figure 34. The library uses a reference that we call the *session*. A session is a channel through which tasks communicate. It stores the number of currently running tasks.

The function $(\mathsf{create}\ [])_{\mathsf{ptr}}$ returns a fresh session, with zero running tasks.

The function $(\mathsf{async}\ [l,f])_{\mathsf{ptr}}$ expects a session l and a closure f as arguments. It first atomically increments the session, hence recording the existence of a new running task, then forks off a thread

[9] That is to say, as soon as every task reaches the linearization point of the fetch-and-add operation to signal that it is done. A task can still execute some code past the linearization point before actually terminating.

that invokes the closure f with no arguments. When this invocation terminates, it atomically decrements the session, thereby recording that this task is finished.

The function $(\mathsf{finish}\ [l])_{\mathsf{ptr}}$ consists of an active waiting loop. This loop ends when it observes that the session contains the value 0, which guarantees that all previously spawned tasks have terminated.

Specifications. The bottom part of Figure 34 presents the specification of our async/finish library.

According to AFCreate, $(\mathsf{create}\ [])_{\mathsf{ptr}}$ consumes one space credit, which corresponds to the space occupied by the session, and returns a location ℓ such that $\mathsf{AF}\ \ell$ holds. This persistent assertion guarantees that ℓ is a session. The postcondition also provides pointed-by-thread and pointed-by-heap assertions for the location ℓ. The pointed-by-heap assertion carries the fraction $\frac{1}{2}$; the other half is hidden from the user.

The specification of $(\mathsf{async}\ [\ell, f])_{\mathsf{ptr}}$ is stated as a triple featuring a souvenir on ℓ. This means that, for the duration of this call, ℓ is a root. The precondition requires ℓ to be a session. A fractional pointed-by-thread assertion for the closure f, as well as an arbitrary assertion Φ, are consumed and transmitted to the new task, which invokes the closure f. The premise of the rule AFAsync requires the user to prove that, under an arbitrary thread identifier π', this invocation is safe and satisfies some postcondition Ψ. The postcondition of $(\mathsf{async}\ [\ell, f])_{\mathsf{ptr}}$ provides a witness that this task was spawned, in the form of the assertion $\mathit{spawned}\,\ell\,\Psi$. This assertion is not persistent: it can be understood as a unique permission to collect Ψ once the task is finished.

The specification of f in the premise of AFAsync is again a triple with a souvenir of ℓ. This formulation allows f to itself use async. Using an ordinary triple there would place a stronger requirement on f and would forbid the use of async inside f.

According to AFFinish, $(\mathsf{finish}\ [\ell])_{\mathsf{ptr}}$ consumes the pointed-by-thread assertion that was produced by create. This forbids any further use of the session ℓ: indeed, both AFAsync and AFFinish require a pointed-by-thread assertion for ℓ.[10] The postcondition contains the persistent assertion $\mathit{finished}\,\ell$, which witnesses that this session has been ended.

The ghost update FinishedSpawned states that if the witness $\mathit{finished}\,\ell$ is at hand then the assertion $\mathit{spawned}\,\ell\,\Psi$ can be converted to Ψ. This reflects the idea that if the session has been ended, then all tasks must have terminated: so, a permission to collect Ψ can indeed be converted to Ψ. The ghost update FinishedFree states that if the session has ended then abandoning the pointed-by-heap assertion for ℓ allows recovering the space credit associated with the session ℓ.

Proof insights. The assertion $\mathsf{AF}\ \ell$ is internally defined as an Iris invariant. Among other things, this invariant imposes a protocol on the pointed-by-thread assertion for the session ℓ. Initially, the invariant contains a pointed-by-thread assertion carrying the fraction $\frac{1}{2}$ and an empty set; the other half is given to the user by Create. Each spawned task gets a fraction of this assertion: indeed, spawning a task involves "fork", and our Fork rule requires updating a pointed-by-thread assertion so as to reflect the fact that ℓ is a root of the new thread. When a task signals that it is finished, it surrenders its fractional pointed-by-thread assertion, carrying an *empty* set of thread identifiers. Hence, once every task has terminated, the invariant again contains $\ell \Lleftarrow_{\frac{1}{2}} \emptyset$.

How and when exactly does a task signal that it is finished? This is done via a fetch-and-add (FAA) operation, which decrements the count of active tasks, and takes effect precisely at the linearization point of this FAA operation. Hence, as soon as this linearization point is reached, the invariant requires this task to surrender its fractional pointed-by-thread assertion. Fortunately, our specification of FAA (§10.2) allows this: the pointed-by-thread assertion $\ell \Lleftarrow_{p} \emptyset$ appears in the public postcondition in FAA.

