We present a Separation Logic with space credits for reasoning about heap space in a sequential call-by-value \( \lambda \)-calculus equipped with garbage collection and mutable state. A key challenge is to design sound, modular, lightweight mechanisms for establishing the unreachability of a block. Prior work in this area uses pointed-by assertions to keep track of the predecessors of every block, but is carried out in the setting of an assembly-like programming language. We take up the challenge in the setting of a high-level language, where a key problem is to identify and reason about the memory locations that the garbage collector considers as roots. For this purpose, we propose novel “stackable” assertions, which keep track of the existence of stack-to-heap pointers without explicitly recording their origin. Furthermore, we explain how to reason about closures—concrete heap-allocated data structures that implement the abstract concept of a first-class function. We demonstrate the expressiveness and tractability of our program logic via a range of examples, including recursive functions on linked lists, objects implemented using closures and mutable internal state, recursive functions in continuation-passing style, and three stack implementations that exhibit different space bounds. These last three examples illustrate reasoning about the reachability of the items stored in a container as well as amortized reasoning about space. All of our results are proved in Coq on top of Iris.

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

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

ACM Reference Format:
https://doi.org/10.1145/3571218

1 INTRODUCTION

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; Jung et al. 2018; O’Hearn 2019] are examples of program logics 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.

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/1-ART25
https://doi.org/10.1145/3571218

Beyond safety and functional correctness, it may be desirable to establish bounds on resource consumption, that is, proving that the resource requirements of a program 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.

This paper is concerned with bounding heap space usage in a garbage-collected language. A fair amount of prior work has focused on establishing bounds on resource consumption. Let us start by reviewing this prior work and explain what makes garbage-collected heap space an especially challenging resource to reason about.

Reasoning about the use of a resource requires a "model" that tells when this resource is consumed or produced, and how much of it is consumed or produced. Such a model is usually an abstraction of some physical reality. For example, to obtain an asymptotic time bound, one can posit that every elementary instruction consumes one unit of time.\(^1\) To obtain an asymptotic bound on stack space, one can posit that every (non-tail) function call consumes one unit of stack space, which is recovered when the function returns.\(^2\) To derive a bound on heap space, when the language has an explicit deallocation instruction, one can posit that an allocation instruction consumes the requested amount of space and that a deallocation instruction recovers the space occupied by the heap block that is about to be deallocated. In all three cases, it is evident in the program where the resource of interest is consumed or produced. In such settings, reasoning about resource consumption can be reduced to reasoning about safety. Indeed, 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 of interest 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. Separation Logic, extended with time credits, has been used to reason about asymptotic amortized time complexity [Atkey 2011; Charguéraud and Pottier 2017; Haslbeck and Lammich 2021; Haslbeck and Nipkow 2018; Mével et al. 2019].

In order to reason about heap space in the presence of explicit allocation and deallocation instructions, traditional Separation Logic [2002] can be extended with space credits. 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

\(^1\)Predicting physical execution time requires access to a compiled version of the program and an accurate model of the processor; see, e.g., Amadio et al. [2014].

\(^2\)Computing a concrete bound, expressed in memory words, requires knowing the size of each stack frame [Amadio et al. 2014; Carbonneaux et al. 2014; Gómez-Londoño et al. 2020].

this idea: LFPL has explicit allocation and deallocation, which consume and produce values of a linear type, written \(\odot\), whose inhabitants behave very much like space credits.

How can one reason about heap space in the presence of garbage collection? In such a setting, there is no explicit 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 about unreachability.

Madiot and Pottier [2022] make a first step towards addressing this problem. They propose a Separation Logic extended with several concepts. To keep track of free space, they use space credits. To enable modular reasoning about unreachability, they use pointed-by assertions [Kassios and Kritikos 2013], which record the predecessors of a memory block. In the absence of a memory deallocation instruction, they view deallocation as a logical operation. It is up to the person who verifies the program to decide at which program points this operation must be used and which memory blocks must thus be logically deallocated. Of course, the GC may physically deallocate a block before or after the point where the user chooses to logically deallocate this block. To account for this fact, Madiot and Pottier introduce a distinction between the physical heap, which the GC manages, and the logical heap, which the programmer (or the user of the program logic) keeps in mind and manages. The physical and logical heaps remain closely related: they must agree on their reachable parts. To ensure that this is the case, a memory block can be logically deallocated only if it is unreachable. However, unreachability is not a local concept. To allow modular reasoning, Madiot and Pottier rephrase this proof obligation in terms of local concepts: an object can be logically deallocated if it has no predecessors and is not a root.

The previous paragraph raises a key question: what is a root? How can this concept be modeled in an operational semantics? How can it be reflected in a program logic? To answer these questions in a simple way, Madiot and Pottier [2022] adopt a nonstandard low-level calculus, SpaceLang, whose design is intended to make the identification of roots trivial. In SpaceLang, a variable denotes the address of a stack cell. Stack cells must be explicitly allocated and deallocated in a well-bracketed manner. At all times, the roots are exactly the stack cells. This approach is conceptually simple, but imposes a low-level programming style on the end user. Because of the pervasive use of stack cells, the language resembles assembly language, and its reasoning rules include many premises that describe stack cells. Furthermore, a stack cell is regarded as a root as long as it exists; if one would like its content to be eligible for collection, one must artificially overwrite it with a unit value. Finally, SpaceLang does not have closures, and, due to the complications created by stack cells, it is not clear how closures can be encoded on top of SpaceLang.

In this paper, we propose to reason directly about a standard, high-level \(\lambda\)-calculus equipped with mutable heap-allocated records, heap-allocated closures, and garbage collection. The notion of root is defined in this language by the standard “free variable rule” [Felleisen and Hieb 1992; Morrisett et al. 1995], which we explain in the next section (§2). For this calculus, we develop a program logic that allows modular reasoning about heap space. Our contributions are the following:

- We present the first Separation Logic for reasoning about heap space in a high-level sequential language equipped with a garbage collector that obeys the free variable rule (§2, §3). Our language and reasoning rules are higher-level and more lightweight than those found in previous work [Madiot and Pottier 2022].
- We introduce a novel assertion, Stackable, which allows keeping track of roots in a modular way (§4). A fractional Stackable assertion is a permission to make a memory location a root. A full Stackable assertion can be exploited to prove that a memory location is not a root.
• We introduce two mechanisms (§5) that help reduce the number of Stackable assertions that must be manipulated during a proof.

• We generalize Madiot and Pottier’s pointed-by assertions [2022] to enable more flexible and lightweight reasoning about the deletion of an edge in the heap. To do so, we introduce possibly-null fractions and signed multisets of predecessors (§4).

• We propose an assertion that describes a closure, as well as reasoning rules for closure construction and closure invocation (§7). The description of a closure captures three aspects of it: its functional behavior, its size, and the pointers to other objects that this closure holds.

• We present a formalization and soundness proof for our logic (§6). It is built inside Coq on top of Iris [Jung et al. 2018]; see [Moine et al. 2022b].

• We illustrate our logic via a collection of examples (§8), including operations on linked lists, a “counter” object with mutable internal state, a recursive function in continuation-passing style, and three stack implementations that exhibit different space bounds.

2 DEALING WITH ROOTS
In this section, we propose a more detailed discussion of our treatment of roots. This concept plays a central role at two distinct levels. First, at the level of the operational semantics, the definition of roots determines which objects are unreachable, that is, which objects can be reclaimed by the garbage collector. Therefore, it determines the space usage of a program: in other words, it defines the cost model that serves as our ground truth. Second, at the level of the program logic, we need mechanisms to keep track of which memory locations may be roots or definitely are not roots.

2.1 The Free Variable Rule
How can the concept of a root be reflected in a small-step, substitution-based operational semantics? A commonly agreed-upon answer is given by the free variable rule (FVR) [Felleisen and Hieb 1992; Morrisett et al. 1995]. Technically, this rule states that a root is a memory location $t$ such that $t$ occurs in the term that is undergoing reduction. In slightly more informal words, $t$ is a root if and only if it appears possible that $t$ might be used in the future, based on the existence of a path from the current program point to a program point where $t$ is used. The FVR represents a conservative approximation of the locations that will be accessed in the future: indeed, depending on which branches are taken, it may turn out that $t$ is in fact never accessed.

Our starting point is an operational semantics where the FVR is built in. We propose a program logic that is sound with respect to this semantics, and we use this logic to establish worst-case space complexity bounds. To obtain a binary program that respects the complexity bounds established using our logic, one needs a compiler (and runtime system) that respect the FVR. As far as we know, many real-world implementations of garbage-collected languages, such as OCaml, SML, Scala, Java, and many more, are meant to respect the FVR. Unfortunately, this intention is often undocumented. A prominent example of a compiler that does respect the FVR is the CakeML verified compiler. Gómez-Londoño et al. [2020] prove that CakeML respects a cost model that is defined at the level of the intermediate language DataLang. 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. If our program logic was adapted to DataLang, one could establish a space complexity bound about a source CakeML program and automatically obtain a guarantee about the compiled program.

2.2 Visible and Invisible Roots
One may wonder why the FVR is so named, since its statement does not contain the word “variable”. The answer lies in the gap between the programmer’s point of view and the semantic point of
let rec rev_append(xs, ys) = if is_nil(xs) then ys else let x = head(xs) in let xs' = tail(xs) in let ys' = cons(x, ys) in rev_append(xs', ys')

Fig. 1. An implementation of linked list reversal

view. A programmer may like to think that the roots are variables. When the programmer focuses on a certain program point, corresponding to a subterm $t$, a variable $x$ that occurs free in $t$ can be regarded by the programmer as a root at this program point—whence the name of the “free variable rule”. In contrast, in the operational semantics, there are no variables: they are substituted away and replaced with closed values. Thus, in the operational semantics and in our reasoning rules (§4), the roots are memory locations. When we write that “the address $x$ is a root” at a certain program point, we mean that, once this program point is reached, the memory location with which the variable $x$ has been replaced is a root.

Let us illustrate reasoning about roots via the example of the function rev_append (Figure 1). This function expects two linked lists and returns a linked list. A call to $\text{rev_append}(xs, ys)$ returns a list whose elements are the elements of $xs$ in reverse order followed with the elements of $ys$. This code is expressed in an untyped language using ML syntax. For simplicity, we do not use pattern matching; instead, we use the auxiliary functions $\text{is-nil}$, $\text{head}$, $\text{tail}$, and $\text{cons}$, whose definitions are omitted. A linked list is represented as a heap block whose first field holds the integer tag 0 or 1. If the tag is 0, then there are no more fields; if the tag is 1, then there are two more fields, holding the head and tail of the list.

