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

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

We present a Separation Logic with space credits for reasoning about heap space in a sequential call-by-value λ-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:
Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 7, POPL, Article 25 (January 2023), 29 pages. 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.

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 $\diamond$, 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 λ-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 ℓ such that ℓ occurs in the term that is undergoing reduction.* In slightly more informal words, ℓ is a root if and only if it appears possible that ℓ might be used in the future, based on the existence of a path from the current program point to a program point where ℓ 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 ℓ 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

```
1  let rec rev_append(xs, ys) =
2    if is_nil(xs) then ys else
3      let x = head(xs) in
4      let xs' = tail(xs) in
5      let ys' = cons(x, ys) in
6      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 `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 `is_nil`, `head`, `tail`, and `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 $\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2$. To reason about this construct, one first focuses on the term t_1, thereby temporarily forgetting the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$, which is pushed onto the unknown context. After the verification of t_1 is completed, this focusing step is reversed: the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$ is popped and one continues with the verification of t_2. These focusing and defocusing steps are described by the "bind" rule 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 ℓ that occurs in the frame $\mathsf{let}\, x = [] \,\mathsf{in}\, t_2$ (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 *Stackable* ℓ p, where p is a rational number such that $0 < p \leq 1$. The presence of a fraction allows *Stackable* assertions to be split and joined.
- The assertion *Stackable* ℓ 1 appears when a fresh memory block is allocated at address ℓ, and is eventually consumed when this block is logically deallocated.
- When ℓ is "pushed onto the stack" in an instance of the "bind" rule, an assertion *Stackable* ℓ p is consumed, where the choice of p is up to the user; when ℓ 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 ℓ onto the stack" requires a fractional assertion *Stackable* ℓ p. Thus, this fractional assertion can be intuitively regarded as a *permission* to push ℓ onto the stack, whence the name *Stackable*. Because this assertion is splittable, it allows pushing ℓ onto the stack as many times as one wishes. One can also see intuitively that if the full assertion *Stackable* ℓ 1 is at hand, then no fraction of it has been consumed, so ℓ currently is not "on the stack", that is, not an invisible root. Thus, *Stackable* ℓ 1 serves as a witness that ℓ 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 *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 *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 λ-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.

$$
\begin{array}{llllll}
\text{Values} & v, w ::= () \mid n \in \mathbb{N} \mid \ell \in \mathcal{L} \mid \mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t \quad \text{where } fv(t) \subseteq \{f\} \cup \vec{x} & & & & \\
\text{Blocks} & b ::= \vec{w} \mid \lightning & & & & \\
\text{Arithmetic} & \odot ::= + \mid - \mid \times & & & & \\
\text{Terms} & t, u ::= v & \textit{value} & & t \odot t & \textit{arithmetic} \\
 & \quad x & \textit{variable} & & \mathsf{alloc}\ t & \textit{heap allocation} \\
 & \quad \mathsf{let}\ x = t\ \mathsf{in}\ t & \textit{sequencing} & & t[t] & \textit{heap load} \\
 & \quad \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \textit{conditional} & & t[t] \leftarrow t & \textit{heap store} \\
 & \quad (t\ \vec{u})_{\mathsf{ptr}} & \textit{code pointer invocation} & & & \\
\text{Contexts} & K ::= \mathsf{let}\ x = \square\ \mathsf{in}\ t & \mid \mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t & \mid \mathsf{alloc}\ \square & \mid \square[t] & \mid v[\square] \\
 & \quad \square[t] \leftarrow t & \mid v[\square] \leftarrow t & \mid v[v] \leftarrow \square & \mid \square \odot t & \mid v \odot \square \\
 & \quad (\square\ \vec{u})_{\mathsf{ptr}} & \mid (v\ (\vec{w} \mathbin{++} \square \mathbin{++} \vec{u}))_{\mathsf{ptr}} & & &
\end{array}
$$

Fig. 2. Syntax of SpaceLambda

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 λ-calculus, where a λ-abstraction is a value. Instead, two approaches come to mind. One approach is to view a λ-abstraction as a primitive expression (not a value) whose evaluation causes the allocation of a closure. Another approach is to adopt a restricted calculus that offers only closed functions (as opposed to general λ-abstractions with free variables) and to *define* closure construction and closure invocation as *macros*, or canned sequences of instructions, on top of this calculus. As shown by Paraskevopoulou and Appel [2019], these 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 $\mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t$ for a (recursive) closed function, and write $(v\ \vec{u})_{\mathsf{ptr}}$ 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* $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$ and *closure invocation* $(\ell\ \vec{u})_{\mathsf{clo}}$ 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. $\mathcal{L}$ 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 ℓ, or a code pointer $\mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t$, 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 $\lightning$. Our operational semantics does not recycle memory locations: when a heap block at address ℓ is deallocated, the store is updated with a mapping from ℓ to $\lightning$. A *heap* or *store* σ is a finite map of locations to memory blocks. We write $\emptyset$ for the empty store.

$$\textsc{HeadBinOp}\quad n \odot m \,/\, \sigma \longrightarrow n \odot_{\mathbb{N}} m \,/\, \sigma \qquad \textsc{HeadLet}\quad \mathsf{let}\, x = v \,\mathsf{in}\, t \,/\, \sigma \longrightarrow [v/x]t \,/\, \sigma \qquad \textsc{HeadCall}\ \dfrac{v = \mu_{\mathrm{ptr}} f.\, \lambda \vec{x}.\, t \qquad |\vec{x}| = |\vec{w}|}{(v\ \vec{w})_{\mathrm{ptr}} \,/\, \sigma \longrightarrow [v/f][\vec{w}/\vec{x}]t \,/\, \sigma}$$

$$\textsc{HeadIfTrue}\ \dfrac{n \neq 0}{\mathsf{if}\, n \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2 \,/\, \sigma \longrightarrow t_1 \,/\, \sigma} \qquad \textsc{HeadIfFalse}\ \dfrac{n = 0}{\mathsf{if}\, n \,\mathsf{then}\, t_1 \,\mathsf{else}\, t_2 \,/\, \sigma \longrightarrow t_2 \,/\, \sigma} \qquad \textsc{HeadAlloc}\ \dfrac{\ell \notin \mathit{dom}(\sigma) \quad \sigma' = [\ell := ()^n]\sigma \qquad \mathit{size}(\sigma') \leq S}{\mathsf{alloc}\ n \,/\, \sigma \longrightarrow \ell \,/\, \sigma'}$$

$$\textsc{HeadLoad}\ \dfrac{\sigma(\ell) = \vec{w} \qquad 0 \leq i < |\vec{w}| \qquad \vec{w}(i) = v}{\ell[i] \,/\, \sigma \longrightarrow v \,/\, \sigma} \qquad \textsc{HeadStore}\ \dfrac{\sigma' = [\ell := [i := v]\vec{w}]\sigma \qquad \sigma(\ell) = \vec{w} \qquad 0 \leq i < |\vec{w}|}{\ell[i] \leftarrow v \,/\, \sigma \longrightarrow () \,/\, \sigma'}$$

Fig. 3. Head reduction

$$\textsc{StepHead}\ \dfrac{t \,/\, \sigma \longrightarrow t' \,/\, \sigma'}{t \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, \sigma'} \qquad \textsc{StepCtx}\ \dfrac{t \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, \sigma'}{K[t] \,/\, \sigma \xrightarrow{\mathsf{step}} K[t'] \,/\, \sigma'} \qquad \textsc{Edge}\ \dfrac{\sigma(\ell) = \vec{w} \qquad \vec{w}(i) = \ell'}{\ell \rightsquigarrow_\sigma \ell'}$$

$$\textsc{GC}\ \dfrac{\mathit{dom}(\sigma') = \mathit{dom}(\sigma) \quad \forall \ell \in \mathit{dom}(\sigma') \begin{cases} \sigma'(\ell) = \sigma(\ell) \\ \vee\ \sigma'(\ell) = \lightning\ \wedge\ \neg\, (\exists r \in R,\ \ r \rightsquigarrow_\sigma^* \ell) \end{cases}}{R \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'} \qquad \textsc{RedGC}\ \dfrac{\mathit{locs}(t) \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'}{t \,/\, \sigma \xrightarrow{\mathsf{step} \cup \mathsf{gc}} t \,/\, \sigma'} \qquad \textsc{RedStep}\ \dfrac{t \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, \sigma'}{t \,/\, \sigma \xrightarrow{\mathsf{step} \cup \mathsf{gc}} t' \,/\, \sigma'}$$

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

Our semantics is parameterized by a function, written $\mathit{words}(n)$, which returns the size in words of an n-field memory block. We define the size of a block, written $\mathit{size}(b)$, by $\mathit{size}(\vec{w}) = \mathit{words}(|\vec{w}|)$ and $\mathit{size}(\lightning) = 0$. 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 $\mathit{words}(n) = n$. 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 $t \,/\, \sigma \longrightarrow t' \,/\, \sigma'$. Load and store operations require a valid location and a valid offset. In HeadLoad and HeadStore, we write $\vec{w}(i)$ 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 $[\ell := \vec{w}]\sigma$ for a store update. In 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 $\mathit{size}(\sigma') \leq S$ in HeadAlloc. Thus, either a garbage collection step can free up a sufficient amount of memory, or the program is stuck.