[10] In the case of AFAsync, this is implicit in the fact that the conclusion of the rule is a triple with a souvenir on ℓ.

StackCreate

$$[\emptyset]\{\Diamond 1\}\ \pi\colon (\mathsf{create}\ [\,])_{\mathsf{ptr}}\ \{\lambda \ell.\ \mathit{stack}\,\ell\,[\,] \ \ast\ \ell \mathrel{\Leftarrow\!\!\!\mid} \{\pi\} \ \ast\ \ell \mathrel{\leftarrow\!\!\!\mid} \emptyset\}$$

StackPush

$$[\{\ell\}] \left\langle \frac{\Diamond 2 \ \ast\ v \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\} \ \ast\ v \mathrel{\leftarrow\!\!\!\mid}^{>0}_q \emptyset}{\forall\, vpqs.\qquad \mathit{stack}\,\ell\ vpqs} \right\rangle \pi\colon (\mathsf{push}\ [\ell; v])_{\mathsf{ptr}} \left\langle \frac{\lambda().\qquad \ulcorner \mathrm{True} \urcorner}{\mathit{stack}\,\ell\ ((v,p,q) :: vpqs)} \right\rangle$$

StackPop

$$[\{\ell\}] \left\langle \frac{\ulcorner \mathrm{True} \urcorner}{\forall v\ p\ q\ vpqs.\ \ \mathit{stack}\,\ell\ ((v,p,q) :: vpqs)} \right\rangle \pi\colon (\mathsf{pop}\ [\ell])_{\mathsf{ptr}} \left\langle \frac{\lambda w.\qquad \ulcorner w = v \urcorner \ \ast\ v \mathrel{\Leftarrow\!\!\!\mid}_p \{\pi\}}{\mathit{stack}\,\ell\ vpqs \ \ast\ \Diamond 2 \ \ast\ v \mathrel{\leftarrow\!\!\!\mid}^{>0}_q \emptyset} \right\rangle$$

StackFree

$$\mathit{stack}\,\ell\ vpqs \ \ast\ \ell \mathrel{\Leftarrow\!\!\!\mid} \emptyset \ \ast\ \ell \mathrel{\leftarrow\!\!\!\mid} \emptyset \quad \Rrightarrow \quad \Diamond(1 + 2 \times |vpqs|) \ \ast \mathop{\Large\ast}\limits_{(v,p,q)\in vpqs} (v \mathrel{\Leftarrow\!\!\!\mid}_p \emptyset \ \ast\ v \mathrel{\leftarrow\!\!\!\mid}^{>0}_q \emptyset)$$

Fig. 35. Specification of Treiber's Stack

The absence of a "later" modality in front of Ψ in FinishedSpawned may seem surprising. As the assertion Ψ has transited through an invariant, an Iris expert might expect it to be guarded by such a modality. The usual way to eliminate a "later" modality is through a physical step, yet this rule is a ghost update. Fortunately, IrisFit supports and takes advantage of *later credits* (§6.2). A later credit is a piece of ghost state that is produced by a physical step and that can later be used to eliminate a "later" modality. With each spawned task, we are able to internally associate one later credit, which we obtain from the function call $(\mathsf{async}\ [\ell, f])_{\mathsf{ptr}}$. By exploiting this later credit, we can eliminate the "later" modality in front of Ψ before giving this assertion back to the user.

10.5 Treiber's Stack

Code. The code that we verify is the code of Figure 2, translated to LambdaFit syntax. A reference is a block of size 1; a stack cell is a block of size 2.

Specifications. Figure 35 presents our specification of Treiber's stack. The stack is described in terms of the abstract predicate $\mathit{stack}\,\ell\ vpqs$, where ℓ is the location of the stack and $vpqs$ is its mathematical model. This model is a list of triples (v, p, q) of a value v and two positive fractions p and q. The list of the values v describes the content of the stack. For each value v, the fractions p and q describe what quantity of the pointed-by-thread and pointed-by-heap assertions for the value v have been acquired by the stack. Having the stack acquire a fractional pointed-by-heap assertion for the value v lets us record that this value is pointed to by a stack cell without revealing or even mentioning the address of this cell. Having the stack acquire a fractional pointed-by-thread assertion for the value v lets us express a plausible specification for "pop". Indeed, "pop" needs to read the value v from the heap: then, the Load rule requires (and updates) a fractional pointed-by-thread assertion for v. Expecting the caller to supply this assertion seems impractical, so it must be found in the stack itself.

The assertion $\mathit{stack}\,\ell\ vpqs$ is not fractional: it represents the full ownership of the stack. To allow the stack to be accessed by several concurrent threads, the user must share this assertion. This is typically achieved via an Iris invariant [Birkedal and Bizjak 2023].