We now wish to explain which locations are roots, at each program point in rev_append, according to the FVR. Before doing so, however, we must point out that, when one reasons about rev_append in isolation, its calling context is unknown. By inspecting the code of this function, one can tell that certain memory locations are roots at certain points; we refer to these as the visible roots. However, in addition, every caller along the unknown call chain may have retained certain memory locations. One can think of them as locations that appear “in the stack”. From a semantic point of view, these locations occur in the evaluation context, so, according to the FVR, they are also roots. We refer to them as the invisible roots. The set of all roots is the union of the sets of visible roots and invisible roots. These sets may overlap.

At the entry point of rev_append (at the beginning of line 2), the locations $xs$ and $ys$ are visible roots, because the variables $xs$ and $ys$ occur free in the code that remains to be executed (that is, the whole function body). Upon entering the else branch, on line 3, $xs$ and $ys$ are still roots. At the beginning of line 5, after reading the “head” and “tail” fields of the first list cell, two more variables (namely, $x$ and $xs'$) are visible roots, but $xs$ is no longer one, as it does not occur on lines 5–6. A somewhat subtle phenomenon takes place at this point: the location $xs$ may or may not be an invisible root. If it is not an invisible root, which means that no caller has retained the address of the list $xs$, then this address is not a root at all, which means that the first list cell can be reclaimed at this program point by the GC. Otherwise, this cell cannot be reclaimed. On line 5, a fresh cell, named $ys'$, is allocated. At the beginning of line 6, $ys$ is no longer a visible root, but $ys'$ is one. The location $ys$ remains reachable via $ys'$, thus the list $ys$ cannot be deallocated. Finally, on line 6, a tail-recursive call is made. The locations $xs'$ and $ys'$ cease to be roots for this instance of rev_append, but immediately become roots for the new instance of rev_append.
What is the (heap) space complexity of rev_append? Two distinct answers can be given. On the one hand, without any assumption about the calling context, one can state that the space complexity is linear in the length of the list xs. This is due to the allocation of a new cell at line 5. On the other hand, under the assumption that the address xs is not retained by the calling context (that is, xs is not an invisible root), rev_append runs in constant heap space. Indeed, in that case, the cost of allocating a new cell at line 5 can be compensated by deallocating the cell xs, which is no longer a root, also at line 5. There is no guarantee that the GC will deallocate the cell xs at this point, but it can do so. Because rev_append is tail-recursive, it runs in constant stack space, too, but that is another story. In this paper, we are not concerned with stack space usage.

Our claims about the space complexity of rev_append in the two scenarios described above are expressed by two specifications that we present later on (§8.1).

2.3 Logical Deallocation and its Requirements

In this paper, we propose a Separation Logic with space credits for our language SpaceLambda, an untyped λ-calculus described in §3. Before delving into a detailed presentation of this program logic, let us explain some of its key assertions and mechanisms, via the example of rev_append. Suppose one wishes to verify the claim made earlier (§2.2) that rev_append runs in constant space, under the assumption that xs is not an invisible root. A key step in this proof takes place at the beginning of line 5. There, one must apply a logical deallocation rule to the list cell xs, so as to recover a number of space credits, which can then be used to pay for the allocation of a new cell on the same line. Our logical deallocation rule requires proving that xs has no predecessors (in the heap) and is not a (visible or invisible) root. More specifically, its requirements are as follows:

- As in traditional Separation Logic [Reynolds 2002], a full points-to assertion for the memory block at address xs is required. This assertion is obtained by unfolding the predicate List (§8.1), which one uses to express assumptions about the lists xs and ys.
- As in Madiot and Pottier’s system [2022], a full pointed-by assertion for xs, carrying an empty multiset of predecessors, is required. This assertion too is obtained by unfolding List.
- A proof that xs is not a visible root is required. To establish this fact, one first computes the visible roots at the beginning of line 5: they are the addresses x, xs’, and ys. Then, one must prove that the address xs is not a member of this set. This check is not syntactic: proving that the address xs is distinct from the addresses x, xs’, and ys requires Separation Logic reasoning. For instance, proving that xs and ys are distinct addresses follows from the presence of separate List assertions about xs and ys.
- A proof that xs is not an invisible root is required. In other words, a proof that no direct or indirect caller has retained the address xs is required. Here, the only way of proving this property is to make it an assumption, that is, to let it appear in the precondition of the function rev_append. This assumption is expressed using a Stackable assertion (§4).

Another key step in the proof takes place at the recursive call rev_append(xs’, ys’) on line 6. To prove that this call is permitted, one must prove that the precondition of rev_append, instantiated with the actual parameters xs’ and ys’, is satisfied. Thus, according to the last bullet point above, one must prove that xs’ is not an invisible root. In other words, one must prove that the cell that follows the cell xs in the linked list is not an invisible root. Where might this evidence come from? The most natural answer, we argue, is to bake it in the definition of List: the definition of a valid linked list must state that a cell that is the destination of a link is never an invisible root.

In summary, we have outlined the requirements of our logical deallocation rule and explained the need for a new Separation Logic assertion, which guarantees that a memory location ℓ is not an invisible root. This assertion, written Stackable ℓ 1, is described next.
2.4 Reasoning about Invisible Roots

To understand how one might keep track in Separation Logic of which memory locations are or are not invisible roots, one must first have a clear picture of what this means and at what points in a proof a location becomes or ceases to be an invisible root.

A proof in Separation Logic is 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 \( \text{let } x = t_1 \text{ in } t_2 \). To reason about this construct, one first focuses on the term \( t_1 \), thereby temporarily forgetting the frame \( \text{let } x = [\] 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 \( \text{let } x = [\] 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 of Separation Logic [Jung et al. 2018, §6.2].

An invisible root is a memory location that occurs in the unknown context \( K \). When this context grows and shrinks, the set of invisible roots grows and shrinks as well. More specifically, when the user of the program logic focuses on \( t_1 \), a location \( \ell \) that occurs in the frame \( \text{let } x = [\] in \( t_2 \) (that is, a location that occurs in \( t_2 \)) becomes an invisible root: it is “pushed onto the stack”, so to speak. (This location may have been an invisible root already, prior to this focusing step.) This is undone when this focusing step is reversed: this location is “popped off the stack”.

To keep track in Separation Logic, on a per-location basis, of whether a location may be or definitely is not an invisible root, we propose the following discipline.

- We introduce an assertion \( \text{Stackable } \ell \ p \), where \( p \) is a rational number such that \( 0 < p \leq 1 \). The presence of a fraction allows \( \text{Stackable} \) assertions to be split and joined.
- The assertion \( \text{Stackable } \ell \ 1 \) appears when a fresh memory block is allocated at address \( \ell \), and is eventually consumed when this block is logically deallocated.
- When \( \ell \) is “pushed onto the stack” in an instance of the “bind” rule, an assertion \( \text{Stackable } \ell \ p \) is consumed, where the choice of \( p \) is up to the user; when \( \ell \) is later “popped off the stack”, as part of the same instance of the “bind” rule, this assertion reappears.

One can see that “pushing a location \( \ell \) onto the stack” requires a fractional assertion \( \text{Stackable } \ell \ p \). Thus, this fractional assertion can be intuitively regarded as a permission to push \( \ell \) onto the stack, whence the name \( \text{Stackable} \). Because this assertion is splittable, it allows pushing \( \ell \) onto the stack as many times as one wishes. One can also see intuitively that if the full assertion \( \text{Stackable } \ell \ 1 \) is at hand, then no fraction of it has been consumed, so \( \ell \) currently is not “on the stack”, that is, not an invisible root. Thus, \( \text{Stackable } \ell \ 1 \) serves as a witness that \( \ell \) currently is not an invisible root. It is one of the key novel requirements of our logical deallocation rule.

Madiot and Pottier’s calculus [2022] has explicit stack cells, so their pointed-by assertions record all predecessors, including heap blocks and stack cells. In other words, they record both heap-to-heap and stack-to-heap pointers. In our work, pointed-by assertions concern heap-to-heap pointers, while \( \text{Stackable} \) assertions concern stack-to-heap pointers. Our pointed-by assertions record all heap-to-heap pointers, therefore can be used to prove their absence. Our \( \text{Stackable} \) assertions do not individually keep track of every stack-to-heap pointer (we have removed the need to do so), yet they can be used to prove the absence of stack-to-heap pointers.

3 Syntax & Semantics of SpaceLambda

Our language, SpaceLambda, is an imperative \( \lambda \)-calculus, equipped with a call-by-value substitution-based small-step semantics. Garbage collection is modeled as a reduction step, which can be interleaved in a non-deterministic manner with computational reduction steps.
The syntax of SpaceLambda appears in Figure 2.

![Figure 2. Syntax of SpaceLambda](image)

### 3.1 Closures

Our presentation of closures [Appel 1992; Landin 1964] deserves a careful explanation. To model the space complexity of programs that involve closures, we must somehow reflect the fact that a closure is a heap-allocated object, which has an address, a size, and may hold pointers to other objects. Thus, we cannot just use the standard substitution-based small-step semantics of the $\lambda$-calculus, where a $\lambda$-abstraction is a value. Instead, two approaches come to mind. One approach is to view $\lambda$-abstractions with free variables) and to define $\lambda$-calculus, where the only variables available in the function body are the function’s name $f$ and the formal parameters $\vec{x}$.

Then, we define $\lambda$-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 general $\lambda$-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 approaches yield the same 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.

We equip SpaceLambda with closed functions, which we also refer to as code pointers. We write $\mu_{ptr} f \cdot \vec{x}. t$ for a (recursive) closed function, and write $(v \ \vec{u})_{ptr}$ for the invocation of the code pointer $v$ with arguments $\vec{u}$. Thus, SpaceLambda does not have primitive closures. This allows us to present a program logic for SpaceLambda and to establish the soundness of this logic without worrying about closures (§4–§6). Then, we define closure construction $\mu_{clo} f \cdot \vec{x}. t$ and closure invocation $(f \ \vec{u})_{clo}$ as macros, and we extend our program logic with high-level reasoning rules for closures (§7). This allows reasoning about these macros without expanding them and without even knowing how they are defined. In summary, SpaceLambda can macro-express closures, and our program logic allows reasoning about closures in the same way as if they were primitive constructs.

### 3.2 Syntax

The syntax of SpaceLambda appears in Figure 2. $L$ is an infinite set of memory locations. A value $v$ is a piece of data that fits in one word of memory. A value is the unit value $()$, a natural number $n$, a location $\ell$, or a code pointer $\mu_{ptr} f \cdot \vec{x}. t$, that is, a closed recursive function, where the only variables available in the function body $t$ are the function’s name $f$ and the formal parameters $\vec{x}$.

A block $b$ is either a heap-allocated mutable tuple of values, written $\vec{w}$, or a special deallocated block, written $\emptyset$. Our operational semantics does not recycle memory locations: when a heap block at address $\ell$ is deallocated, the store is updated with a mapping from $\ell$ to $\emptyset$. A heap or store $\sigma$ is a finite map of locations to memory blocks. We write $\emptyset$ for the empty store.