3.4 Reduction under a Context & Garbage Collection

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

Garbage Collection. The relation $R \vdash \sigma \xrightarrow{\text{gc}} \sigma'$, defined by the rule GC in Figure 4, lets a store σ evolve to a store σ' through a GC step that respects a set of roots R. During such a GC step, any location ℓ that is unreachable from every root $r \in R$ may be deallocated. This is reflected by setting $\sigma'(\ell)$ to $\lightning$. The existence of a path in the store from r to ℓ is written $r \rightsquigarrow^*_\sigma \ell$. This is the reflexive and transitive closure of the *edge* relation defined by the rule 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 REDGC and REDSTEP, is the union of reduction under a context and garbage collection. In REDGC, the parameter R, which represents the set of roots that the GC must respect, is instantiated with $locs(t)$, 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 Φ for assertions, $\ulcorner P \urcorner$ for a pure assertion, $\Phi * \Phi'$ for a separating conjunction and $\Phi \mathrel{-\!\!*} \Phi'$ for a separating implication. We express the logical equivalence of two formulas as $\Phi \equiv \Phi'$.

Triples take the form $\{\Phi\}\ t\ \{\Psi\}$. A postcondition Ψ is of the form $\lambda v.\, \Phi'$, where the metavariable v, which denotes the value of the term t, is bound in Φ'. We write $\{\Phi\}\ t\ \{\lambda \ell.\, \Phi'\}$, where the metavariable ℓ denotes a location, as syntactic sugar for $\{\Phi\}\ t\ \{\lambda v.\ \exists \ell.\ \ulcorner v = \ell \urcorner * \Phi'\}$. 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 $\Phi \Rrightarrow_V \Phi'$ is parameterized with a set V of visible roots. This parameter is instantiated with $locs(t)$ in the consequence rule, as shown below. The frame rule retains its standard form.

$$\frac{\Phi \Rrightarrow_{locs(t)} \Phi' \qquad \{\Phi'\}\ t\ \{\Psi\}}{\{\Phi\}\ t\ \{\Psi\}}\ \textsc{Conseq} \qquad\qquad \frac{\{\Phi\}\ t\ \{\Psi\}}{\{\Phi * \Phi'\}\ t\ \{\lambda v.\ \Psi\, v * \Phi'\}}\ \textsc{Frame}$$

4.2 Logical Deallocation

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

LOGICALFREE
$$\ell \mapsto_1 b \ *\ \ell \hookleftarrow_1 \emptyset \ *\ \ulcorner \ell \notin V \urcorner \ *\ \mathit{Stackable}\ \ell\ 1 \quad \Rrightarrow_V \quad \diamond size(b) \ *\ \dagger\, \ell$$

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 $\ell \notin V$, which ensures that ℓ is not a visible root (§4.4); and a *Stackable* assertion, which ensures that ℓ is not an invisible root (§2.4).

Two assertions are produced by this ghost update, namely the space credits $\diamond size(b)$ associated with the deallocated block b, and a *deallocation witness* $\dagger\, \ell$ (§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 $\ell' \hookleftarrow_q L$, 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:

$$v \hookleftarrow_{q_1+q_2} (L_1 \uplus L_2) \;\twoheadrightarrow\; (v \hookleftarrow_{q_1} L_1 \;*\; v \hookleftarrow_{q_2} L_2) \quad \text{if } q_1 > 0 \land q_2 > 0 \qquad \textsc{SplitPointedBy}$$

$$(v \hookleftarrow_{q_1} L_1 \;*\; v \hookleftarrow_{q_2} L_2) \;\twoheadrightarrow\; v \hookleftarrow_{q_1+q_2} (L_1 \uplus L_2) \qquad \textsc{JoinPointedBy}$$

Our program logic provides a *full knowledge* property: a *full* pointed-by assertion $\ell' \hookleftarrow_1 L$ (with fraction 1) guarantees that the multiset L contains *all* of the predecessors of ℓ'. Of particular interest is the assertion $\ell' \hookleftarrow_1 \emptyset$, 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 $\ell[i]$ and overwrites the value ℓ'_1 with the value ℓ'_2. The reasoning rule for this operation (§4.7) removes ℓ from a pointed-by assertion for ℓ'_1 and adds ℓ to a pointed-by assertion for ℓ'_2.

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 $\ell' \hookleftarrow_q L$ 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 $\ell' \hookleftarrow_0 \{-\ell\}$, 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 $\mathbb{Z}$. The disjoint union operation $\uplus$ is the pointwise addition of multiplicities. We write $+x$ for a positive occurrence of x and $-x$ for a negative occurrence of x. For example, $\{+x; +x\} \uplus \{-x\}$ is $\{+x\}$.

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 $\ell' \hookleftarrow_1 L$ does *not* exclude the existence of a separate pointed-by assertion with fraction zero, say $\ell' \hookleftarrow_0 L'$. 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 $\ell' \hookleftarrow_1 L$ remains a sound over-approximation of the true multiset of predecessors of ℓ'. In particular, as desired, the assertion $\ell' \hookleftarrow_1 \emptyset$ does guarantee that ℓ' has no predecessors.

We extend pointed-by assertions to arbitrary values and introduce an assertion of the form $v \hookleftarrow_q L$. If v is a location ℓ', then this assertion is defined as $\ell' \hookleftarrow_q L$. Otherwise, it is defined as $\ulcorner \text{True} \urcorner$. Likewise, we extend the *Stackable* assertion to arbitrary values and define *Stackable* $v\ p$. Finally, we write $\ell' \hookleftarrow_q^{>0} L$ as a short-hand for the assertion $\ulcorner q > 0 \urcorner * \ell' \hookleftarrow_q L$. This notation is used for instance in the reasoning rule Store (§4.7).

4.4 Reasoning about Visible Roots

The rule LogicalFree requires $\ell \notin V$. As explained earlier (§4.1), the set V is instantiated by the consequence rule with *locs*(t). Thus, a user of our logic is expected to fulfill a proof obligation of the form $\ell \notin$ *locs*(t). The term t is known: it is the subterm on which the user is focusing. The set

$locs(t)$ can therefore be computed by Coq. Thus, the proof obligation $\ell \notin V$ can be reformulated as follows: for each location ℓ' that appears in the set $locs(t)$, the user must prove $\ell \neq \ell'$.