According to StackCreate, creating a new stack consumes one space credit. This is the size of the reference that holds the address of the top stack cell. The result is a fresh location ℓ that represents an empty stack.

The specification of $(\mathsf{push}\ [\ell; v])_{\mathsf{ptr}}$, expressed by StackPush, is an atomic triple with a souvenir on ℓ. The private precondition requires two space credits, which is the size of a new stack cell, as well as fractional pointed-by-heap and pointed-by-thread assertions for the value v that is pushed onto the stack. Together, the public precondition and postcondition indicate that the model of the stack is atomically updated from *vpqs* updated to $(v, p, q) :: vpqs$ at the linearization point.

The specification of $(\mathsf{pop}\ [\ell])_{\mathsf{ptr}}$, expressed by StackPop, is also an atomic triple with a souvenir on ℓ. The public precondition and postcondition indicate that the model of the stack is atomically updated from $(v, p, q) :: vpqs$ to *vpqs*. Furthermore, according to the public postcondition, at the linearization point, two space credits are produced, and a pointed-by-heap assertion for v, carrying an empty multiset of predecessors, is produced as well, as a pointer from the stack to v has been destroyed.

Our specification of "pop" exhibits a certain asymmetry: whereas the space credits and the pointed-by-heap assertion appear in the *public* postcondition, which means that they are produced at the linearization point, the pointed-by-thread assertion appears in the *private* postcondition. which means that it is produced when the function returns. The space credits and the pointed-by-heap assertion can be produced at the linearization point because there we are already able to logically deallocate the stack cell and to argue that a pointer from the stack to v has been destroyed. However, the pointed-by-thread assertion cannot be surrendered as part of the public postcondition, because the value v is read from the heap *after* the linearization point has been passed.

The last rule in Figure 35, StackFree, logically deallocates a (possibly nonempty) stack. The assertion $\mathit{stack}\ \ell\ \mathit{vpqs}$, as well as empty pointed-by-thread and pointed-by-heap assertions for ℓ, are consumed. A number of space credits are produced, which reflect the overall size occupied by the stack data structure in the heap: one credit for the toplevel reference, plus two credits per stack cell. The pointed-by-thread and pointed-by-heap assertions associated with every triple (v, p, q) in the stack are also produced. Of course, in the common case where *vpqs* is an empty list, this rule can be significantly simplified.

Proof insights. As argued earlier (§3), the main difficulty of the proof is to produce space credits when a "pop" operation succeeds. This requires logically deallocating the stack cell that is being extracted. This in turn requires exhibiting both an empty pointed-by-thread assertion and an empty pointed-by-heap assertion for this cell. Yet, neither of these assertions is easy to obtain.

Let us discuss the pointed-by-thread assertion first. The difficulty is that "push" and "pop" are *invisible readers* [Alistarh et al. 2018]: these operations read the top of the stack (that is, the address of a stack cell) without synchronization. Such a read normally requires updating a pointed-by-thread assertion for the cell whose address is thus obtained. However, here, we do not wish to record that this cell is pointed to by the current thread. Fortunately, these reads occur inside protected sections. Hence, we use LoadInside, which updates an "*inside*" assertion instead of a pointed-by-thread assertion. This allows the stack's invariant to keep an *empty* pointed-by-thread assertion, at all times, for every stack cell. This in turn allows a successful "pop" operation to extract this empty pointed-by-thread assertion out of the invariant. Maintaining empty pointed-by-thread assertions for locations that are acquired only inside protected sections is a typical idiom.

Next, let us discuss the pointed-by-heap assertion. Here, the difficulty is that a stack cell ℓ may be pointed to by a new cell ℓ' that has just been allocated by an ongoing "push" operation. This scenario was discussed earlier (§3.2). Hence, each ongoing "push" holds an assertion $\ell \mapsfrom_p \{+\ell'\}$, where ℓ is the stack cell that "pop" is attempting to extract and ℓ' is the new stack cell that "push" has allocated. Now, how can "pop" obtain the assertion $\ell \mapsfrom_1 \emptyset$ that is required to allow logical deallocation? We answer this question via an original technique that we dub *helping with logical deallocation*: the thread that successfully pops the stack cell ℓ also takes care of logically deallocating the predecessor cells ℓ' that have been allocated by ongoing “push” operations.[11] The logical deallocation of these locations is made possible by the protected section in “push”. This approach has a somewhat strange consequence: in the proof of “push”, it may be the case that the cell ℓ' has been logically deallocated by another thread, yet “push” still needs to access this cell. Fortunately, IrisFit allows this: for example, the proof of “push” makes use of the rule StoreDead.