A High-Level Separation Logic for Heap Space under Garbage Collection

Syntax of contexts (Figure 2) dictates a standard left-to-right call-by-value evaluation strategy.

3.4 Reduction under a Context & Garbage Collection

Reduction under a Context. The reduction relation \( t / \sigma \xrightarrow{\text{step}} t' / \sigma' \) allows one head reduction step under an evaluation context \( K \). It is defined by the rules \( \text{StepHead} \) and \( \text{StepCtx} \) in Figure 4. The syntax of contexts (Figure 2) dictates a standard left-to-right call-by-value evaluation strategy.

Our semantics is parameterized by a function, written \( \text{words}(n) \), which returns the size in words of an \( n \)-field memory block. We define the size of a block, written \( \text{size}(\vec{w}) = \text{words}(|\vec{w}|) \) and \( \text{size}(\emptyset) = 0 \). The first equation indicates that the size of a block depends only on the number of its fields; the second equation indicates that a deallocated block occupies no space. Later in the paper (§7, §8), we use \( \text{words}(n) = n \). For example, our representation of linked list cells (§8.1) uses three fields (the tag, head and tail), so a list cell has size 3, a realistic figure. Besides, we define the size of a store as the sum of the sizes of its blocks.

Fig. 4. Reduction under a context, garbage collection, and their combination

Our semantics is parameterized by a function, written \( \text{words}(n) \), which returns the size in words of an \( n \)-field memory block. We define the size of a block, written \( \text{size}(\vec{w}) = \text{words}(|\vec{w}|) \) and \( \text{size}(\emptyset) = 0 \). The first equation indicates that the size of a block depends only on the number of its fields; the second equation indicates that a deallocated block occupies no space. Later in the paper (§7, §8), we use \( \text{words}(n) = n \). For example, our representation of linked list cells (§8.1) uses three fields (the tag, head and tail), so a list cell has size 3, a realistic figure. Besides, we define the size of a store as the sum of the sizes of its blocks.

3.3 Head Reduction

Figure 3 defines the head reduction relation, written \( t / \sigma \rightarrow t' / \sigma' \). Load and store operations require a valid location and a valid offset. In \( \text{HeadLoad} \) and \( \text{HeadStore} \), we write \( \vec{w}(i) = v \) to refer to the \( i \)-th field of a block, and write \( [i := v] \vec{w} \) for a block update. We write \( \sigma(\ell) \) to refer to the contents of a store location, and write \( \sigma(\ell) \) for a store update. In \( \text{HeadAlloc} \), we write \( ()^n \) for a block of \( n \) fields, initialized with unit values.

Following Madiot and Pottier [2022], we parameterize the operational semantics with a limit \( S \) on the size of the heap. An allocation instruction that attempts to exceed this limit cannot take a step: this is expressed by the premise \( \text{size}(\sigma') \leq S \) in \( \text{HeadAlloc} \). Thus, either a garbage collection step can free up a sufficient amount of memory, or the program is stuck.
Garbage Collection. The relation \( R \vdash \sigma \xrightarrow{\text{gc}} \sigma' \), defined by the rule GC in Figure 4, lets a store \( \sigma \) evolve to a store \( \sigma' \) through a GC step that respects a set of roots \( R \). During such a GC step, any location \( \ell \) that is unreachable from every root \( r \in R \) may be deallocated. This is reflected by setting \( \sigma'(\ell) \) to \( \mathbf{f} \). The existence of a path in the store from \( r \) to \( \ell \) is written \( r \xrightarrow{\ast} \sigma \ell \). This is the reflexive and transitive closure of the edge relation defined by the rule \text{Edge} in Figure 4.

Main Reduction Relation. The main reduction relation \( t / \sigma \xrightarrow{\text{step}\cup\text{gc}} t' / \sigma' \), defined by the rules \text{RedGC} and \text{RedStep}, is the union of reduction under a context and garbage collection. In \text{RedGC}, the parameter \( R \), which represents the set of roots that the GC must respect, is instantiated with \( \text{locs}(t) \), the set of all locations that occur in the term \( t \). This expresses the free variable rule (§2.1).

4 A PROGRAM LOGIC FOR SPACELAMBDA

In this section, we present the core of our program logic for SpaceLambda. We postpone the treatment of closures to §7. We start by introducing assertions and triples (§4.1). Next, we present the logical deallocation rule, which plays a central role in our work (§4.2). Then, we provide more detail on the ingredients that appear in this rule, including pointed-by assertions (§4.3), a condition of non-membership in the set of visible roots (§4.4), space credits (§4.5), and deallocation witnesses (§4.6). Last, we review the remaining reasoning rules (§4.7).

4.1 Assertions and Triples

We build our Separation Logic on top of Iris [Jung et al. 2018], and reuse its syntax. In particular, we write \( \Phi \) for assertions, \( \sigma \xrightarrow{\mathit{P}} \) for a pure assertion, \( \Phi \rightarrow \Phi' \) for a separating conjunction and \( \Phi \Rightarrow \Phi' \) for a separating implication. We express the logical equivalence of two formulas as \( \Phi \equiv \Phi' \).

Triples take the form \( \{ \Phi \} t \{ \Psi \} \). A postcondition \( \Psi \) is of the form \( \lambda \ell. \Phi' \), where the metavariable \( \ell \), which denotes the value of the term \( t \), is bound in \( \Phi' \). We write \( \{ \Phi \} t \{ \lambda \ell. \Phi' \} \), where the metavariable \( \ell \) denotes a location, as syntactic sugar for \( \{ \Phi \} \{ \lambda \ell. \Phi' \} \). We take the convention that multi-lines assertions are implicitly joined by a separating conjunction.

Iris features ghost state and ghost updates [Jung et al. 2018, §5.4]. In our work, the ghost update modality \( \Phi \Rightarrow_{\nu} \Phi' \) is parameterized with a set \( V \) of visible roots. This parameter is instantiated with \( \text{locs}(t) \) in the consequence rule, as shown below. The frame rule retains its standard form.

\[
\frac{\Phi \Rightarrow_{\text{locs}(t)} \Phi' \quad \{ \Phi' \} t \{ \Psi \} \quad \{ \Phi \} \left\{ \Psi \right\}}{\{ \Phi \} t \{ \Psi \} \quad \text{CONSEQ} \quad \{ \Phi \} t \{ \Psi \} \quad \text{FRAME}}
\]

4.2 Logical Deallocation

A central aspect of our contribution is a novel logical deallocation rule, \text{LogicalFree}. It allows logically deallocating a heap block \( b \) at location \( \ell \) and reclaiming the space occupied by this block. It is a ghost update, parameterized by a set \( V \) of visible roots \( V \) (recall \text{CONSEQ}, §4.1).

\text{LogicalFree}

\[
\ell \xrightarrow{\mathit{1}} b \ast \ell \xleftarrow{\mathit{1}} 1 \ast \mathbf{f} \ast \ell \not\in V \Rightarrow \ell \ast \mathbf{size}(b) \ast \mathbf{†} \ell
\]

Four assertions are consumed by this ghost update. These four requirements have been informally presented already (§2.3); we give more details about some of them in the following. They are a points-to assertion; a pointed-by assertion (§4.3); the proposition \( \ell \not\in V \), which ensures that \( \ell \) is not a visible root (§4.4); and a \text{Stackable} assertion, which ensures that \( \ell \) is not an invisible root (§2.4).

Two assertions are produced by this ghost update, namely the space credits \( \mathbf{size}(b) \) associated with the deallocated block \( b \), and a deallocation witness \( \mathbf{†} \ell \) (§4.6). The above rule deallocates a single block. Following Madiot and Pottier [2022], we provide a more general rule that frees several blocks at once and allows deallocating cyclic structures [Moine et al. 2022a].
4.3 Pointed-By Assertions

A location ℓ is a predecessor of ℓ’ if the block at location ℓ contains the value ℓ’. We use pointed-by assertions to keep track of the predecessors of every location. A pointed-by assertion takes the form ℓ’ ←q L, where L is a multiset of locations and q is a fraction. Pointed-by assertions can be split and joined using the following two rules:

\[
\begin{align*}
v ←_{q_1 + q_2} (L_1 ⊔ L_2) & \quad ⇔ \quad (v ←_{q_1} L_1 * v ←_{q_2} L_2) \quad \text{if } q_1 > 0 \land q_2 > 0 \quad \text{SplitPointedBy} \\
(v ←_{q_1} L_1 * v ←_{q_2} L_2) & \quad ⇔ \quad v ←_{q_1 + q_2} (L_1 ⊔ L_2) \quad \text{JoinPointedBy}
\end{align*}
\]

Our program logic provides a full knowledge property: a full pointed-by assertion ℓ’ ←1 L (with fraction 1) guarantees that the multiset L contains all of the predecessors of ℓ’. Of particular interest is the assertion ℓ’ ←1 0, which guarantees that ℓ’ has no predecessors.

A pointed-by assertion for a location ℓ initially appears when this location is allocated, and disappears forever when this location is deallocated. Pointed-by assertions evolve during “store” operations. For example, consider a store operation that updates the field ℓ[𝑖] and overwrites the value ℓ′, with the value ℓ′′. The reasoning rule for this operation (§4.7) removes ℓ from a pointed-by assertion for ℓ′ and adds ℓ to a pointed-by assertion for ℓ′′.

We generalize Madiot and Pottier’s pointed-by assertions [2022] by allowing possibly-null fractions and predecessors with negative multiplicity. Our pointed-by assertions take the form ℓ’ ←q L where L is a signed multiset and q is a rational number in the closed interval [0, 1]. We impose the following restriction on the pair (q, L): if the fraction q is 0, then no location can have positive multiplicity in L. This generalization allows us to express the assertion ℓ’ ←0 {−ℓ}, which represents a permission to remove ℓ (once) from the predecessors of ℓ’. This assertion enables us to propose a more elementary formulation of the reasoning rule for stores (§4.7).

Signed multisets [Hailperin 1986], also known as generalized sets [Whitney 1933] or hybrid sets [Loeb 1992], are a generalization of multisets: they allow an element to have negative multiplicity. Blizard [1990] offers a survey. A signed multiset is a partial function from elements to ℤ. The disjoint union operation ⊔ 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} ⊔ {−x} is {+x}.

Possibly-null fractions are new. In traditional Separation Logics with fractional permissions [Bornat et al. 2005; Boyland 2003], 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. Therefore, the “full knowledge” property holds. Here, in contrast, the fraction 0 is allowed, so a full pointed-by assertion ℓ’ ←1 L does not exclude the existence of a separate pointed-by assertion with fraction zero, say ℓ’ ←0 L’. Still, thanks to our requirement that no location can have positive multiplicity in L’, the “full knowledge” property holds: the multiset L that appears in ℓ’ ←1 L remains a sound over-approximation of the true multiset of predecessors of ℓ’. In particular, as desired, the assertion ℓ’ ←1 0 does guarantee that ℓ’ has no predecessors.