As pointed out earlier (§2.3), this proof obligation is not a syntactic check: the user must prove that the metavariables ℓ and ℓ' 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: $\ell \mapsto_1 b \ast \ell' \mapsto_p b'$ entails $\ell \neq \ell'$, and, likewise, $\ell \hookleftarrow_1 L \ast \ell' \hookleftarrow_q^{\geq 0} L'$ entails $\ell \neq \ell'$. When the user applies LogicalFree, the assertions $\ell \mapsto_1 b$ and $\ell \hookleftarrow_1 \emptyset$ are at hand already. Thus, to prove $\ell \neq \ell'$, it suffices to exhibit either $\ell' \mapsto_p b'$, or $\ell' \hookleftarrow_q^{\geq 0} L'$. 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 $\diamond n$ 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: $\diamond(n_1 + n_2) \equiv \diamond n_1 \ast \diamond n_2$. One can forge zero credits, via the rule: $\ulcorner \text{True} \urcorner \Rrightarrow_V \diamond 0$.

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 ℓ has been deallocated. It is *persistent*: once it holds, it holds forever. It serves as a permission to remove an occurrence of ℓ from a predecessor multiset—*deallocated predecessors need not be recorded.* The Cleanup rule, shown below, expresses this possibility. This rule was introduced by Madiot and Pottier [2022].

$$\dagger\, \ell \;\ast\; \ell' \hookleftarrow_q (L \uplus \{+\ell\}) \quad \Rrightarrow_V \quad \ell' \hookleftarrow_q L \qquad\qquad \text{Cleanup}$$

4.7 Reasoning Rules for Terms

Figure 5 gives the syntax-directed reasoning rules of our logic. The rules BinOp, IfTrue, IfFalse, LetVal, and 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.

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 *Stackable* assertion. These assertions indicate that there are no pointers from the heap or the stack to the new block.

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 Store rule is slightly more complex than the standard rule of Separation Logic. Like the standard rule, it requires a full points-to assertion $\ell \mapsto_1 \vec{w}$ and produces an updated assertion $\ell \mapsto_1 [i := v']\vec{w}$. 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 ℓ to the value v', an assertion of the form $v' \hookleftarrow_q^{\geq 0} L$ is changed to $v' \hookleftarrow_q^{\geq 0} L \uplus \{+\ell\}$.

BinOp

$$\left\{\ulcorner \mathrm{True} \urcorner\right\}\ m \odot n\ \left\{\lambda v.\ \ulcorner v = m \odot_{\mathbb{N}} n \urcorner\right\}$$

IfTrue

$$\frac{n \neq 0 \qquad \triangleright \{\Phi\}\ t_1\ \{\Psi\}}{\{\Phi\}\ \mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \{\Psi\}}$$

IfFalse

$$\frac{n = 0 \qquad \triangleright \{\Phi\}\ t_2\ \{\Psi\}}{\{\Phi\}\ \mathsf{if}\ n\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2\ \{\Psi\}}$$

Bind

$$\frac{\begin{array}{c} \mathit{dom}(M) = \mathit{locs}(K) \qquad \{\Phi\}\ t\ \{\Psi'\} \\ \forall v.\ \{\Psi'\ v\ *\ \mathit{Stackables}\ M\}\ K[v]\ \{\Psi\} \end{array}}{\{\Phi\ *\ \mathit{Stackables}\ M\}\ K[t]\ \{\Psi\}}$$

LetVal

$$\frac{\{\Phi\}\ [v/x]t\ \{\Psi\}}{\{\Phi\}\ \mathsf{let}\ x = v\ \mathsf{in}\ t\ \{\Psi\}}$$

CallPtr

$$\frac{\begin{array}{c} v = \mu_{\mathrm{ptr}} f.\ \lambda \vec{x}.\ t \qquad |\vec{x}| = |\vec{w}| \\ \triangleright \{\Phi\}\ [v/f][\vec{w}/\vec{x}]t\ \{\Psi\} \end{array}}{\{\Phi\}\ (v\ \vec{w})_{\mathrm{ptr}}\ \{\Psi\}}$$

Alloc

$$\left\{\diamond \mathit{size}(()^n)\right\}\ \mathsf{alloc}\ n\ \left\{\lambda \ell.\ \begin{array}{c} \ell \mapsto_1 ()^n \\ \ell \mapsfrom_1 \emptyset \\ \mathit{Stackable}\ \ell\ 1 \end{array}\right\}$$

Load

$$\left\{\ell \mapsto_p \vec{w}\right\}\ \ell[i]\ \left\{\lambda v.\ \begin{array}{c} \ulcorner v = \vec{w}(i) \urcorner \\ \ell \mapsto_p \vec{w} \end{array}\right\}$$

Store

$$\frac{\vec{w}(i) = v}{\left\{\begin{array}{c} \ell \mapsto_1 \vec{w} \\ v' \mapsfrom_q^{>0} L \\ \end{array}\right\}\ \ell[i] \leftarrow v'\ \left\{\lambda\_.\ \begin{array}{c} \ell \mapsto_1 [i := v']\vec{w} \\ v' \mapsfrom_q^{>0} L \uplus \{+\ell\} \\ v \mapsfrom_0 \{-\ell\} \end{array}\right\}}$$

Fig. 5. Reasoning rules for terms

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 ℓ to the value v, the assertion $v \mapsfrom_0 \{-\ell\}$ appears in the postcondition. As explained earlier (§4.3), this assertion is a permission to remove ℓ 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 Store rule requires the assertion $v \mapsfrom_p L$ and changes it to $v \mapsfrom_p L \setminus \{\ell\}$. 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 Bind rule, let us begin with a special case. Suppose we wish to reason about the term $\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2$ and suppose $\mathit{locs}(t_2)$ 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 $\mathit{Stackable}\ \ell\ p$ 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 $\mathit{Stackable}\ \ell\ p$ in the following rule:

Particular Case of Bind

$$\frac{\mathit{locs}(t_2) = \{\ell\} \qquad \{\Phi\}\ t_1\ \{\Psi'\} \qquad \forall v.\ \{\Psi'\ v * \mathit{Stackable}\ \ell\ p\}\ [v/x]t_2\ \{\Psi\}}{\{\Phi * \mathit{Stackable}\ \ell\ p\}\ \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2\ \{\Psi\}}$$

The choice of the fraction p is up to the user. Our 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 $\mathit{locs}(K)$, a *Stackable* assertion. This is expressed as follows: for some map M of the locations in $\mathit{locs}(K)$ to fractions, the rule forcibly frames out the assertion $\mathit{Stackables}\ M$, which is defined as an iterated separating conjunction of *Stackable* assertions:

$$\mathit{Stackables}\ M = \mathop{\Huge *}_{(\ell, p) \in M} \mathit{Stackable}\ \ell\ p.$$

$$\textsc{BindWithSouvenir}\quad \frac{dom(M) = locs(K) \setminus R \qquad \langle R \cup locs(K)\rangle\, \{\Phi\}\, t\, \{\Psi'\} \qquad \forall v.\, \langle R\rangle\, \{\Psi'\, v * \mathit{Stackables}\ M\}\, K[v]\, \{\Psi\}}{\langle R\rangle\, \{\Phi * \mathit{Stackables}\ M\}\, K[t]\, \{\Psi\}}$$

$$\textsc{AddSouvenir}\quad \frac{\langle R \cup \{\ell\}\rangle\, \{\Phi\}\, t\, \{\Psi\}}{\langle R\rangle\, \{\Phi * \mathit{Stackable}\ \ell\ p\}\, t\, \{\lambda v.\, \Psi\ v * \mathit{Stackable}\ \ell\ p\}} \qquad \textsc{ForgetSouvenir}\quad \frac{R' \subseteq R \qquad \langle R'\rangle\, \{\Phi\}\, t\, \{\Psi\}}{\langle R\rangle\, \{\Phi\}\, t\, \{\Psi\}}$$

Fig. 6. Key reasoning rules for triples with souvenir

$$\textsc{BindNoFree}\quad \frac{\langle \mathsf{NoFree}\rangle\, \{\Phi\}\, t\, \{\Psi'\} \qquad \forall v.\, \langle R^?\rangle\, \{\Psi'\, v\}\, K[v]\, \{\Psi\}}{\langle R^?\rangle\, \{\Phi\}\, K[t]\, \{\Psi\}} \qquad \textsc{ConseqMode}\quad \frac{\begin{array}{c} m \ = \ \mathsf{if}\ (R^? = \mathsf{NoFree})\ \mathsf{then}\ \bot\ \mathsf{else}\ \top \\ \Phi \Rrightarrow^{m}_{locs(t)} \Phi' \qquad \langle R^?\rangle\, \{\Phi'\}\, t\, \{\Psi\} \end{array}}{\langle R^?\rangle\, \{\Phi\}\, t\, \{\Psi\}}$$

Fig. 7. Key reasoning rules for the NoFree mode

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 R of roots. While establishing a triple with souvenir, logical deallocation 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 deallocation is forbidden and no *Stackable* assertions at all are forcibly framed.