11 RELATED WORK

11.1 Polling Points

A stop-the-world event may be viewed as an asynchronous interruption: a thread that requests garbage collection stops the execution of all other threads. Such an interruption can be implemented using hardware interrupts, but this scheme can be expensive and non-portable [Feeley 1993]. Another approach is to let the compiler insert explicit tests for interruptions into the code. These tests appear in the literature under various names, including *polling points* [Feeley 1993], *GC points* [Agesen 1998], *yield points* [Lin et al. 2015], and *safe points* [Sivaramakrishnan et al. 2020]. Let us refer to them collectively as *safe points*. Safe points are typically inserted by the compiler in such a way that no computation can run forever without encountering a safe point. When a thread encounters a safe point, it tests whether some other thread has requested garbage collection. If so, it pauses and passes control to the runtime system. Once all threads have paused in this way, the runtime system performs a global garbage collection phase.

Safe points are used in the Jalapeño/Jikes RVM [Alpern et al. 1999, 2005] and in OCaml 5 [Sivaramakrishnan et al. 2020]. The existence of safe points is not revealed to the programmer, who is not expected to know about their existence and is given no means of controlling their placement. As an experimental feature, the OCaml 5 compiler does offer a `[@poll error]` attribute [Jaffer 2021]. This attribute is placed on a function definition. An attempt by the compiler to insert a safe point into a function that carries this attribute causes a compile-time error. This lets the programmer check that a function body does not contain any safe point, therefore is (de facto) a protected section. At this time, there is not a clear consensus whether this feature is useful and corresponds to the needs of expert programmers.

Safe points, as described above, and polling points, as proposed in this paper, are two related yet distinct concepts. Indeed, in our view, safe points play two distinct roles. On the one hand, they are *polling points*, in the sense of this paper: they are points where a thread must stop and allow garbage collection to take place if it has been requested. On the other hand, at the same time, they are delimiters (that is, starting points and ending points) of *protected sections*: indeed, *the GC cannot run unless every thread has reached a safe point.* We believe that our design, where protected sections and polling points are separate concepts, is better behaved. In particular, it enjoys *monotonicity* properties: inserting a new polling point, creating a new protected section, or enlarging an existing protected section *restricts* the set of possible behaviors of the program.[12] In contrast, in a setting where only a “safe point” construct is offered by the language, inserting a new safe point creates one more program point where the GC is allowed to run, therefore can *enlarge* the set of possible behaviors of the program and compromise the program’s worst-case heap space complexity. In short, in such a setting, automated safe point insertion is arguably unsafe!

In our approach, the user *explicitly* inserts enough protected sections to (verifiably) obtain the desired worst-case heap space complexity, then lets the compiler *implicitly* insert enough polling

[11] Note that these ongoing “push” operations will fail, because the top stack cell previously observed has been replaced.

[12] Polling points must be inserted only outside protected sections. In our setting, inserting a new polling point does not create a new opportunity for the GC to run, because outside protected sections, the GC is everywhere allowed to run.

points to guarantee liveness, without endangering the program's space complexity. This is expressed by Theorem 7.2.

11.2 Protected Sections

In the production systems that we are aware of, the concept that seems closest to our protected sections appears in the .NET runtime system, where it was introduced in 2015, with performance in mind [Lander 2015]. The API of the `GC` module [Microsoft 2024] provides a method `TryStartNoGCRegion(Int64)` and a method `EndNoGCRegion()`. A "NoGC region" is not quite a protected section in our sense, though, as allocation is permitted inside a "NoGC region". The integer parameter of the method `TryStartNoGCRegion` is a request for a certain amount of free heap space: garbage collection takes place at this point so as to guarantee that this much free space exists. Allocation requests within the "NoGC region" are then served out of this pre-allocated free space. However, if the runtime system runs out of free space while some thread is inside a "NoGC region", then garbage collection will take place.

Beside performance, another possible motivation for temporarily disabling garbage collection is safety. Feeley [1993, §1.2.1] discusses why "critical sections"—sections in which the GC must not run—may be needed for safety reasons. He takes the example of a store instruction that stores a 64-bit pointer into memory and that is decomposed into two 32-bit stores. In between the two stores, the memory is in an inconsistent state and must not be read by the GC.

To the best of our knowledge, our paper is the first where a notion of protected section is introduced for complexity reasons, that is, with the aim of guaranteeing tighter worst-case heap space complexity bounds.