We extend pointed-by assertions to arbitrary values and introduce an assertion of the form v ←q L. If v is a location ℓ’, then this assertion is defined as ℓ’ ←q L. Otherwise, it is defined as ⊤True. Likewise, we extend the Stackable assertion to arbitrary values and define Stackable v p. Finally, we write ℓ’ ←q L as a short-hand for the assertion ⊤q > 0 ⇒ ℓ’ ←q L. This notation is used for instance in the reasoning rule Store (§4.7).

4.4 Reasoning about Visible Roots

The rule LOGICALFREE requires ℓ \not∈ V. As explained earlier (§4.1), the set V is instantiated by the consequence rule with locs(i). Thus, a user of our logic is expected to fulfill a proof obligation of the form ℓ \not∈ locs(i). The term i is known: it is the subterm on which the user is focusing. The set
The assertion 

As pointed out earlier (§2.3), this proof obligation is not a syntactic check: the user must prove that the metavariables \( \ell \) and \( \ell' \) denote distinct heap addresses. Fortunately, in practice, proving that two locations are distinct is usually straightforward. It is typically done by exploiting the exclusivity of points-to or pointed-by assertions: \( \ell \mapsto b \iff b' \text{ entails } \ell \neq \ell' \), and, likewise, \( \ell \leftarrow L \iff \ell' \leftarrow 0_q L' \text{ entails } \ell \neq \ell' \). When the user applies \texttt{LogicalFree}, the assertions \( \ell \mapsto b \) and \( \ell \leftarrow \emptyset \) are at hand already. Thus, to prove \( \ell \neq \ell' \), it suffices to exhibit either \( \ell' \mapsto b' \), or \( \ell' \leftarrow 0_q L' \).

In practice, such assertions are usually available in the precondition at hand, because they are likely to be required to reason about the term \( t \).

4.5 Space Credits

Following Madiot and Pottier [2022, §3.2], we use space credits to reason about space consumption. The assertion \( \circ n \) asserts that there exist \( n \) memory words that are free or can be freed by the GC. Furthermore, it asserts the unique ownership of this available space. Space credits can be split and joined, via the rule: \( \circ(n_1 + n_2) \equiv \circ n_1 \circ n_2 \). One can forge zero credits, via the rule: \( \text{true} \iff \emptyset \).

In the present paper, we extend space credits from natural numbers to rational numbers: we let \( n \) range over the non-negative rational numbers. Of course, a physical word of memory cannot be split, so the total number of space credits in existence remains an integral number, and so do the numbers of credits involved in allocations and deallocations. Still, rational numbers appear essential in certain amortized complexity analyses, as illustrated by the example of chunked stacks (§8.4).

4.6 Deallocation Witnesses

The assertion \( \dagger \ell \) [Madiot and Pottier 2022] indicates that the location \( \ell \) has been deallocated. It is persistent: once it holds, it holds forever. It serves as a permission to remove an occurrence of \( \ell \) from a predecessor multiset—deallocated predecessors need not be recorded. The \texttt{Cleanup} rule, shown below, expresses this possibility. This rule was introduced by Madiot and Pottier [2022].

\[
\dagger \ell \triangleright \ell' \leftarrow q (L \cup \{+\ell\}) \implies_V \ell' \leftarrow q L \quad \texttt{Cleanup}
\]

4.7 Reasoning Rules for Terms

Figure 5 gives the syntax-directed reasoning rules of our logic. The rules \texttt{BinOp}, \texttt{IfTrue}, \texttt{IfFalse}, \texttt{LetVal}, and \texttt{CallPtr} are standard. The “later” modality \( \triangleright \) [Jung et al. 2018, §5.5] in the premises indicates that the term in the premise is obtained from the term in the conclusion by performing a reduction step. A reader who is unfamiliar with Iris may safely ignore this aspect.

\texttt{Alloc} generalizes the allocation rule of Separation Logic in two ways. First, its precondition requires enough space credits to pay for the space occupied by the new block; they are consumed. Second, in addition to a points-to assertion for the new block, its postcondition contains a pointed-by assertion and a \texttt{Stackable} assertion. These assertions indicate that there are no pointers from the heap or the stack to the new block.

\texttt{Load} is the standard rule of Separation Logic. Compared with Madiot and Pottier’s rule, which exhibits five preconditions and five postconditions, our rule is significantly simpler. Getting rid of mutable stack cells is what enables this simplification.

Our \texttt{Store} rule is slightly more complex than the standard rule of Separation Logic. Like the standard rule, it requires a full points-to assertion \( \ell \mapsto_1 \bar{w} \) and produces an updated assertion \( \ell \mapsto_1 \bar{w} \) [\( i := v' \bar{w} \)]. In addition, it performs bookkeeping of predecessor multisets, so as to reflect the fact that the value \( v \) that was previously stored in the field \( \ell[i] \) is replaced with \( v' \). First, to reflect the creation of an edge from \( \ell \) to the value \( v' \), an assertion of the form \( v' \leftarrow 0_q L \) is changed to \( v' \leftarrow 0_q L \cup \{+\ell\} \).
with a value $p$. The choice of the fraction $v$ for some map $M$ forcibly frame out, for each location in $\text{locs}(K)$, a Stackable assertion. This is expressed as follows: for some map $M$ of the locations in $\text{locs}(K)$ to fractions, the rule forcibly frames out the assertion $\text{Stackables } M$, which is defined as an iterated separating conjunction of Stackable assertions:

$$
\text{Stackables } M = \bigstar_{(t, p) \in M} \text{Stackable } t \ p.
$$

Here, $q$ must be a positive fraction. The multiset $L$ could be constrained in this rule to be the empty multiset, without loss of expressiveness. Second, to reflect the deletion of an edge from $\ell$ to the value $v$, the assertion $v \leftarrow \ell \ {−\ell}$ appears in the postcondition. As explained earlier (§4.3), this assertion is a permission to remove $\ell$ once from a multiset of predecessors of $v$. Thanks to it, the treatment of edge deletion in our system is more lightweight than Madiot and Pottier’s: their rule requires the assertion $v \leftarrow p \ L$ and changes it to $v \leftarrow p \ L \ \{\ell\}$. It is also more permissive: a pointed-by assertion for $v$ is not required in order to destroy a pointer to $v$.

To explain our $\text{BIND}$ rule, let us begin with a special case. Suppose we wish to reason about the term $x = t_1$ in $t_2$ and suppose $\text{locs}(t_2)$ is a singleton set $\{\ell\}$. We would like to first establish a triple about $t_1$, then establish another triple about $t_2$, where the variable $x$ has been replaced with a value $v$ that stands for the result of $t_1$. A key novel aspect of our rule, which was informally explained earlier (§2.4), is that the assertion $\text{Stackable } t \ p$ is required in the beginning, taken away from the user while reasoning about $t_1$, and given back to the user once she is ready to reason about $t_2$. In other words, it is forcibly framed out while reasoning about $t_1$. This is expressed by the two occurrences of $\text{Stackable } t \ p$ in the following rule:

$$
\text{Particular Case of BIND}
$$

The choice of the fraction $p$ is up to the user. Our $\text{BIND}$ rule (Figure 5) generalizes this idea to an arbitrary evaluation context $K$, in which an arbitrary number of locations may occur. The idea is to forcibly frame out, for each location in $\text{locs}(K)$, a Stackable assertion. This is expressed as follows: for some map $M$ of the locations in $\text{locs}(K)$ to fractions, the rule forcibly frames out the assertion $\text{Stackables } M$, which is defined as an iterated separating conjunction of Stackable assertions:
5 WORKING WITH STACKABLE ASSERTIONS

Each application of the BIND rule requires a number of Stackable assertions to be extracted from the current precondition and framed out. In what follows, we present two simple extensions of our program logic that help tame the number of Stackable assertions that must be forcibly framed out in this way. The first mechanism, triples with souvenir (§5.1), explicitly keeps track of a set of roots. While establishing a triple with souvenir, logical deallocations cannot be applied to a location in R. In exchange for this restriction, a relaxed "bind" rule can be used, which does not forcibly frame Stackable assertions for the locations in R. Furthermore, we use the set R to reduce the proof obligations generated by the LOGICALFREE rule (§5.2). The second mechanism, the NoFree mode (§5.3), relies on a roughly similar restriction, applied to all locations. It is a degraded mode where logical deallocations are forbidden and no Stackable assertions at all are forcibly framed.

5.1 Triples with Souvenir

A triple with souvenir takes the form ⟨R⟩ {Φ} t {Ψ}. The new parameter R denotes a set of locations for which a Stackable assertion has been forcibly framed out already, higher up in the Separation Logic proof tree, in an enclosing application of the “bind” rule. This set can be interpreted as a souvenir (a remembrance) of framed Stackable assertions. The following equivalence explains triples with souvenir in terms of ordinary triples. M is a map of locations to fractions; the condition R ⊆ dom(M) requires M to cover every location in R.

\[ \langle R \rangle \{ \Phi \} t \{ \Psi \} \equiv (\forall M. \ R \subseteq \text{dom}(M) \implies \{ \Phi \ast \text{Stackables} M \} t \{ \lambda v. \text{Stackable} M \} ) \]

Conversely, a plain triple {Φ} t {Ψ} is equivalent to a triple with an empty souvenir ⟨∅⟩ {Φ} t {Ψ}.

For every reasoning rule in Figure 5, we provide a new rule (not shown) that operates on triples with souvenir in terms of ordinary triples. A working with stackable assertions extends the current souvenir with a location ℓ. This requires framing a Stackable assertion for ℓ. The rule FORGETSOUVENIR shrinks the current souvenir.

\[
\begin{align*}
\text{BindWithSouvenir} & \quad \text{dom}(M) = \text{locs}(K) \setminus R \quad \langle R \cup \text{locs}(K) \rangle \{ \Phi \} t \{ \Psi \} \\
& \quad \forall v. \langle R \rangle \{ \Psi' v \ast \text{Stackables} M \} K[v] \{ \Psi \} \\
& \quad \langle R \rangle \{ \Phi \ast \text{Stackables} M \} K[t] \{ \Psi \} \\
\text{AddSouvenir} & \quad \langle R \cup \{ \ell \} \rangle \{ \Phi \} t \{ \Psi \} \\
& \quad \langle R \rangle \{ \Phi \ast \text{Stackable} \ell \ p \} t \{ \lambda v. \text{Stackable} \ell \ p \} \\
\text{ForgetSouvenir} & \quad R' \subseteq R \quad \langle R' \rangle \{ \Phi \} t \{ \Psi \} \\
\text{ConseqMode} & \quad m = \text{if} (R^2 = \text{NoFree}) \text{then } \bot \text{ else } \top \\
& \quad \Phi \Rightarrow^{m}_{\text{loc}(t)} \Phi' \quad \langle R^2 \rangle \{ \Phi' \} t \{ \Psi \} \\
& \quad \langle R^2 \rangle \{ \Phi \} t \{ \Psi \}
\end{align*}
\]

Fig. 6. Key reasoning rules for triples with souvenir

Fig. 7. Key reasoning rules for the NoFree mode
We exploit triples with souvenir to improve the readability of specifications and of proof obligations and to increase proof automation. Our tactic for reasoning about a function call automatically applies `ADD_SOUVENIR` and `FORGET_SOUVENIR` so as to make the current souvenir match the souvenir that appears in the specification of the function. The implementation of our tactics leverages the Diaframe library [Mulder et al. 2022] for all of our rules except `BIND_WITH_SOUVENIR`. For the latter, we define a custom tactic that computes the set of locations `locs(K) \ R`, then attempts to automatically gather, in the precondition, the required `Stackable` assertions.