5.1 Triples with Souvenir

A *triple with souvenir* takes the form $\langle R\rangle\, \{\Phi\}\, t\, \{\Psi\}$. 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 $R \subseteq dom(M)$ requires M to cover every location in R.

$$\langle R\rangle\, \{\Phi\}\, t\, \{\Psi\} \ \equiv\ \big(\forall M.\ \ R \subseteq dom(M) \implies \{\Phi\ *\ \mathit{Stackables}\ M\}\ t\ \{\lambda v.\, \Psi\, v\ *\ \mathit{Stackables}\ M\}\big)$$

Conversely, a plain triple $\{\Phi\}\ t\ \{\Psi\}$ is equivalent to a triple with an empty souvenir $\langle\emptyset\rangle\, \{\Phi\}\, t\, \{\Psi\}$.

For every reasoning rule in Figure 5, we provide a new rule (not shown) that operates on triples with souvenir and that is polymorphic in R. This is done simply by inserting R in every triple in the premises and conclusion. In addition, we establish three new reasoning rules, presented in Figure 6. The rule BindWithSouvenir is similar to Bind, but does not require *Stackable* assertions for the locations that are already part of the souvenir R. Furthermore, it augments the current souvenir by changing R to $R \cup locs(K)$ in its second premise. Thus, nested applications of BindWithSouvenir do not require repeated (redundant) force-framing of *Stackable* assertions. The rule AddSouvenir extends the current souvenir with a location ℓ. This requires framing a *Stackable* assertion for ℓ. The rule ForgetSouvenir shrinks the current souvenir.

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 AddSouvenir and ForgetSouvenir 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 BindWithSouvenir. For the latter, we define a custom tactic that computes the set of locations $locs(K) \setminus R$, 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 LogicalFree carries the premise $\ell \notin V$, 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 $\ell \neq \ell'$ 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 $\ell \notin V$ can often be automatically met, or at least reduced to a weaker obligation $\ell \notin V'$, where V' is a subset of V. This idea is expressed by the following two rules:

$$\textsc{ConseqWithSouvenir}\quad \frac{\Phi \Rrightarrow_{locs(t)\setminus R} \Phi' \qquad \langle R\rangle\,\{\Phi'\}\,t\,\{\Psi\}}{\langle R\rangle\,\{\Phi\}\,t\,\{\Psi\}}$$

$$\textsc{UpdateWithSouvenir}\quad \frac{\Phi \Rrightarrow_{V\setminus\{\ell'\}} \Phi'}{\Phi * \mathit{Stackable}\ \ell'\ p \Rrightarrow_V \Phi' * \mathit{Stackable}\ \ell'\ p}$$

ConseqWithSouvenir is a variant of Conseq. When the user would like to deallocate a location ℓ, this rule automatically proves $\ell \notin R$, so the proof obligation $\ell \notin locs(t)$ is replaced with $\ell \notin locs(t) \setminus R$. This proof obligation can be further weakened using the UpdateWithSouvenir 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 $\ell \notin V$ in an application of the logical deallocation rule LogicalFree is reduced to $\ell \notin V \setminus \{\ell'\}$.

5.3 Triples in NoFree mode

Inside the proof of a triple with souvenir $\langle R\rangle\,\{\Phi\}\,t\,\{\Psi\}$, 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 $\langle R^?\rangle\,\{\Phi\}\,t\,\{\Psi\}$ 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 BindNoFree is an alternative to BindWithSouvenir. 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 ConseqMode is a generalized formulation of the consequence rule Conseq (§4.1). In short, this rule simply forbids logical deallocation in NoFree mode. In greater detail, we parameterize the ghost update modality $\Rrightarrow_V^m$ with a *mode* m that is either $\bot$ (deallocation forbidden) or $\top$ (deallocation permitted). The modality $\Rrightarrow_V$ that appears in the rule LogicalFree coincides with $\Rrightarrow_V^\top$, so deallocation requires the mode $\top$. The first premise of ConseqMode in Figure 7 selects a suitable mode, depending on $R^?$, so as to forbid deallocation in 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 *Stackable* assertions; thus, during a proof, one can switch back and forth between ordinary triples and triples with souvenir (AddSouvenir, ForgetSouvenir). In contrast, NoFree mode is more restricted: once one switches to NoFree mode via 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 / σ_1 is *final* if the term t_1 is a value. A configuration t_1 / σ_1 is *reducible* if after one step of garbage collection it can take a proper reduction step: that is, if there exist two stores σ_1' and σ_2 and a term t_2 such that $locs(t_1) \vdash \sigma_1 \xrightarrow{\text{gc}} \sigma_1'$ and $t_1 / \sigma_1' \xrightarrow{\text{step}} t_2 / \sigma_2$ hold. A configuration is *stuck* if it is neither final nor reducible. A program t is *safe* if $t / \emptyset \xrightarrow{\text{step} \cup \text{gc}}{}^* t_1 / \sigma_1$ implies that the configuration t_1 / σ_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, a limit on 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* $\{\diamond S\}\ t\ \{\lambda\_.\ \ulcorner \mathit{True} \urcorner\}$ *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 $wp\ t\ \Psi$ means that it is safe to execute t and that, if this execution produces a value v, then $\Psi\ v$ holds. The definition of wp refers to the small-step semantics of the language and to a *central invariant* (also known as the *state interpretation invariant*) which relates the physical state and the ghost state. Our definition of 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 wp to allow reasoning independently of the evaluation context, that is, to admit a "bind" rule. Therefore, our definition of 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,

$$
\begin{array}{l}
\mathit{wp}\, m\, t_1\ \Psi \ \triangleq\ \forall \sigma_1, \kappa.\ \ \mathit{interp}\ m\ \sigma_1\ \kappa\ \mathit{locs}(t_1) \ \Rrightarrow \\
\quad \text{match } t_1 \text{ with} \\
\quad \mid \text{Val } v \Rightarrow\ \ \mathit{interp}\ m\ \sigma_1\ \kappa\ \mathit{locs}(t_1) * \Psi\ v \\
\quad \mid \_ \Rightarrow\ \ulcorner \mathit{reducible}\ \mathit{dom}(\kappa)\ t_1\ \sigma_1 \urcorner * \\
\qquad \triangleright (\forall t_2, \sigma_2.\ \ulcorner \mathit{dom}(\kappa) \vdash t_1 \,/\, \sigma_1 \xrightarrow{\mathsf{ctx} \cup (\mathsf{gc}\,;\,\mathsf{head})} t_2 \,/\, \sigma_2 \urcorner \ \Rrightarrow\ (\mathit{interp}\ m\ \sigma_2\ \kappa\ \mathit{locs}(t_2) * \mathit{wp}\, m\, t_2\ \Psi))
\end{array}
$$