11.3 Reasoning about Space without a GC

Hofmann [1999, 2003] introduces space credits in the setting of an affine type system for the λ-calculus. Hofmann [2000] and Aspinall and Hofmann [2002] adapt the idea to LFPL, a first-order functional programming language without GC and with explicit destructive pattern matching. There, a value of type $\diamond$ exists at runtime and can be understood as a pointer to a free block in the heap. Subsequent work aims at automating space complexity analyses. In particular, Hofmann and Jost [2003] propose an affine type system where types carry space credits. Hofmann and Jost [2006]; Hofmann and Rodriguez [2009, 2013] analyze a variant of Java where garbage collection has been replaced with explicit deallocation. RaML [Hoffmann et al. 2012a,b, 2017] analyzes a fragment of OCaml, also without GC and with explicit destructive pattern matching. Niu and Hoffmann [2018] present a type-based amortized space analysis for a pure, first-order programming language where destructive pattern matching can be applied to shared objects, an unusual feature. Their system performs significant over-approximations: when a data structure becomes shared, the logic charges the cost of creating a copy of this data structure. As far as we understand, this analysis can be used to reason in a sound yet very conservative way about a programming language with GC. Kahn and Hoffmann [2021] present a system that is equipped with more flexible typing rules than its predecessors and therefore can derive tighter resource consumption bounds. Hoffmann and Jost [2022] offer a survey of two decades of work on automated amortized resource analysis (AARA).

Following the ideas of LFPL, Lorenzen et al. [2023] introduce a calculus with "reuse" credits. Explicit destructive pattern matching produces reuse credits, which can be used to satisfy a new allocation. Because the system allows fragmentation, reuse credits cannot be joined. The goal of Lorenzen et al. [2023] is to statically detect *fully in-place* functions—that is, functions that do not need to allocate new memory. This includes, for example, functions that reuse the heap space occupied by their arguments.

Chin et al. [2005, 2008] present a type system that automatically keeps track of data structure sizes. The type system incorporates an alias analysis, which distinguishes between shared and unique objects and allows unique objects to be explicitly deallocated. Shared objects can never be logically deallocated. Specifications indicate how much memory a method may need (a high-water mark) and how much memory it releases, in terms of the sizes of the arguments and results.

Compared with type systems, program logics offer weaker automation but greater expressiveness. Aspinall et al. [2007] propose a VDM-style program logic, where postconditions depend not only on the pre-state, post-state, and return value, but also on a cost. Atkey [2011] proposes an extension of Separation Logic with an abstract notion of resource, such as time or space, and introduces an assertion that denotes the ownership of a certain amount of resources.

All of the work cited above concerns languages with explicit memory deallocation, where there is no need to reason about unreachability. Reasoning about unreachability in the setting of a static analysis or program logic is a central challenge.

11.4 Reasoning about Space with a GC

Hur et al. [2011] propose a Separation Logic for the combination of a low-level language with explicit deallocation and a high-level language with a GC. They are interested in verifying just safety, not space complexity.

Madiot and Pottier [2022] and Moine et al. [2023] propose Separation Logics that allow reasoning about space in the presence of a GC.

The logic presented by Madiot and Pottier [2022] concerns a low-level language with explicit stack cells. Its reasoning rules are intended to support concurrency, but the paper does not provide any case study.

The logic presented in our previous paper [Moine et al. 2023] concerns a high-level language, where the call stack is implicit, but is restricted to a sequential setting. This paper also introduces support for closures. The logic relies on a distinction between *visible roots*—the roots of the term under focus—and *invisible roots*—the roots of the evaluation context. The logic keeps track of invisible roots using a *Stackable* assertion, and introduces the idea that *Stackable* assertions must be “forcibly framed out” at applications of the Bind rule. We re-use this idea in our own Bind rule (§6.4), but replace *Stackable* assertions with pointed-by-thread assertions, which are better suited to a concurrent setting. In so doing, we remove the distinction between visible roots and invisible roots, which does not seem to make sense in a concurrent setting; our pointed-by-thread assertions keep track of all (ordinary) roots. In contrast, Moine et al. [2023] do not keep track of visible roots via an a dedicated assertion: indeed, in their setting, it suffices to inspect the term under focus to determine the set of visible roots. This allows them to offer a standard Load rule, whereas our Load rule updates a pointed-by-thread assertion for the value that is loaded (§6.2).

Our mechanization [Moine 2024] includes an encoding inside IrisFit of our previous logic for sequential programs [Moine et al. 2023]. This encoding demonstrates that our concurrent program logic can be used to reason about sequential programs with no overhead.

11.5 Space-Related Results for Compilers

Paraskevopoulou and Appel [2019] prove that, in the presence of a GC, closure conversion is safe for space: that is, it does not change the space consumption of a program. They view closure conversion as a transformation from a CPS-style λ-calculus into itself. This calculus is equipped with two different environment-based big-step operational semantics. The “source” semantics implicitly constructs a closure for each function definition by capturing the relevant part of the environment and storing it in the heap. The “target” semantics performs no such construction: it requires every function to be closed. In either semantics, the roots are defined as the locations that occur in the environment. Up to the stylistic difference between a substitution-based semantics and an environment-based semantics, this definition is equivalent to the "free variable rule" (FVR) [Morrisett et al. 1995].