### 5.2 Reasoning about Visible Roots via Stackable Assertions

Earlier (§2.3, §4.4), we have explained that the logical deallocation rule `LOGICAL_FREE` carries the premise `ℓ \notin V`, which requires the user to show that the location `ℓ` is not a visible root. When this rule is applied, `V` is instantiated with `locs(t)`, where `t` is the subterm under focus; and `ℓ` is the location that the user wishes to deallocate, so the user must exhibit the assertion `Stackable ℓ 1`. Therefore, for every location `ℓ'`, if an assertion `Stackable ℓ' p` is also at hand, then `ℓ \neq ℓ'` follows. Thus, by exploiting the `Stackable` assertions at hand, and by exploiting the souvenir `R` (which implicitly stands for a conjunction of `Stackable` assertions), the proof obligation `ℓ \notin V` can often be automatically met, or at least reduced to a weaker obligation `ℓ \notin V'`, where `V'` is a subset of `V`. This idea is expressed by the following two rules:

\[
\begin{align*}
\text{CONSEQ_WITH_SOUVENIR} & : \quad \Phi \Rightarrow locs(t) \setminus R \quad \Phi' \\
& \quad \langle R \rangle \{ \Phi' \} t \{ \Psi \} \\
& \quad \langle R \rangle \{ \Phi \} t \{ \Psi \} \\
\text{UPDATE_WITH_SOUVENIR} & : \quad \Phi \Rightarrow V \setminus \{ t' \} \quad \Phi' \\
& \quad \Phi \ast Stackable t' p \Rightarrow V \quad \Phi' \ast Stackable t' p \\
\end{align*}
\]

`CONSEQ_WITH_SOUVENIR` is a variant of `CONSEQ`. When the user would like to deallocate a location `ℓ`, this rule automatically proves `ℓ \notin R`, so the proof obligation `ℓ \notin locs(t)` is replaced with `ℓ \notin locs(t) \setminus R`. This proof obligation can be further weakened using the `UPDATE_WITH_SOUVENIR` rule. This rule exploits the presence of the assertion `Stackable ℓ' p` to automatically remove `ℓ'` from the set `V` of roots. Thus, the proof obligation `ℓ \notin V` in an application of the logical deallocation rule `LOGICAL_FREE` is reduced to `ℓ \notin V \setminus \{ ℓ' \}`.

### 5.3 Triples in NoFree mode

Inside the proof of a triple with souvenir `⟨ R \{ \Phi \} t \{ \Psi \} `, none of the locations in `R` can be logically deallocated, since a `Stackable` assertion for each of these locations is implicitly present in the postcondition. Thus, a triple with souvenir can be viewed as a triple that is established in a restricted mode where the locations in `R` cannot be deallocated. We now propose a more radical variant of this idea and introduce an even more restricted (yet still useful) mode where the user cannot deallocate any location. In exchange, she is rewarded with a simplified "bind" rule that does not require framing out any `Stackable` assertion. We refer to this mode as the NoFree mode.

To this end, we introduce a generalized triple `⟨ R' \{ \Phi \} t \{ \Psi \} ` where `R'` is either a set `R` of locations or the special token NoFree. Its definition is found in the extended version of this paper [Moine et al. 2022a]. Again, for each of the reasoning rules in Figure 5, one can establish a generalized rule, which is polymorphic in `R'`. The rules in Figure 6 remain valid. They can be applied when `R'` is a set of locations `R`, but not when `R'` is NoFree. In addition, we establish two new reasoning rules, presented in Figure 7.

The rule `BIND_NO_FREE` is an alternative to `BIND_WITH_SOUVENIR`. It enters NoFree mode while reasoning about the subterm `t`. By instantiating `R'` with NoFree, it can also be used when one is already in NoFree mode. This rule is just as easy to use as the traditional "bind" rule: it does not require any `Stackable` assertions.
The rule $\text{ConseqMode}$ is a generalized formulation of the consequence rule $\text{Conseq}$ (§4.1). In short, this rule simply forbids logical deallocation in $\text{NoFree}$ mode. In greater detail, we parameterize the ghost update modality $\Rightarrow^\psi$ with a mode $m$ that is either $\bot$ (deallocation forbidden) or $\top$ (deallocation permitted). The modality $\Rightarrow^\psi$ that appears in the rule $\text{LogicalFree}$ coincides with $\Rightarrow^\top$, so deallocation requires the mode $\top$. The first premise of $\text{ConseqMode}$ in Figure 7 selects a suitable mode, depending on $R'$, so as to forbid deallocation in $\text{NoFree}$ mode.

Although “triples with souvenir” and “NoFree mode” may appear to be based on roughly similar ideas, they are quite different. Triples with souvenir are defined in terms of ordinary triples and $\text{Stackable}$ assertions; thus, during a proof, one can switch back and forth between ordinary triples and triples with souvenir ($\text{AddSouvenir}$, $\text{ForgetSouvenir}$). In contrast, NoFree mode is more restricted: once one switches to NoFree mode via $\text{BindNoFree}$, there is no way of escaping it. For this reason, we tend to use NoFree mode near the leaves of proof trees.

6 SOUNDNESS

A configuration $t_1 / \sigma_1$ is $\text{final}$ if the term $t_1$ is a value. A configuration $t_1 / \sigma_1$ is $\text{reducible}$ if after one step of garbage collection it can take a proper reduction step: that is, if there exist two stores $\sigma'_1$ and $\sigma_2$ and a term $t_2$ such that $\text{loc}(t_1) \vdash \sigma_1 \xrightarrow{\text{gstep}} \sigma'_1$ and $t_1 / \sigma'_1 \xrightarrow{\text{step}} t_2 / \sigma_2$ hold. A configuration is $\text{stuck}$ if it is neither final nor reducible.

A program $t$ is $\text{safe}$ if $t / 0 \xrightarrow{\text{gstep}} t_1 / \sigma_1$ implies that the configuration $t_1 / \sigma_1$ is either final or reducible—therefore not stuck.

In our setting, the notion of a stuck configuration is more subtle than usual. Our operational semantics includes garbage collection steps, which may reduce the size of the heap. Furthermore, it is parameterized with $S$, the size of the heap ($§3$). An allocation step that exceeds this limit is not permitted. Thus, a program is stuck if, no matter how much memory the GC is able to reclaim, it cannot avoid growing the heap beyond $S$. In other words, a program is stuck if its live heap size is about to exceed $S$. In the contrapositive form, if a program is safe, then its live heap size never exceeds $S$.

Our soundness theorem states that if a program can be verified, using our program logic, under an allowance of $S$ space credits, then this program is safe.

**Theorem 6.1 (Soundness).** If $\{\exists S \mid t \{\lambda_·, \top \text{True}!\}\}$ holds, then $t$ is safe.

Therefore, if a program can be verified under $S$ space credits, then its live heap size never exceeds $S$. This result holds for every $S$. Thus, the space bounds that are established via our program logic are indeed correct.

The proof of Theorem 6.1 is found in our Coq development [Moine et al. 2022b], and a high-level presentation of the key definitions and lemmas is given in the extended version of this paper [Moine et al. 2022a]. Here, we summarize the key novel ingredients of our proof, compared to the soundness proof of a standard Iris-based program logic.

In Iris-based program logics, the triple is usually defined in terms of a “weakest precondition" modality: the assertion $\text{wp} t \Psi$ means that it is safe to execute $t$ and that, if this execution produces a value $\nu$, then $\Psi \nu$ holds. The definition of $\text{wp}$ refers to the small-step semantics of the language and to a $\text{central invariant}$ (also known as the $\text{state interpretation invariant}$) which relates the physical state and the ghost state. Our definition of $\text{wp}$, which appears in Figure 8, follows this pattern. One important aspect of this definition is that it does not refer to the standard semantics of SpaceLambda, which was defined in Figures 3 and 4. Indeed, the structure of the definitions in Figures 3 and 4 does not allow a GC step to take place under an evaluation context. This is inconvenient, as we want $\text{wp}$ to allow reasoning independently of the evaluation context, that is, to admit a “bind" rule.

Therefore, our definition of $\text{wp}$ refers to an alternative small-step semantics, whose judgment takes the form $R \vdash t / \sigma \xrightarrow{\text{ctx} \cup (\text{gc}; \text{head})} t' / \sigma'$. This judgment is parameterized with a set of invisible roots $R$, ...
A High-Level Separation Logic for Heap Space under Garbage Collection

\[ \text{interp} m \sigma_1 \kappa \text{locs}(t_1) \Rightarrow \]
\[ \text{match } t_1 \text{ with } \]
\[ | \text{Val } v \Rightarrow \text{interp} m \sigma_1 \kappa \text{locs}(t_1) \ast \Psi v \]
\[ | _\_ \Rightarrow r \text{ reducible } \text{dom}(\kappa) t_1 \sigma_1 \gamma \ast \]
\[ \triangleright (\forall t_2, \sigma_2. r \text{ dom}(\kappa)) t_1 / \sigma_1 \overset{\text{ctx u (gc; head)}}{\Rightarrow} t_2 / \sigma_2 \gamma \Rightarrow (\text{interp} m \sigma_2 \kappa \text{locs}(t_2) \ast \text{wp} m t_2 \Psi)) \]

Fig. 8. Definition of the weakest precondition modality \( \text{wp} \)

and allows a GC step to take place under an evaluation context. We prove that the two semantics are equivalent—a necessary step, since our soundness theorem is stated with respect to the standard semantics.

In the definition of \( \text{wp} \), the set of invisible roots takes the form \( \text{dom}(\kappa) \), where \( \kappa \) is a finite map of memory locations to nonzero fractions. The metavariable \( \kappa \) is universally quantified at the root of the definition, in the same way as the initial store \( \sigma_1 \). This reflects the intuitive idea that the assertion \( \text{wp} m t_1 \Psi \) expresses a property of the term \( t_1 \) that holds whatever the initial store may be and whatever the invisible roots may be. It is nevertheless possible to constrain the store \( \sigma_1 \) via a points-to assertion [Jung et al. 2018, §6.3.2] and, in a similar way, to constrain the map \( \kappa \) via a \textit{Stackable} assertion. We note that our \( \text{wp} \) is parameterized with a Boolean mode \( m \in \{ \bot; \top \} \), where \( \top \) describes the normal mode, and \( \bot \) denotes the \textit{NoFree} mode (§5.3), a restricted mode where logical deallocation is not permitted. Our central invariant \textit{interp} is parameterized not only with a store, as usual, but also by a mode \( m \), a map \( \kappa \) (the invisible roots), and a set of visible roots. More details appear in the extended version of this paper [Moine et al. 2022a].