Fig. 8. Definition of the weakest precondition modality *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 *wp*, the set of invisible roots takes the form $dom(\kappa)$, where κ is a finite map of memory locations to nonzero fractions. The metavariable κ is universally quantified at the root of the definition, in the same way as the initial store σ_1. This reflects the intuitive idea that the assertion $wp\, m\, t_1\ \Psi$ 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 σ_1 via a points-to assertion [Jung et al. 2018, §6.3.2] and, in a similar way, to constrain the map κ via a *Stackable* assertion. We note that our *wp* is parameterized with a Boolean mode $m \in \{\bot; \top\}$, where $\top$ describes the normal mode, and $\bot$ denotes the NoFree mode (§5.3), a restricted mode where logical deallocation is not permitted. Our central invariant *interp* is parameterized not only with a store, as usual, but also by a mode m, a map κ (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 *closure construction* $\mu_{\mathsf{clo}} f.\,\lambda\vec{x}.\,t$ and *closure invocation* $(\ell\ \vec{u})_{\mathsf{clo}}$ as macros, which expand to sequences of primitive 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 *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 $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$, which asserts that, at location ℓ in the heap, one finds a closure that behaves like the function $\mu f.\lambda\vec{x}.\,t$ under the environment E. Crucially, in this assertion, the term $\mu f.\lambda\vec{x}.\,t$ *can* have free variables, whose values are given by E. This assertion does not reveal how a closure is represented in memory, but does reveal its code. We give an overview of this low-level API (§7.1, §7.2) and reveal some details of its implementation (§7.3). Second, we define a high-level assertion $\mathit{Spec}\, n\, E\, P\, \ell$, which describes the behavior of a closure in a more abstract way. It asserts that, at location ℓ, one finds a closure that corresponds to a n-ary function, whose behavior is described by the predicate P, and whose environment is E. The 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.

$$\textsc{MkClo}\quad \frac{\vec{y} = fvclo(f, \vec{x}, t) \qquad E = zip\ \vec{w}\ \vec{q} \qquad |\vec{w}| = |\vec{y}| \qquad f \notin \vec{x}}{\left\{ \begin{array}{c} \diamond(1 + |E|) \\ \mathop{\Huge *}\limits_{(w,q)\in E} w \hookleftarrow_q^{>0} \emptyset \end{array} \right\}\ [\vec{w}/\vec{y}]\ (\mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t)\ \left\{ \lambda\ell.\ \begin{array}{c} Closure\, E\, f\, \vec{x}\, t\, \ell \\ Stackable\ \ell\ 1\ *\ \ell \hookleftarrow_1 \emptyset \end{array} \right\}}$$

$$\textsc{CallClo}\quad \frac{\begin{array}{c} \vec{y} = fvclo(f, \vec{x}, t) \qquad E = zip\ \vec{w}\ \vec{q} \qquad |\vec{x}| = |\vec{v}| \\ \triangleright \{Closure\, E\, f\, \vec{x}\, t\, \ell * \Phi\}\ [\ell/f][\vec{w}/\vec{y}][\vec{v}/\vec{x}]t\ \{\Psi\} \end{array}}{\{Closure\, E\, f\, \vec{x}\, t\, \ell * \Phi\}\ (\ell\ \vec{v})_{\mathrm{clo}}\ \{\Psi\}}$$

$$\textsc{FreeClo}\quad \begin{pmatrix} \ulcorner \ell \notin V \urcorner \\ Closure\, E\, f\, \vec{x}\, t\, \ell \\ Stackable\ \ell\ 1 \\ \ell \hookleftarrow_1 \emptyset \end{pmatrix} \Rrightarrow_V^{\top} \begin{pmatrix} \diamond(1 + |E|) \\ \mathop{\Huge *}\limits_{(w,q)\in E} w \hookleftarrow_q^{>0} \emptyset \end{pmatrix}$$

Fig. 9. Low-level interface for closures

7.1 Environments

We write $fvclo(f, \vec{x}, t)$ for a list of the free variables of the function $\mu f.\lambda\vec{x}.\, t$, that is, for a list of the variables in the set $fv(t) \setminus \{f, \vec{x}\}$. The order in which the variables occur in this list 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 $fvclo(f, \vec{x}, t)$. 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 *Closure* and *Spec*.

7.2 Low-Level Closure API

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

The rule MkClo specifies a closure construction operation. The term, written $[\vec{w}/\vec{y}]\ \mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t$, is the application of a multi-substitution of some values $\vec{w}$ for the free variables $\vec{y}$ of the function $\mu f.\lambda\vec{x}.\, t$ to the closure construction macro $\mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t$. 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 $\mu_{\mathrm{clo}} f.\, \lambda\vec{x}.\, t$. The values $\vec{w}$ 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 MkClo, an environment E is built by pairing up the values $\vec{w}$ with nonzero fractions $\vec{q}$. 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 MkClo, for each value w in the list $\vec{w}$, the closure construction operation consumes $w \hookleftarrow_q^{>0} \emptyset$. (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 $\vec{w}$ in a flat closure. According to the postcondition in MkClo, this operation produces the assertion $Closure\, E\, f\, \vec{x}\, t\, \ell$, which guarantees that there is a well-formed closure at address ℓ, as well as a *Stackable* assertion and a pointed-by assertion for ℓ, which guarantee that we have a unique pointer to this closure.

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

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

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

7.3 Low-Level Closure API: Implementation Details

The $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ assertion is defined as follows.

$$\begin{aligned}\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell \quad &\triangleq \quad \ulcorner f \notin \vec{x} \;\wedge\; |E| = |\mathit{fvclo}(f,\vec{x},t)|\urcorner \;*\\ &\qquad \ell \mapsto_1 (\mathit{codeclo}(f,\vec{x},t) :: \mathit{map\,fst}\,E) \;*\; \mathop{\Huge *}_{(v,\,q)\in E} (v \hookleftarrow_q \{+\ell\})\end{aligned}$$

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 ℓ points to a block of size $1 + |E|$. Its first field contains the code of the closure, $\mathit{codeclo}(f,\vec{x},t)$, 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 $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ reveals the code of the closure. As a result, a user of the *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 $\mathit{Spec}\,n\,E\,P\,\ell$ (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 ℓ 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 *Spec* while establishing an instance of *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 *Spec* assumption, if desired. For presentation purposes, we first present a simplified assertion, $\mathit{Spec}'\,n\,E\,P\,\ell$, which handles closures that are not recursive and do not self-destruct (Figure 10). Then, we show the general case (Figure 11).