Besson et al. [2019] prove that (an enhanced version of) CompCert [Leroy 2021] preserves memory consumption when compiling C programs.

In a sequential setting, Gómez-Londoño et al. [2020] prove that the CakeML compiler respects a cost model that is defined at the level of the intermediate language DataLang, which serves as the target of closure conversion. Our cost model is analogous to theirs. Our work and theirs are complementary: whereas they prove that the CakeML compiler respects the DataLang cost model, we show how to establish space complexity bounds about source programs, based on a similar cost model. One could in principle adapt IrisFit to DataLang. Then, one would be able to use IrisFit to establish a space complexity bound about a source CakeML program, to compile this program down to machine code using the CakeML compiler, and to obtain a machine-checked space complexity guarantee about the compiled code.

11.6 Safe Memory Reclamation Schemes

Manual memory management can be so difficult in a concurrent setting that programmers often rely on semi-automatic *safe memory reclamation* (SMR) schemes. Two main families exist, namely hazard pointers [Michael 2004; Michael et al. 2023] and read-copy-update (RCU) [McKenney 2004; McKenney et al. 2023]. The two families offer roughly similar APIs. First, the user declares *hazardous* locations for a delimited scope. While it is marked hazardous, a location is not deallocated. Second, the user can *retire* a location to indicate that this location is no longer needed. The SMR implementation deallocates a retired location once it is not marked hazardous by any thread.

RCU seems particularly close to our concept of a protected section. Indeed, RCU declares *every* pointer hazardous inside a certain section of the code. Yet, there is not a perfect analogy between the two. Indeed, garbage collection provides a strong guarantee: *no dangling pointer can exist.* SMR schemes, on the contrary, tolerate dangling pointers. Hence, with RCU, a location that the code mentions, but without reading or writing it, does not need to be protected. For example, the "push" operation of Treiber's stack does *not* need an RCU section [Jung et al. 2023, mechanization], whereas the "pop" operation does need one. Indeed, the push operation never accesses the content of an internal list cell. Hence, it is not dangerous if such a location is deallocated in the meantime.

Equipping SMR schemes with abstract Separation Logic specifications and verifying them has long been a challenge. Treiber's stack has been the first data structure based on hazard pointers to be verified. This task was tackled several times using different variants of Concurrent Separation Logic [Parkinson et al. 2007; Fu et al. 2010]. Tofan et al. [2011] verify Treiber's stack both with hazard pointers and with garbage collection (though without a heap space complexity analysis). They show that a large part of the main invariant can be shared between the two proofs. Gotsman et al. [2013] provide the first general framework for verifying programs using SMR schemes in Separation Logic, making use of temporal logic reasoning. Jung et al. [2023] provide a more abstract framework, where temporal reasoning is replaced with ownership arguments. Their work unveils a close relationship between RCU and garbage collection. Indeed, RCU allows accessing any location that was *not* retired when the current RCU section was entered. (There is a loose analogy with our liveness-based cancellable invariants: to access such an invariant, one must eliminate the case where ℓ has been logically deallocated.) To prove that a location is *not* retired at a certain point in time, Jung et al. [2023] express the topology of data structures using pointed-by-heap assertions, which they borrow from our prior paper [Moine et al. 2023]. Like us, when retiring a location, they require the predecessors of this location to have been previously retired.

Outside the Separation Logic world, Meyer and Wolff [2019] propose an API for SMR schemes, in the form of an observer automaton, inspired by the temporal reasoning of Gotsman et al. [2013]. Meyer and Wolff [2019] make use of the observer automaton to decorrelate the verification of lock-free data structures from the SMR implementation, allowing them to develop an automatic linearizability checker.

12 CONCLUSION AND FUTURE WORK

We have presented LambdaFit, a lambda-calculus with shared-memory concurrency and tracing garbage collection. In particular, LambdaFit is equipped with protected sections, a new, realistic construct that programmers can and sometimes must exploit to ensure that fine-grained concurrent data structures have the desired worst-case heap space complexity. We believe that protected sections are a necessary part of a concurrent programmer's toolbox, and that they should be considered for inclusion in high-level languages.