7 CLOSURES

As explained earlier (§3.1), SpaceLambda does not have primitive closures. Instead, we define \textit{closure construction} \( \mu f. \lambda \vec{x}. t \) and \textit{closure invocation} \( (\ell \vec{u})_{\text{clo}} \) as macros, which expand to sequences of primitive SpaceLambda instructions. We omit the definitions of these macros, which are standard [Appel 1992, Chapter 10]; they can be found in the extended version of this paper [Moine et al. 2022a]. Suffice it to say that we use flat closures, represented as records containing a code pointer and the values captured by the closures, called its \textit{environment}. Our point is precisely that the end user need not know how these macros are defined: indeed, we propose high-level rules that allow reasoning about closures as if they were primitive objects, and publish a high-level cost-model.

Our construction of these reasoning rules is in two layers. First, we introduce a low-level assertion \textit{Closure} \( E f \vec{x} t \ell \), which asserts that, at location \( \ell \) 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 (§7.1, §7.2) and reveal some details of its implementation (§7.3). Second, we define a high-level assertion \textit{Spec} \( nE P \ell \), which describes the behavior of a closure in a more abstract way. It asserts that, at location \( \ell \), 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 type and meaning of \( P \) are explained later on. Although the environment \( E \) does not participate in the description of the behavior of the closure, it remains needed in order to reason about its size and about the pointers that it contains. We give an overview of this high-level API (§7.4), then describe its implementation (§7.5). In practice, only the high-level layer is exposed to the end user; the low-level layer remains internal.
We write $\text{fvcl}(f, \vec{x}, t)$ 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 $\text{fv}(t) \setminus \{f, \vec{x}\}$. The order in which the variables occur in this list is irrelevant but is chosen in a deterministic manner. An environment $E$ is a list of pairs of a value and a nonzero fraction (for use in a pointed-by assertion). The length and order of the list $E$ are intended to match the length and order of the list $\text{fvcl}(f, \vec{x}, t)$. We stress that an environment $E$ is not a runtime object: it is a mathematical object that we use as a parameter of the predicates $\text{Closure}$ and $\text{Spec}$.

### 7.2 Low-Level Closure API

Our low-level reasoning rules for closures, shown in Figure 9, involve the predicate $\text{Closure}$, which describes the layout of a closure in memory. A user views $\text{Closure}$ as an abstract predicate; its definition is given in the next section (§7.3).

The rule $\text{MkCLO}$ specifies a closure construction operation. The term, written $[[\tilde{w}/\tilde{y}] \mu_{\text{clo}} f. \lambda \vec{x}. t]$, is the application of a multi-substitution of some values $\tilde{w}$ for the free variables $\tilde{y}$ of the function $\mu f. \lambda \vec{x}. t$ to the closure construction macro $\mu_{\text{clo}} f. \lambda \vec{x}. t$. 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 $\mu_{\text{clo}} f. \lambda \vec{x}. t$. The values $\tilde{w}$ that appear in this multi-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 $\text{MkCLO}$, an environment $E$ is built by pairing up the values $\tilde{w}$ with nonzero fractions $\tilde{q}$. These fractions, whose choice is up to the user, determine what fractional pointed-by assertion is consumed by the closure for each of these values. Indeed, according to the precondition in $\text{MkCLO}$, for each value $w$ in the list $\tilde{w}$, the closure construction operation consumes $w \leftarrow ^\circ_q 0$. (This notation, introduced at the end of §4.3, requires $q > 0$.) In addition, this operation consumes $1 + |E|$ space credits, reflecting the space needed to store a code pointer and the values $\tilde{w}$ in a flat closure. According to the postcondition in $\text{MkCLO}$, this operation produces the assertion $\text{Closure}$ $E$ $f$ $\vec{x}$ $t$ $t$, which guarantees that there is a well-formed closure at address $t$, as well as a $\text{Stackable}$ assertion and a pointed-by assertion for $t$, which guarantee that we have a unique pointer to this closure.

The rule $\text{CallCLO}$ closely resembles the rule $\text{CallPtr}$ for primitive function calls (Figure 5). One difference is that $\text{CallCLO}$ requires the assertion $\text{Closure}$ $E$ $f$ $\vec{x}$ $t$ $t$, which describes the closure. Another difference is that, whereas a primitive function $\mu_{\text{ptr}} f. \lambda \vec{x}. t$ must be closed, a general function can have a nonempty list of free variables $\tilde{y}$, an alias for $\text{fvcl}(f, \vec{x}, t)$. In the last premise...
of \text{CallClo}, which requires reasoning about the function’s body, the variables \(\vec{y}\) are replaced with the values \(\vec{w}\) captured at closure construction time, which are recorded in the environment \(E\).

The rule \text{FreeClo} allows the logical deallocation of a closure. It closely resembles \text{LogicalFree}. The main difference is that, instead of consuming a points-to assertion, it consumes the abstract assertion \(\text{Closure} E f \vec{x} t \ell\). Furthermore, it releases the pointed-by assertions that were captured at closure construction time.

The postcondition of \text{FreeClo} matches the premise of \text{MkClo}, and the precondition of \text{FreeClo} matches the postcondition of \text{MkClo} (up to the assumption that the closure is not a visible root). Thus, a (physical) closure allocation followed with a (logical) closure deallocation does not alter the set of assertions at hand.

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

### 7.3 Low-Level Closure API: Implementation Details

The \text{Closure} \(E f \vec{x} t \ell\) assertion is defined as follows.

\[
\text{Closure} E f \vec{x} t \ell \triangleq \forall f \not\in \vec{x} \land |E| = |f_{clo}(f, \vec{x}, t)| \forall \ell \mapsto_{\tau} (\text{codeclo}(f, \vec{x}, t) :: \text{map fst } E) \ast \left((v, q) \in E \Rightarrow (v \leftarrow q \{+\ell\}) \right)
\]

This assertion 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 \(\ell\) points to a block of size \(1 + |E|\). Its first field contains the code of the closure, \(\text{codeclo}(f, \vec{x}, t)\), whose definition appears in the extended version of this paper [Moine et al. 2022a]. The other fields contain the values recorded in the environment \(E\). Finally, for every pair \((v, q)\) in \(E\), a pointed-by assertion appears, reflecting the fact that \(v\) is pointed to by the closure.

### 7.4 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 \text{Closure} \(E f \vec{x} t \ell\) reveals the code of the closure. As a result, a user of the \text{Closure} API naturally wishes to hide this information via an existential quantification over this code. This is common enough and difficult enough that we offer a higher-level API where this existential quantification is built in. We introduce the assertion \text{Spec} \(n E P \ell\) (defined in §7.5), where \(n\) is the arity of the function; \(E\) is the environment of the closure; \(P\) describes the behavior of the closure; and \(\ell\) is the location of the closure in memory.

The design of this high-level API is made particularly challenging by two aspects. First, a closure may invoke itself recursively. Thus, we want to assume an instance of \text{Spec} while establishing an instance of \text{Spec}. Second, a closure may self-destruct, that is, logically deallocate itself after it has been invoked. Thus, we want to be able to consume this \text{Spec} assumption, if desired. For presentation purposes, we first present a simplified assertion, \text{Spec}' \(n E P \ell\), which handles closures that are not recursive and do not self-destruct (Figure 10). Then, we show the general case (Figure 11).

\textit{A Simplified Interface.} The \text{Spec'} API is presented in Figure 10. Let us first examine the rule \text{Call-Spec‘}. A call of the form \((\ell \vec{a})_{clo}\) admits a precondition \(\Phi\) and a postcondition \(\Psi\) if the entailment \(\forall u. P \vec{a} u \Rightarrow \{\Phi\} \ast \{\Psi\}\) holds. Intuitively, \(u\) denotes the instantiated function body that was visible in the low-level rule \text{CallClo}; however, this function body is now abstracted away by the universal quantification over \(u\). The predicate \(P\) represents the specification of the function. For example, in
Accordingly, in the postcondition of the call, in the conclusion of (§7.2), the work of reasoning about the function body is shifted from the closure invocation site to cumbersome. Self-parameterization moves the need for a recursive definition from the API user argument, the predicate \( z \), to the closure address \( \bar{w} \). This allows the specification of a closure to refer to the address of the closure: for

\[
\begin{align*}
\text{non-expansive } P & \quad \Rightarrow \quad \forall w. \ P \bar{w} u \rightarrow \{ \Phi \} \ u \ \{ \Psi \} \\
\{ \text{Spec' } n \ E P \ell \ast \Phi \} \ (\ell \bar{w}) \ _\text{clo} \ \{ \lambda v. \ \text{Spec' } n \ E P \ell \ast \Psi v \} & \quad \Rightarrow_V \quad \forall (w, q) \in E \left( \lambda f. \ \text{Spec' } n \ E P \ell \ _\text{Stackable } \ell \ast \ell \_1 \ast \_1 \ 0 \right)
\end{align*}
\]

Fig. 10. Simplified high-level interface for closures: non-recursive, non-self-destructing functions

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: \( \lambda u. \ \forall v. \ P \bar{w} u \rightarrow \{ \Phi \} \ u \ \{ \Psi \} \Rightarrow \forall w. \ P \bar{w} u \rightarrow \{ \Phi \} \ u \ \{ \Psi \} \left( \ell \bar{w} u \rightarrow \{ \text{Spec } n \ E P \ell \ast \Phi \} \ (\ell \bar{w}) \ _\text{clo} \ \{ \lambda v. \ \text{Spec } n \ E P \ell \ast \Psi v \} \right) \)

Let us now consider the rule \text{MkSpec'}. Its first four premises are the same as in \text{MkClo}. In addition, the premises on the second line ensure that \( P \) is a valid description of the behavior of the function body, whose concrete form \( \bar{w} \) is visible. In comparison with the low-level API (§7.2), the work of reasoning about the function body is shifted from the closure invocation site to the closure construction site.