A Simplified Interface. The Spec' API is presented in Figure 10. Let us first examine the rule CallSpec'. A call of the form $(\ell\ \vec{v})_{\mathrm{clo}}$ admits a precondition Φ and a postcondition Ψ if the entailment $\forall u.\ P\ \vec{v}\ u \twoheadrightarrow \{\Phi\}\ u\ \{\Psi\}$ holds. Intuitively, u denotes the instantiated function body that was visible in the low-level rule 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

$$\textsc{MkSpec'}\quad \dfrac{\begin{array}{c}\vec{y} = \mathit{fvclo}(f,\vec{x},t) \qquad E = \mathit{zip}\ \vec{w}\ \vec{q} \qquad |\vec{w}| = |\vec{y}| \qquad f \notin \vec{x} \qquad f \notin \mathit{fv}(t)\\ n = |\vec{x}| \qquad \forall \vec{v}.\ \ulcorner |\vec{v}| = n \urcorner \mathrel{-\!\!*} \text{let } u = [\vec{w}/\vec{y}][\vec{v}/\vec{x}]t \text{ in } \Box(P\,\vec{v}\,u)\end{array}}{\left\{\begin{array}{c}\diamond(1+|E|)\\ \mathop{\huge *}\limits_{(w,q)\in E} w \hookleftarrow^{\geq 0}_{q} \emptyset\end{array}\right\}\ [\vec{w}/\vec{y}]\ (\mu_{\mathrm{clo}} f.\, \lambda \vec{x}.\, t)\ \left\{\lambda \ell.\ \begin{array}{c}\mathit{Spec'}\ n\,E\,P\,\ell\\ \mathit{Stackable}\ \ell\ 1 \ast \ell \hookleftarrow_1 \emptyset\end{array}\right\}}$$

$$\textsc{CallSpec'}\quad \dfrac{|\vec{v}| = n \qquad \triangleright(\forall u.\ P\,\vec{v}\,u \mathrel{-\!\!*} \{\Phi\}\ u\ \{\Psi\})}{\{\mathit{Spec'}\ n\,E\,P\,\ell \ast \Phi\}\ (\ell\ \vec{v})_{\mathrm{clo}}\ \{\lambda v.\ \mathit{Spec'}\ n\,E\,P\,\ell \ast \Psi\,v\}}$$

$$\textsc{FreeSpec'}\quad \left(\begin{array}{c}\ulcorner \ell \notin V \urcorner\\ \mathit{Spec'}\ n\,E\,P\,\ell\\ \mathit{Stackable}\ \ell\ 1\\ \ell \hookleftarrow_1 \emptyset\end{array}\right) \Rrightarrow^{\top}_{V} \left(\begin{array}{c}\diamond(1+|E|)\\ \mathop{\huge *}\limits_{(w,q)\in E} w \hookleftarrow^{\geq 0}_{q} \emptyset\end{array}\right)$$

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

$$\textsc{MkSpec}\quad \dfrac{\begin{array}{c}\vec{y} = \mathit{fvclo}(f,\vec{x},t) \qquad E = \mathit{zip}\ \vec{w}\ \vec{q} \qquad |\vec{w}| = |\vec{y}| \qquad f \notin \vec{x} \qquad n = |\vec{x}| \qquad \mathit{NonExpansive}\ P\\ \forall \ell, \vec{v}.\ \ulcorner |\vec{v}| = n \urcorner \mathrel{-\!\!*} \text{let } u = [\ell/f][\vec{w}/\vec{y}][\vec{v}/\vec{x}]t \text{ in } \Box\big(\mathit{Spec}\,n\,E\,P\,\ell \mathrel{-\!\!*} P\,\ell\,\vec{v}\,u\,(\mathit{Spec}\,n\,E\,P\,\ell)\big)\end{array}}{\left\{\begin{array}{c}\diamond(1+|E|)\\ \mathop{\huge *}\limits_{(w,q)\in E} w \hookleftarrow^{\geq 0}_{q} \emptyset\end{array}\right\}\ [\vec{w}/\vec{y}]\ (\mu_{\mathrm{clo}} f.\, \lambda \vec{x}.\, t)\ \left\{\lambda \ell.\ \begin{array}{c}\mathit{Spec}\,n\,E\,P\,\ell\\ \mathit{Stackable}\ \ell\ 1 \ast \ell \hookleftarrow_1 \emptyset\end{array}\right\}}$$

$$\textsc{CallSpec}\quad \dfrac{|\vec{v}| = n \qquad \triangleright\,(\forall u.\ P\,\ell\,\vec{v}\,u\,(\mathit{Spec}\,n\,E\,P\,\ell) \mathrel{-\!\!*} \{\Phi\}\ u\ \{\Psi\})}{\{\mathit{Spec}\,n\,E\,P\,\ell \ast \Phi\}\ (\ell\ \vec{v})_{\mathrm{clo}}\ \{\Psi\}}$$

$$\textsc{FreeSpec}\quad \left(\begin{array}{c}\ulcorner \ell \notin V \urcorner\\ \mathit{Spec}\,n\,E\,P\,\ell\\ \mathit{Stackable}\ \ell\ 1\\ \ell \hookleftarrow_1 \emptyset\end{array}\right) \Rrightarrow^{\top}_{V} \left(\begin{array}{c}\diamond(1+|E|)\\ \mathop{\huge *}\limits_{(w,q)\in E} w \hookleftarrow^{\geq 0}_{q} \emptyset\end{array}\right)$$

Fig. 11. High-level interface for closures: general case

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: $\lambda\,\vec{v}\,u.\ \forall\,r,\ n.\ulcorner\vec{v} = [r]\urcorner \mathrel{-\!\!*} \{r \mapsto [n]\}\ u\ \{\lambda().\ r \mapsto [n+1]\}$.

Let us now consider the rule MkSpec'. Its first four premises are the same as in 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 $[\vec{w}/\vec{y}][\vec{v}/\vec{x}]t$ 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.

A General Interface. The *Spec* API appears in Figure 11. *Spec* generalizes *Spec'* in three ways. First, in the last premise of MkSpec, we make the assumption $\mathit{Spec}\,n\,E\,P\,\ell$ 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 CallSpec, we get just Ψ, whereas in CallSpec' we would automatically get $\mathit{Spec'}\ n\,E\,P\,\ell$ in addition to Ψ. Thus, with 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 Ψ. Second, we parameterize P with the location ℓ 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 *Stackable* assertion for ℓ. Third, we parameterize P over the assertion $\mathit{Spec}\,n\,E\,P\,\ell$. As a result, the applications of P in the premises of MkSpec and CallSpec take the form $P\ \ell\ \vec{v}\ u\ (\mathit{Spec}\,n\,E\,P\,\ell)$. Without this form of "self-parameterization", many uses of the *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 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 $\mathit{Spec}\, n\, E\, P\, \ell$, thereby allowing the closure to recursively invoke itself or to logically deallocate itself.