Furthermore, we have presented IrisFit, a Concurrent Separation Logic with space credits, which allows expressing and verifying worst-case heap space bounds about LambdaFit programs. IrisFit features pointed-by-heap and pointed-by-thread assertions, which offer a compositional means of keeping track of the various ways through which a memory block is reachable. These assertions can be used to prove that a block is unreachable, or more accurately, that by the time the garbage collector is allowed to run, this block will be unreachable. IrisFit provides special treatment of temporary roots within protected sections and is thereby able to take advantage of protected sections to establish stronger worst-case heap space bounds.

All of our results are mechanized in the Coq proof assistant using the Iris library [Jung et al. 2018] and its dedicated Proof Mode [Krebbers et al. 2018]. Our definitions and proofs are available in electronic form [Moine 2024]. Discounting blank lines and comments, the definition of LambdaFit and of its oblivious semantics occupy roughly 2800LOC; the construction of IrisFit, including the reasoning rules and the core soundness theorem, represent 9200LOC; the definition of the default semantics of LambdaFit and the proof of the safety and liveness theorems take up 4500LOC; and the verification of the case studies represents 6400LOC. In addition to these numbers, we re-use about 3700LOC of proofs from Madiot and Pottier [2022] and from our own previous work [Moine et al. 2023]. We provide tactics that facilitate reasoning with IrisFit and achieve a basic level of automation thanks to the Diaframe library [Mulder et al. 2022].

In future work, we would like to determine whether immutable data structures could be specified and verified in a more pleasant and lightweight manner. At present, IrisFit offers no special support for immutable data structures: every memory block is considered mutable by default, and it is up to the user to exploit the logical tools offered by Iris, such as invariants, to indicate that a memory block is immutable. In this paper, we have done so in the special case of closures: we have been able to describe the behavior of a closure via a *persistent* predicate, while still allowing for its deallocation. We would like to investigate whether this approach can be extended to all immutable data structures. Independently of this question, we would like to apply IrisFit to more ambitious case studies. This includes larger examples as well as subtler concurrent examples. For the latter, Harris's list [Harris 2001] and multi-CAS algorithms such as RDCSS [Harris et al. 2002] seem good candidates.

We would also like to draw upon our experience with IrisFit to investigate automated static analyses of the worst-case heap space complexity of a program in the presence of garbage collection. As far as we know, relatively few such analyses have been presented in the literature [Braberman et al. 2008; Unnikrishnan and Stoller 2009; Albert et al. 2013] and none of them is justified by a machine-checked argument. It would be interesting to justify existing analyses by reduction to the reasoning rules of IrisFit or to draw inspiration from these rules to design new analyses.

REFERENCES

Ole Agesen. 1998. *GC Points in a Threaded Environment.* Technical Report SMLI TR-98-70. Sun Microsystems, Inc.

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2013. Heap space analysis for garbage collected languages. *Science of Computer Programming* 78, 9 (2013), 1427–1448.

Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. ThreadScan: Automatic and Scalable Memory Reclamation. *ACM Trans. Parallel Comput.* 4, 4, Article 18 (May 2018).

Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. 1999. Implementing Jalapeño in Java. In *Object-Oriented Programming Systems, Languages & Applications (OOPSLA).* 314–324.

Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (2005), 399–418.

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press.

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A program logic for resources. *Theoretical Computer Science* 389, 3 (2007), 411–445.

David Aspinall and Martin Hofmann. 2002. Another Type System for In-Place Update. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 2305).* Springer, 36–52.

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011), 1–33.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Springer.

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: a Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. *Journal of Automated Reasoning* 63, 2 (2019), 369–392.

Lars Birkedal and Aleš Bizjak. 2023. Lecture notes on Iris: Higher-order concurrent separation logic. (2023). Unpublished.

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

Wayne D. Blizard. 1990. Negative membership. *Notre Dame Journal of Formal Logic* 31, 3 (1990), 346–368.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distributed Comput.* 37, 1 (1996), 55–69.

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL).* 259–270.

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2694).* Springer, 55–72.

Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. 2008. Parametric prediction of heap memory requirements. In *International Symposium on Memory Management.* 141–150.

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65.

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Programming Language Design and Implementation (PLDI).* 467–478.

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 519–538.

Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. 2008. Analysing memory resource bounds for low-level programs. In *International Symposium on Memory Management.* 151–160.

Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. 2005. Memory Usage Verification for OO Programs. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 3672).* Springer, 70–86.

William R. Cook. 2009. On understanding data abstraction, revisited. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 557–572.

Karl Crary and Stephanie Weirich. 2000. Resource bound certification. In *Principles of Programming Languages (POPL).* 184–198.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231.

Marc Feeley. 1993. Polling Efficiently on Stock Hardware. In *Functional programming languages and computer architecture (FPCA)*. 179–190.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (1992), 235–271.

Jean-Christophe Filliâtre. 2011. Deductive software verification. *Software Tools for Technology Transfer* 13, 5 (2011), 397–403.