\text{A General Interface.} The \text{Spec} API appears in Figure 11. \text{Spec} generalizes \text{Spec'} in three ways. First, in the last premise of \text{MkSpec}, we make the assumption \text{Spec } n \ E P \ell \ast \Phi \available while reasoning about the body of the function. This assumption is exploited and consumed in the course of this reasoning. Accordingly, in the postcondition of the call, in the conclusion of \text{CallSpec}, we get just \( \Psi \), whereas in \text{CallSpec'} we would automatically get \text{Spec' } n \ E P \ell \ast \Phi \available in addition to \( \Psi \). Thus, with \text{CallSpec}, the existence of the closure is not necessarily preserved through the call. If it is preserved, then this can be expressed via suitable choices of \( P \) and \( \Psi \). Second, we parameterize \( P \) with the location \( \ell \) of the closure. This allows the specification of a closure to refer to the address of the closure: for example, the specification may require a \text{Stackable} assertion for \( \ell \). Third, we parameterize \( P \) over the assertion \text{Spec } n \ E P \ell. As a result, the applications of \( P \) in the premises of \text{MkSpec} and \text{CallSpec} take the form \( P \bar{w} u \rightarrow \{ \text{Spec } n \ E P \ell \} \). Without this form of "self-parameterization", many uses of the \text{Spec} API would require instantiating \( P \) with a recursively-defined predicate, which would be cumbersome. Self-parameterization moves the need for a recursive definition from the API user.

\[\begin{align*}
\text{MkSpec'} \\
\bar{y} = \text{fvclo}(f, \bar{x}, t) \\
E = \text{zip } \bar{w} \bar{q} \\
|\bar{w}| = |\bar{y}| \\
f \notin \bar{x} \\
n = |\bar{x}| \\
\forall \bar{u}. \ \Rightarrow \quad (1 + |E|) \ast \\
\begin{pmatrix} \ast w \leftarrow 0_q \ \emptyset \end{pmatrix} \\
|\bar{w}/\bar{y}| \ (\mu_{\text{clo}} f. \lambda \bar{x}. t) \left\{ \lambda f. \ \text{Spec' } n \ E P \ell \ _\text{Stackable } \ell \ast \ell \_1 \ast \_1 \ 0 \right\}
\end{align*}\]
to the API implementor, thereby making the API significantly easier to use. The use of the “self” parameter is illustrated by the definition of the predicate Counter (§8.3), which appears in the extended version of this paper [Moine et al. 2022a]. Overall, these changes allow the function body to exploit and possibly to consume the assertion Spec n E P ℓ, thereby allowing the closure to recursively invoke itself or to logically deallocate itself.

The condition NonExpansive P in MkSpec asserts that Φ ≜ Φ′ implies P ℓ v u Φ ≜ P ℓ v u Φ′, where ≜ is equivalence down to step-index n [Jung et al. 2018, §5]. Our triples, in particular, are non-expansive in their postcondition: this matters because P is typically instantiated with a triple or a conjunction of triples.

7.5 High-Level Closure API: Implementation Details

The assertion Spec n E P ℓ is defined as follows:

\[ Spec n E P ℓ \triangleq \exists f, x, t, Q. \langle [\bar{x}] = n^\gamma \ast \text{Closure } E f x t \ast \]

let \( \bar{w} = \text{map fst } E \) in let \( \bar{y} = \text{fvclo}(f, x, t) \) in let body \( \bar{v} = [t/f][\bar{w}/\bar{y}][\bar{v}/\bar{x}]t \) in

\[ \triangleright \square(v_\bar{y}. \langle [\bar{y}] = n^\gamma \ast \text{Spec } n E Q t \ast \rightleftharpoons Q t \bar{v} \ast (\text{body } \bar{v}) \ast (\text{Spec } n E Q t)) \ast \]

\[ \triangleright \square(v_\bar{y}. \langle [\bar{y}] = n^\gamma \ast \rightleftharpoons Q t \bar{v} \ast (\text{body } \bar{v}) \ast (\text{Spec } n E P t)) \]

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 \( f, x, \) and \( t \). It is also existentially quantified over a specification predicate \( Q \), which is required to be stronger than \( P \). This internal distinction between \( P \) and \( Q \) allows us to establish a consequence rule (that is, a weakening lemma), which is part of the Spec API, but is not shown in Figure 11.

8 EXAMPLES

We now showcase the expressiveness of our program logic via a series of representative examples. The description of linked lists and the two specifications of \( \text{rev} \_ \text{append} \) (§8.1) complete the opening discussion of the paper (§2.2). This example illustrates a “container” data structure, which holds pointers to the container’s elements. Linked list concatenation in continuation-passing style (§8.2) demonstrates how to reason about (a chain of) one-shot, self-destructing closures. This example also involves a recursive closure. A “counter” object with two methods, implemented by two closures that share a private mutable reference (§8.3), demonstrates how to reason about closures with shared private state. Finally, we present a specification for a “stack” abstract data type (§8.4). This API illustrates the transfers of pointed-by assertions that take place when an element is inserted into or extracted out of a container. We propose three implementations of stacks, which have a common behavior, but different space usage. The first implementation demonstrates a use of our linked lists. The second implementation relies on a mutable array, and demonstrates that if one omits to overwrite an array slot when a value is popped off the stack, then a memory leak appears and the code cannot be verified. The third implementation is a generic construction of a stack as a stack of stacks. It demonstrates modular reasoning as well as an amortized space complexity analysis that exploits rational space credits.

In the examples that follow, we use SpaceLambda’s support for closures (§7) when defining local functions, and we use its primitive code pointers (§3) when defining closed toplevel functions. It would arguably be more elegant to avoid such a mixture and use closures everywhere: after all, a true high-level programming language offers just one kind of functions. However, constructing closures at the top level raises a technical problem that is orthogonal to the topic of this paper, namely the problem of modeling toplevel effectful expressions in Iris. (A closure construction
expression in SpaceLambda is effectful: it allocates memory.) We have carried out a preliminary investigation of this problem, but leave its resolution to future work.

We find that, in practice, pointed-by assertions and Stackable assertions often go hand in hand. This is not surprising, since the former keep track of predecessors in the heap, while the latter keep track of predecessors in the stack. This leads us to introduce \( v \leftarrow_p L \) as sugar for the conjunction \( v \leftarrow_p L \wedge \text{Stackable } v \). We refer to this new assertion as a handle for the value \( v \). In particular, the assertion \( v \leftarrow_1 \emptyset \), which we call a full empty handle, guarantees that there are no pointers to \( v \) from the heap or from the invisible part of the stack. If this assertion is at hand and if, in addition, \( v \) is not a visible root, then \( v \) can be logically deallocated.

### 8.1 Linked Lists and Linked List Reversal

We represent linked lists as blocks whose first field contains a tag. An empty list is encoded as a block of size 1 with tag 0. A list cell is encoded as a block of size 3 with tag 1 and two fields holding the head and tail of the list. The predicate \( \text{List } vps \) \( \) \( xs \) asserts that there is a well-formed linked list at location \( xs \) whose logical model is \( vps \), a mathematical list of pairs of a value \( v \) and a nonzero fraction \( p \). It is defined as follows:

\[
\text{List } vps \ xs \ \triangleq \ \text{match } vps \ \text{with}
\]

\[
\begin{align*}
& | [] \Rightarrow xs \mapsto [0] \\
& | (v, p) :: vps' \Rightarrow \exists xs'. \ xs \mapsto [1; v; xs'] \wedge v \leftarrow_p \{xs\} \wedge xs' \leftarrow_1 \{xs\} \wedge \text{List } vps' \ xs'
\end{align*}
\]

We adopt the convention that when a value \( v \) is inserted into a linked list, a fractional handle \( v \leftarrow_p \emptyset \) is consumed; if \( v \) is later extracted out of the linked list, this handle is returned to the caller. This allows the linked list to internally record that a certain linked list cell is a predecessor of \( v \), without revealing the address of this cell to the user. We use this idiom in the description of containers other than linked lists: see, for instance, our stack API (§8.4).

The definition of \( \text{List } vps \ xs \) has a standard overall structure [Reynolds 2002]. When \( vps \) is empty, this predicate boils down to the points-to assertion \( xs \mapsto [0] \). When \( vps \) is a nonempty list \( (v, p) :: vps' \), the predicate begins with the points-to assertion \( xs \mapsto [1; v; xs'] \), which describes a 3-field cell. Moreover, what is new, it contains the handles \( v \leftarrow_p \{xs\} \) and \( xs' \leftarrow_1 \{xs\} \). As explained above, the first handle records the fact that the cell \( xs \) is a predecessor of the value \( v \). The second handle records the fact that the cell \( xs \) is a predecessor of the next cell, \( xs' \). Because this handle carries the fraction 1, the pointer from \( xs \) to \( xs' \) is a unique pointer: there are no other pointers (from the heap or the stack) to \( xs' \). More generally, there can be no direct pointers from the outside to an internal cell. The ability to express this property is unusual: indeed, via points-to assertions, traditional Separation Logic can express unique ownership, that is, control who may dereference a pointer; however, it cannot express the fact that a pointer is unique.\(^3\) Pointed-by assertions [Kassios and Kritikos 2013; Madiot and Pottier 2022] and our Stackable assertions add this ability, enabling us to express properties about the shape of the heap, in a way that is reminiscent of the literature on ownership types and uniqueness types [Clarke et al. 2013].

We are now ready to present the two specifications of the function \( \text{rev}\_append \) that we mentioned at the beginning of the paper. Recall (§2.2) that this function expects two lists \( xs \) and \( ys \) and returns a list whose elements are the elements of \( xs \) in reverse order followed with the elements of \( ys \). The two specifications appear in Figure 12. The first specification allows the caller to retain the root \( xs \) (a triple with souvenir expresses this) and asserts that \( \text{rev}\_append \) has linear heap space complexity: it requires \( 3 \times |vps| \) space credits. The second specification requires the caller to provide (and give up)

\(^3\)The predicate \( \text{List} \) in traditional Separation Logic does forbid two valid linked lists from sharing a suffix, but does not rule out the existence of a rogue pointer (without any access permission) from the outside into a linked list.
A High-Level Separation Logic for Heap Space under Garbage Collection

25:23

linked list

ys invokes this closure, with a closure for the identity function as a continuation.

Its implementation appears in Figure 13. The main function, append, constructs the concatenation of two linked lists and is written in continuation-passing style (CPS).

8.2 Continuation-Passing Style

a unique pointer to xs and asserts that rev_append has constant heap space complexity. Indeed, in this case, rev_append requires zero space credits because, at each step, one cell of the list xs can be logically freed before one new list cell is allocated.

Both specifications in Figure 12 require the handle ys \leftarrow_1 \emptyset. That is, they require a unique pointer to the list ys. This is necessary because ys becomes a suffix of the list that is returned by rev_append. If the caller was allowed to keep a copy of the pointer ys, then this copy would become a pointer from the outside to an internal cell, a situation which our definition of List forbids.

The two specifications differ slightly in their postconditions. The postcondition of the first specification is more complex because the values contained in the input list xs become shared between the input list xs and the output list zs. We express this by splitting fractions: \frac{1}{2}vps denotes a copy of the list vps where the fraction associated with every value has been halved.