The condition *NonExpansive* P in MkSpec asserts that $\Phi \overset{n}{=} \Phi'$ implies $P\,\ell\,\vec{v}\,u\,\Phi \overset{n}{=} P\,\ell\,\vec{v}\,u\,\Phi'$, where $\overset{n}{=}$ 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 $\mathit{Spec}\, n\, E\, P\, \ell$ is defined as follows:

$$
\begin{aligned}
&\mathit{Spec}\, n\, E\, P\, \ell \;\triangleq\; \exists f, \vec{x}, t, Q.\; \ulcorner |\vec{x}| = n \urcorner \;*\; \mathit{Closure}\, E\, f\, \vec{x}\, t \;* \\
&\quad \text{let } \vec{w} = \mathit{map\,fst}\, E \text{ in } \;\; \text{let } \vec{y} = \mathit{fvclo}(f, \vec{x}, t) \text{ in } \;\; \text{let } \mathit{body}\, \vec{v} = [\ell/f][\vec{w}/\vec{y}][\vec{v}/\vec{x}]t \text{ in} \\
&\qquad \triangleright\, \square(\forall \vec{v}.\; \ulcorner |\vec{v}| = n \urcorner \twoheadrightarrow \mathit{Spec}\, n\, E\, Q\, \ell \twoheadrightarrow Q\, \ell\, \vec{v}\, (\mathit{body}\, \vec{v})\, (\mathit{Spec}\, n\, E\, Q\, \ell)) \;* \\
&\qquad \triangleright\, \square(\forall \vec{v}.\; \ulcorner |\vec{v}| = n \urcorner \twoheadrightarrow Q\, \ell\, \vec{v}\, (\mathit{body}\, \vec{v})\, (\mathit{Spec}\, n\, E\, Q\, \ell) \twoheadrightarrow P\, \ell\, \vec{v}\, (\mathit{body}\, \vec{v})\, (\mathit{Spec}\, n\, E\, P\, \ell))
\end{aligned}
$$

This is a guarded recursive definition: *Spec* appears (under a "later" modality) in its own definition. The definition is existentially quantified over the code of the closure, represented by f, $\vec{x}$, and t. It is also existentially quantified over a 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 *rev_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 $v \hookleftarrow_p L$ as sugar for the conjunction $v \mapsfrom_p L * \mathit{Stackable}\ v\ p$. We refer to this new assertion as a *handle* for the value v. In particular, the assertion $v \hookleftarrow_1 \emptyset$, 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 *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:

$$\begin{array}{l}
\mathit{List}\ \mathit{vps}\ \mathit{xs} \triangleq \text{ match } \mathit{vps} \text{ with} \\
\quad \mid [\,] \;\Rightarrow\; \mathit{xs} \mapsto [0] \\
\quad \mid (v,p) :: \mathit{vps}' \;\Rightarrow\; \exists \mathit{xs}'.\ \mathit{xs} \mapsto [1; v; \mathit{xs}'] \ *\ v \hookleftarrow_p \{\mathit{xs}\} \ *\ \mathit{xs}' \hookleftarrow_1 \{\mathit{xs}\} \ *\ \mathit{List}\ \mathit{vps}'\ \mathit{xs}'
\end{array}$$

We adopt the convention that when a value v is inserted into a linked list, a fractional handle $v \hookleftarrow_p \emptyset$ 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 *List vps xs* has a standard overall structure [Reynolds 2002]. When *vps* is empty, this predicate boils down to the points-to assertion $\mathit{xs} \mapsto [0]$. When *vps* is a nonempty list $(v, p) :: \mathit{vps}'$, the predicate begins with the points-to assertion $\mathit{xs} \mapsto [1; v; \mathit{xs}']$, which describes a 3-field cell. Moreover, what is new, it contains the handles $v \hookleftarrow_p \{\mathit{xs}\}$ and $\mathit{xs}' \hookleftarrow_1 \{\mathit{xs}\}$. 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 *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 *rev_append* has *linear* heap space complexity: it requires $3 \times |\mathit{vps}|$ space credits. The second specification requires the caller to provide (and give up)

[3]The predicate *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.

$$\langle\, \{xs\}\,\rangle \begin{Bmatrix} \mathit{List}\ \mathit{vps}\ \mathit{xs}\ *\ \diamond(3\times|\mathit{vps}|) \\ \mathit{List}\ \mathit{wqs}\ \mathit{ys}\ *\ \mathit{ys} \hookleftarrow_1 \emptyset \end{Bmatrix} (\mathit{rev\_append}\ [xs; ys])_{\mathsf{ptr}} \begin{Bmatrix} \mathit{List}\ (\frac{1}{2}\mathit{vps})\ \mathit{xs} \\ \lambda zs.\ \mathit{List}\ (\mathit{rev}\ (\frac{1}{2}\mathit{vps}) +\!\!+\ \mathit{wqs})\ zs \\ zs \hookleftarrow_1 \emptyset \end{Bmatrix}$$

$$\begin{Bmatrix} \mathit{List}\ \mathit{vps}\ \mathit{xs}\ *\ \mathit{xs} \hookleftarrow_1 \emptyset \\ \mathit{List}\ \mathit{wqs}\ \mathit{ys}\ *\ \mathit{ys} \hookleftarrow_1 \emptyset \end{Bmatrix} (\mathit{rev\_append}\ [xs; ys])_{\mathsf{ptr}} \begin{Bmatrix} \lambda zs.\ \begin{matrix} \mathit{List}\ (\mathit{rev}\ \mathit{vps} +\!\!+\ \mathit{wqs})\ zs \\ zs \hookleftarrow_1 \emptyset\ *\ \diamond 1 \end{matrix} \end{Bmatrix}$$

Fig. 12. Two specifications for *rev_append*

```
$append \triangleq \mu_{\mathsf{ptr}}\_.\ \lambda[xs; ys].$
  let $aux = \mu_{\mathsf{clo}} self.\ \lambda[xs; k].$
    if $(is\_nil\ [xs])_{\mathsf{ptr}}$ then $(k\ [ys])_{\mathsf{clo}}$ else
      let $x = (head\ [xs])_{\mathsf{ptr}}$ in
      let $xs' = (tail\ [xs])_{\mathsf{ptr}}$ in
      let $k' = \mu_{\mathsf{clo}}\_.\ \lambda[r].$
        let $p = (cons\ [x; r])_{\mathsf{ptr}}$ in $(k\ [p])_{\mathsf{clo}}$ in
      $(self\ [xs'; k'])_{\mathsf{clo}}$ in
  let $id = \mu_{\mathsf{clo}}\_.\ \lambda[x].\ x$ in
  $(aux\ [xs; id])_{\mathsf{clo}}$
```

```
$mk\_counter \triangleq \mu_{\mathsf{ptr}}\_.\ \lambda[].$
  let $c = (ref\ [0])_{\mathsf{ptr}}$ in
  let $i = \mu_{\mathsf{clo}}\_.\ \lambda[].\ (incr\ [c])_{\mathsf{ptr}}$ in
  let $g = \mu_{\mathsf{clo}}\_.\ \lambda[].\ (get\ [c])_{\mathsf{ptr}}$ in
  $\{\, i\ , g \,\}$
```

Fig. 13. Linked list concatenation in continuation-passing style, and a "Counter" object

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 $ys \hookleftarrow_1 \emptyset$. 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 second specification describes the output list as $\mathit{List}\ (\mathit{rev}\ \mathit{vps} +\!\!+\ \mathit{wqs})\ zs$. It also contains one space credit: this is the size of the "nil" block that terminates the list *xs*, which is freed. 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: $\frac{1}{2}\mathit{vps}$ 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

$$\langle\{xs\}\rangle \left\{\begin{matrix} \diamond(2\times 3\times|vps|+3) \\ \mathit{List}\ vps\ xs \\ \mathit{List}\ wqs\ ys * ys \hookleftarrow_1 \emptyset \end{matrix}\right\} (\mathit{append}\ [xs; ys])_{\mathsf{ptr}} \left\{\lambda zs.\begin{matrix} \diamond(3\times|vps|+3) \\ \mathit{List}\ \tfrac{1}{2} vps\ xs \\ \mathit{List}\ (\tfrac{1}{2} vps +\!\!+\ wqs)\ zs \\ zs \hookleftarrow_1 \emptyset \end{matrix}\right\}$$

$$\left\{\begin{matrix} \diamond(3\times|vps|+3) \\ \mathit{List}\ vps\ xs * xs \hookleftarrow_1 \emptyset \\ \mathit{List}\ wqs\ ys * ys \hookleftarrow_1 \emptyset \end{matrix}\right\} (\mathit{append}\ [xs; ys])_{\mathsf{ptr}} \left\{\lambda zs.\begin{matrix} \diamond(3\times|vps|+4) \\ \mathit{List}\ (vps +\!\!+\ wqs)\ zs \\ zs \hookleftarrow_1 \emptyset \end{matrix}\right\}$$

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

$$\{\diamond 7\}\ (\mathit{mk\_counter}\ [\,])_{\mathsf{ptr}} \left\{\lambda \ell.\ \exists i,g.\ \begin{matrix} \ell \mapsto [i;g] * \ell \hookleftarrow_1 \emptyset \\ i \hookleftarrow_1 \{\ell\} * g \hookleftarrow_1 \{\ell\} \\ \mathit{Counter}\, 0\, i\, g \end{matrix}\right\}$$

$$\{\mathit{Counter}\, n\, i\, g\} \qquad (i\ [\,])_{\mathsf{clo}} \qquad \{\lambda\_.\ \mathit{Counter}\,(n+1)\, i\, g\}$$

$$\{\mathit{Counter}\, n\, i\, g\} \qquad (g\ [\,])_{\mathsf{clo}} \qquad \{\lambda m.\ \ulcorner m = n \urcorner * \mathit{Counter}\, n\, i\, g\}$$

$$\left(\begin{matrix} \mathit{Counter}\, n\, i\, g \\ i \hookleftarrow_1 \emptyset * g \hookleftarrow_1 \emptyset \end{matrix}\right) \qquad \Rrightarrow^{\top}_{\emptyset} \qquad (\diamond 5)$$

Fig. 15. An interface for a "Counter" object

front of the linked list produced by the concatenation of xs' and ys. The closure k' captures the values of k, x and xs. After allocating this closure, *aux* invokes itself with arguments xs' and k'.