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 388–402.

Alejandro Gómez-Londoño and Magnus O. Myreen. 2021. A flat reachability-based measure for CakeML's cost semantics. In *Implementation of Functional Languages (IFL)*. 1–9.

Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? A verified space cost semantics for CakeML programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 204:1–204:29.

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 249–269.

Theodore Hailperin. 1986. Formalization of Boole's Logic. In *Boole's Logic and Probability*. Studies in Logic and the Foundations of Mathematics, Vol. 85. Elsevier, 135–172.

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing*, Dahlia Malkhi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–279.

Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. 2009. Memory Usage Verification Using Hip/Sleek. In *Automated Technology for Verification and Analysis (ATVA) (Lecture Notes in Computer Science, Vol. 5799)*. Springer, 166–181.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012a. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems* 34, 3 (2012), 14:1–14:62.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012b. Resource Aware ML. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 781–786.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*. 359–373.

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science* 32, 6 (2022), 729–759.

Martin Hofmann. 1999. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science (LICS)*. 464–473.

Martin Hofmann. 2000. A type system for bounded space and functional in-place update. *Nordic Journal of Computing* 7, 4 (2000), 258–289.

Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Information and Computation* 183, 1 (2003), 57–85.

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages (POPL)*. 185–197.

Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 3924)*. Springer, 22–37.

Martin Hofmann and Dulma Rodriguez. 2009. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Computer Science Logic (Lecture Notes in Computer Science, Vol. 5771)*. Springer, 317–331.

Martin Hofmann and Dulma Rodriguez. 2013. Automatic Type Inference for Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 593–613.

Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2011. Separation Logic in the Presence of Garbage Collection. In *Logic in Computer Science (LICS)*. 247–256.

Sadiq Jaffer. 2021. OCaml Compiler Pull Request 10462: Add [@poll error] attribute. https://github.com/ocaml/ocaml/pull/10462.

Richard E. Jones and Rafael Dueire Lins. 1996. *Garbage collection – algorithms for automatic dynamic memory management*. Wiley.

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 828–856.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*. 637–650.

David M. Kahn and Jan Hoffmann. 2021. Automatic amortized resource analysis with the quantum physicist's method. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29.

Ioannis T. Kassios and Eleftherios Kritikos. 2013. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 149–168.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 77:1–77:30.

Rich Lander. 2015. Announcing .NET Framework 4.6. https://devblogs.microsoft.com/dotnet/announcing-net-framework-4-6/.

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320.

Xavier Leroy. 2021. The CompCert C compiler. http://compcert.org/.

Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and go: understanding yieldpoint behavior. In *Symposium on Memory Management (ISMM)*. 70–80.

Daniel Loeb. 1992. Sets with a negative number of elements. *Advances in Mathematics* 91, 1 (1992), 64–74.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 275–304.

Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 718–747.

Paul McKenney, Michael Wong, Maged M. Michael, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, Olivier Giroux, David Vernet, and Timur Doumler. 2023. Read-Copy Update (RCU). P2545R4 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2545r4.pdf.

Paul E. McKenney. 2004. *Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels*. Ph.D. Dissertation. Oregon Health & Science University.

Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.* 3, POPL, Article 58 (jan 2019), 31 pages. https://doi.org/10.1145/3290371

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2023. Hazard Pointers for C++26. P2530R3 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2530r3.pdf.

Microsoft. 2024. Documentation of the GC class of the .NET 8.0 framework.

Alexandre Moine. 2024. Will it Fit? Verifying Heap Space Bounds for Concurrent Programs under Garbage Collection with Separation Logic (Artifact).

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 718–747.

J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. 1995. Abstract Models of Memory Management. In *Functional Programming Languages and Computer Architecture (FPCA)*. 66–77.

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *Programming Language Design and Implementation (PLDI)*. 809–824.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 1–27.

Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR) (EPiC Series in Computing, Vol. 57)*. 543–563.

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95.

Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 83:1–83:29.

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Principles of Programming Languages (POPL)*. 297–302.

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 225–252.

John C. Reynolds. 1975. *User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*. Technical Report 1278. Carnegie Mellon University.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 113:1–113:30.

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 283–311.

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Theoretical Aspects of Computing (ICTAC) (Lecture Notes in Computer Science, Vol. 6916)*. Springer, 239–255.

R. Kent Treiber. 1986. Systems programming: Coping with parallelism.

Leena Unnikrishnan and Scott D. Stoller. 2009. Parametric heap usage analysis for functional programs. In *International Symposium on Memory Management*. 139–148.

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90.

Hassler Whitney. 1933. Characteristic Functions and the Algebra of Logic. *Annals of Mathematics* 34, 3 (1933), 405–414.