Certain functions, such as rev_append and append, admit several useful and incomparable specifications. In systems that keep track of ownership, whether they are program logics or type systems (such as Rust), it often appears unavoidable to assign several specifications to an operation. Avoiding duplication (of code, specifications, or proofs) remains a challenging problem.

8.2 Continuation-Passing Style

To demonstrate our ability to reason about nontrivial use of closures, we present a function that constructs the concatenation of two linked lists and is written in continuation-passing style (CPS). Its implementation appears in Figure 13. The main function, append, expects two linked lists xs and ys. It first allocates a (recursive) closure aux, described below, which captures ys. Then, it invokes this closure, with a closure for the identity function as a continuation.

The function aux expects two arguments xs and k. If xs is nil, then it applies the closure k to the linked list ys. Otherwise, it allocates a new closure k′, whose purpose is to “cons” the element x in
counter, that is, a pair of closures. A reference is represented as a single-field block, and a pair as a two-field block. The syntactic sugar \{ i, g \} allocates, initializes and returns a pair, represented as a heap block of size 2. The definitions of the auxiliary functions ref, incr and get are not shown.

Fig. 13 presents the code. The toplevel function mk_counter allocates and returns a fresh counter; a closure \( i \) increments the counter; a closure \( g \) gets its current value. Although these closures share an internal state, they can be passed around and invoked independently.

Figure 13 presents the code. The toplevel function mk_counter allocates and returns a fresh counter; a closure \( i \) increments the counter; a closure \( g \) gets its current value. Although these closures share an internal state, they can be passed around and invoked independently.

Like \( \text{rev}_\text{append} \) (§8.1), append admits two specifications presented in Figure 14, which differ in their assumption about \( xs \). If the linked list \( xs \) comes with a full empty handle, then it can be logically deallocated, which pays for the space occupied by the new list that is constructed; otherwise, this space must be paid for. Besides, internally, append needs a certain amount of temporary storage, whose size is linear in the length of the list \( xs \), and which is released when append returns. This temporary storage is described by the space credits that appear both in the precondition and in the postcondition, reflecting a “high water mark”.

The number \( 3 \times |vps| \) that appear in these specifications, where \( |vps| \) denotes the length of the linked list \( xs \), corresponds to the space usage of the linked chain of continuations that is formed in the heap. In the first triple, an additional \( 3 \times |vps| \) credits are needed, because of the allocation of new linked list cells. One credit is used by the identity closure. Another two credits are used by the closure aux. In the second triple, one credit is recovered by deallocating an empty linked list.

The continuations involved in this example are one-shot, i.e., called only once. They are self-destructing continuations: in our proofs, we logically deallocate them as soon as they are invoked.

8.3 Counter Objects

We now present an example of a procedural abstraction [Reynolds 1975], also known as an object: in Cook’s words [2009], “an object is a value exporting a procedural interface to data or behavior”. Our example is a “counter” object, whose internal state is stored in a mutable reference, and whose procedural interface is given by a pair of closures: a closure \( i \) increments the counter; a closure \( g \) gets its current value. Although these closures share an internal state, they can be passed around and invoked independently.

Fig. 14. Specifications for linked list concatenation in continuation-passing style

Fig. 15. An interface for a “Counter” object
allow testing whether a stack is empty and testing whether a stack is full. Finally, the logical deallocation of a stack allows recovering all of the space occupied by the stack, namely the stack’s capacity. This requirement is trivially satisfied if $C = +\infty$. Two additional operations (not shown) allow testing whether a stack is empty and testing whether a stack is full. Finally, the logical deallocation of a stack allows recovering all of the space occupied by the stack, namely $A + B \times |vps|$ space credits, where $|vps|$ is the number of elements of the stack. We emphasize that

![Fig. 16. An interface for possibly-bounded stacks](image)

The specification of the counter (Figure 15) consists of four components: the specification of $mk\_counter$, which returns a pair of closures; the specifications of these closures; and a ghost update that allows the joint logical deallocation of these closures and of the counter’s internal state.

The precondition of $mk\_counter$ states that the creation of a counter requires 7 space credits: one credit for the shared reference $c$, two credits for each of the two closures, and two credits for the pair of closures. Its postcondition indicates that $mk\_counter$ returns a full empty handle on a pair of unique pointers $i$ and $g$. The abstract predicate $Counter\ 0\ i\ g$ describes the current value of the counter (which initially is zero) and asserts that $i$ and $g$ are its “increment” and “get” methods. Its definition appears in the extended version of this paper [Moine et al. 2022a]. Thus, the fact that there is a pair at location $\ell$ is exposed, but the nature of the objects at locations $i$ and $g$ is not revealed. The user may access the pair to obtain the addresses $i$ and $g$. If or when so desired, she may also logically deallocate the pair.

The closure invocations $(i\ [])\clo$ and $(g\ [])\clo$ require the assertion $Counter\ n\ i\ g$. The former updates this assertion to $Counter\ (n + 1)\ i\ g$, reflecting the fact that the internal state of the counter has been changed. The latter leaves this assertion unchanged, and asserts that the result of the invocation is $n$, the current value of the counter.

Deallocating a counter (a logical operation) consumes $Counter\ n\ i\ g$ as well as the handles for $i$ and $g$ and produces 5 space credits. By deallocating the pair of $i$ and $g$, one can recover 2 more credits. These credits add up to 7, which was the cost of allocating a counter in the first place.

### 8.4 Stacks

We verify an unbounded-capacity mutable stack implemented as a linked list, a bounded-capacity stack implemented as an array, and a functor that constructs a stack of stacks. Figure 16 presents a common interface for all our stacks. This interface is parameterized with a capacity $C$, which is either an integer or $+\infty$. It is also parameterized with two constants: $A$ is the number of credits required to allocate an empty stack, and $B$ is the number of credits required by a push operation.

The specifications rely on the abstract predicate $Stack\ vps\ \ell$, which asserts that at address $\ell$ there is a valid stack whose elements are described by the mathematical list $vps$. As in the specification of linked lists (§8.1), $vps$ is a list of pairs of a value and a nonzero fraction. According to these specifications, $create$ consumes $A$ space credits and produces a fresh empty stack; $push$ consumes $B$ space credits and a fractional handle for the value that is inserted into the stack; $pop$ gives up these assertions. In addition, $push$ requires the number of elements in the stack to be less than the stack’s capacity $C$. This requirement is trivially satisfied if $C = +\infty$. Two additional operations (not shown) allow testing whether a stack is empty and testing whether a stack is full. Finally, the logical deallocation of a stack allows recovering all of the space occupied by the stack, namely $A + B \times |vps|$ space credits, where $|vps|$ is the number of elements of the stack. We emphasize that
the deallocation of a stack is an implicit operation at runtime: there is no code for it. Nevertheless, a
deallocation lemma must be included in the stack API and must be established inside the abstraction
boundary of stacks.

Our three implementations of stacks (not shown) differ in their space complexity. Each of them
is verified with respect to a particular instantiation of the parameters $A$, $B$, and $C$.

Our first implementation consists of a mutable reference to a linked list. The reference occupies
1 word, an empty list occupies 1 word, and each list cell occupies 3 words. This stack has unbounded
capacity. Hence, this implementation satisfies our common interface for the parameters $A = 2$, $B = 3$, and $C = +\infty$.

Our second implementation consists of a record where one field holds the logical size of the
stack and one field holds a pointer to an array of fixed capacity $T$. Every unused cell in this array
is filled with a unit value. This implementation satisfies our interface with creation cost $A = T + 2$, insertion cost $B = 0$, and bounded capacity $C = T$.

Our third implementation is generic: it is a functor that expects two implementations of stacks, say
$X$-stacks and $Y$-stacks, and produces a new implementation, say $Z$-stacks. A $Z$-stack is implemented
as a pair made of (1) a nonempty $Y$-stack storing the elements at the top of the stack, and (2) a $X$-stack of full $Y$-stacks, storing all the remaining elements. To simplify the explanations, we
assume that $Y$-stacks are bounded—an assumption that our formalization does not make. Let us
write $X.A$ and $X.B$ and $X.C$ for the space complexity parameters of $X$-stacks, and likewise for
$Y$-stacks. We formally establish that our $Z$-stacks have creation cost $A = X.A + Y.A + 2$, insertion cost $B = Y.B + (Y.A + X.B) / Y.C$, and capacity $C = X.C \times (1 + Y.C)$. The insertion cost is of particular interest. An empty $Y$-stack is allocated and pushed on the $X$-stack only every $Y.C$ push operations
on the $Z$-stack: this explains the fractional cost $(Y.A + X.B) / Y.C$. Obtaining this bound requires
rational space credits and an amortized analysis, which involves defining a suitable potential
function and saving space credits in the definition of $Stack$ for $Z$-stacks.

By applying the functor to our previous two implementations of stacks as arrays and stacks as
linked lists, one obtains a time- and space-efficient implementation of chunked stacks, that is, linked
lists of fixed-capacity arrays.

9 RELATED WORK

Reasoning about Space without a GC. Hofmann [1999, 2003] introduces space credits in the setting
of an affine type system for the $\lambda$-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.

Chin et al. [2008, 2005] 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 deallocation, therefore with no need to reason about unreachability. Reasoning about unreachability is a central challenge in the presence of garbage collection.

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. As far as we are aware, only Madiot and Pottier [2022] have proposed a Separation Logic that allows reasoning about space in the presence of a GC. Their logic concerns a low-level language with explicit stack cells. In contrast, we propose a program logic for a high-level language, a call-by-value \( \lambda \)-calculus with support for closures. Their logic supports concurrency, but they do not propose any examples of concurrent programs with verified space complexity bounds. We do not deal with concurrency.

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 \( \lambda \)-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” [Morrisett et al. 1995]. In Paraskevopoulou and Appel’s low-level calculus, there is no notion of “invisible” roots. In our high-level calculus, on the contrary, we believe that the distinction between “invisible” and “visible” roots naturally arises.

Besson et al. [2019] prove that (an enhanced version of) CompCert [Leroy 2021] preserves memory consumption when compiling C programs. 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 target of closure conversion; our cost model is analogous to theirs. As explained earlier (§2), our work is complementary: adapting our program logic to DataLang would allow obtaining end-to-end space complexity guarantees about CakeML programs.

10 CONCLUSION

We have presented a Separation Logic to reason about heap space consumption in a high-level programming language with mutable dynamically-allocated data structures, closures, and garbage collection. Our main contributions include a novel treatment of the concept of root in a program logic and novel high-level reasoning rules for closures. We have verified a gallery of challenging examples, which involve abstract “container” data structures, first-class closures, closures with shared internal state, and amortized complexity arguments, among other aspects. Our main directions for future work include supporting concurrency; supporting weak references [Hallet and Kfoury 2005] and ephemeron [Hayes 1997]; and refining our cost model to account for certain compiler optimizations, such as the static allocation of structured constants.
REFERENCES