Like *rev_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.

Figure 13 presents the code. The toplevel function *mk_counter* allocates and returns a fresh "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.

$$\left\{\diamond A\right\} \quad (\mathit{create}\ [\,])_{\mathsf{ptr}} \quad \left\{\lambda \ell.\ \begin{array}{c}\mathit{Stack}\ [\,]\ \ell \\ \ell \hookleftarrow_1 \emptyset\end{array}\right\}$$

$$\langle\, \{\ell\}\,\rangle \left\{\begin{array}{c}\ulcorner |\mathit{vps}| < C \urcorner \\ \mathit{Stack}\ \mathit{vps}\ \ell \\ \diamond B \ * \ v \hookleftarrow_p \emptyset\end{array}\right\} (\mathit{push}\ [v; \ell])_{\mathsf{ptr}} \ \left\{\lambda\_.\ \mathit{Stack}\ ((v,p) :: \mathit{vps})\ \ell\right\}$$

$$\langle\, \{\ell\}\,\rangle \left\{\mathit{Stack}\ ((v,p) :: \mathit{vps})\ \ell\right\} \quad (\mathit{pop}\ [\ell])_{\mathsf{ptr}} \quad \left\{\lambda v.\ \begin{array}{c}\mathit{Stack}\ \mathit{vps}\ \ell \\ \diamond B \ * \ v \hookleftarrow_p \emptyset\end{array}\right\}$$

$$\left(\begin{array}{c}\mathit{Stack}\ \mathit{vps}\ \ell \\ \ell \hookleftarrow_1 \emptyset\end{array}\right) \qquad \Rrightarrow^{\top}_{\emptyset} \qquad \left(\begin{array}{c}\diamond(A + B \times |\mathit{vps}|) \\ \mathop{\Huge *}_{(v,p)\in \mathit{vps}} \ (v \hookleftarrow_p \emptyset)\end{array}\right)$$

Fig. 16. An interface for possibly-bounded stacks

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 $\mathit{Counter}\ 0\ i\ g$ 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 ℓ 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 $(i\ \ [\,])_{\mathsf{clo}}$ and $(g\ \ [\,])_{\mathsf{clo}}$ require the assertion $\mathit{Counter}\ n\ i\ g$. The former updates this assertion to $\mathit{Counter}\ (n+1)\ i\ g$, 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 $\mathit{Counter}\ n\ i\ g$ 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 $\mathit{Stack}\ \mathit{vps}\ \ell$, which asserts that at address ℓ 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 is $+\infty$. 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 $A + B \times |\mathit{vps}|$ space credits, where $|\mathit{vps}|$ 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 λ-calculus. Hofmann [2000] and Aspinall and Hofmann [2002] adapt the idea to LFPL, a first-order functional programming language without GC and with explicit destructive pattern matching. There, a value of type $\diamond$ exists at runtime and can be understood as a pointer to a free block in the heap. Subsequent work aims at automating space complexity analyses. In particular, Hofmann and Jost [2003] propose an affine type system where types carry space credits. Hofmann and Jost [2006]; Hofmann and Rodriguez [2009, 2013] analyze a variant of Java where garbage collection has been replaced with explicit deallocation. RaML [Hoffmann et al. 2012a,b, 2017] analyzes a fragment of OCaml, also without GC and with explicit destructive pattern matching. Niu and Hoffmann [2018] present a type-based amortized space analysis for a pure, first-order programming language where destructive pattern matching can be applied to shared objects, an unusual feature. Their system performs significant over-approximations: when a data structure becomes shared, the logic charges the cost of creating a copy of this data structure. As far as we understand, this analysis can be used to reason in a sound yet very conservative way about a programming language with GC.

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 indicates 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 λ-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 λ-calculus into itself. This calculus is equipped with two different environment-based big-step operational semantics. The "source" semantics implicitly constructs a closure for each function definition by capturing the relevant part of the environment and storing it in the heap. The "target" semantics performs no such construction: it requires every function to be closed. In either semantics, the roots are defined as the locations that occur in the environment. Up to the stylistic difference between a substitution-based semantics and an environment-based semantics, this definition is equivalent to the "free variable rule" [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 ephemerons [Hayes 1997]; and refining our cost model to account for certain compiler optimizations, such as the static allocation of structured constants.

REFERENCES

Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis (Lecture Notes in Computer Science, Vol. 8552)*. Springer, 1–18.

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

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

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

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

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

Wayne D. Blizard. 1990. Negative membership. *Notre Dame Journal of Formal Logic* 31, 3 (1990), 346 – 368. https://doi.org/10.1305/ndjfl/1093635499

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

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

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

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation (PLDI)*. 270–281.

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

Arthur Charguéraud and François Pottier. 2017. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* (Sept. 2017).

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

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

Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Lecture Notes in Computer Science, Vol. 7850. Springer, 15–58.

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

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

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

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

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

Theodore Hailperin. 1986. Chapter 2 Formalization of Boole's Logic. In *Boole's Logic and Probability*. Studies in Logic and the Foundations of Mathematics, Vol. 85. Elsevier, 135–172. https://doi.org/10.1016/S0049-237X(08)70247-7

Joseph J. Hallet and Assaf Kfoury. 2005. *A Formal Semantics for Weak References*. Technical Report. Boston University.

Maximilian P. L. Haslbeck and Peter Lammich. 2021. For a Few Dollars More - Verified Fine-Grained Algorithm Analysis Down to LLVM. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 12648)*. Springer, 292–319.

Maximilian P. L. Haslbeck and Tobias Nipkow. 2018. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 155–171.

Barry Hayes. 1997. Ephemerons: A New Finalization Mechanism. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 176–183. https://doi.org/10.1145/263698.263733

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Daniel Loeb. 1992. Sets with a negative number of elements. *Advances in Mathematics* 91, 1 (1992), 64–74. https://doi.org/10.1016/0001-8708(92)90011-9

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

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022a. A High-Level Separation Logic for Heap Space under Garbage Collection (Extended Version). https://hal.inria.fr/hal-03823056.

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022b. A High-Level Separation Logic for Heap Space under Garbage Collection (Proof Artifact). https://gitlab.inria.fr/amoine/spacelambda, DOI: https://doi.org/10.5281/zenodo.7129301.

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

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3519939.3523432

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

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

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

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

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

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

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

Received 2022-07-07; accepted 2022-11-07