A Separation Logic for Heap Space under Garbage Collection

Jean-Marie Madiot, François Pottier

A Separation Logic for Heap Space under Garbage Collection

JEAN-MARIE MADIOT, Inria, France
FRANÇOIS POTTIER, Inria, France

We present SL⋄, a Separation Logic that allows controlling the heap space consumption of a program in the presence of dynamic memory allocation and garbage collection. A user of the logic works with *space credits*, a resource that is consumed when an object is allocated and produced when a group of objects is *logically deallocated*, that is, when the user is able to prove that it has become unreachable and therefore can be collected. To prove such a fact, the user maintains *pointed-by* assertions that record the immediate predecessors of every object. Our calculus, SpaceLang, has mutable state, shared-memory concurrency, and code pointers. We prove that SL⋄ is sound and present several simple examples of its use.

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:
Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proc. ACM Program. Lang.* 6, POPL, Article 11 (January 2022), 28 pages. https://doi.org/10.1145/3498672

1 INTRODUCTION

In a world where software can be found in almost every kind of device, including critical embedded systems, mobile phones, and desktop computers, it is desirable to statically guarantee not only that this software will run safely and behaves in a correct manner, but also that it will not attempt to consume more resources than expected. Although one can think of many kinds of resources, including electric energy, money, and more, two of the most basic computational resources are time and space. One must distinguish several kinds of space, or storage areas, that are managed in different ways: these include stack space, heap space, disk space, and more. In this paper, we are specifically interested in reasoning about *heap space usage* in the presence of *tracing garbage collection*, a widely used technique for automatic reclamation of unused memory. The problem is challenging because it is not evident in the code where memory can be reclaimed: there is no memory deallocation instruction. Instead, objects implicitly become eligible for deallocation once they are unreachable. The problem is to statically determine *what data is live* at each program point and to statically predict and control the size of this live data. We do not keep track of stack space; others have done so before [Carbonneaux et al. 2014].

To analyze and control the space usage of programs, many techniques have been developed, including static analyses, type systems, and program logics. A fraction of this rich literature is reviewed in §7. In this paper, we aim to develop a program logic, that is, a set of program verification rules. Naturally, these rules must be sound: a verified software component should never consume more space than advertised by its specification. They must be compositional: we wish to verify software components independently, while being assured that verified components can be safely

Authors' addresses: Jean-Marie Madiot, Inria, Paris, France, jean-marie.madiot@inria.fr; François Pottier, Inria, Paris, France, francois.pottier@inria.fr.

2475-1421/2022/1-ART11
https://doi.org/10.1145/3498672

linked. Finally, they should be expressive: although this is an ill-defined goal, we would like to be able to reproduce the informal reasoning patterns that programmers commonly use. As an example, we wish to be able to state and prove that a list copy (§6.1) or list reversal function normally requires $2 + 4n$ words of available space, where n is the length of the original list. More subtly, in case no pointer to the original list is retained, we wish to argue that no available space is in fact required, as the new list cells can occupy the space that is freed up as the original list cells become unreachable. As another example (§6.2), we wish to be able to state and prove that creating an empty stack and pushing an element onto this stack each require 4 words of memory, that popping an element off the stack frees up 4 words, and that letting go of the last pointer to the stack frees up $4 + 4n$ words of memory, where n is the number of elements in the stack. We also need to keep track of the fact that pushing an element onto the stack creates a pointer from the stack to the element, and that popping this element off the stack (or abandoning the stack altogether) destroys this pointer.

We base our work on Separation Logic [Brookes and O'Hearn 2016; O'Hearn 2019; Reynolds 2002] and its modern variants, specifically Iris [Jung et al. 2018], because they are natural and powerful frameworks in which one can reason about programs. The question is, how can one exploit Separation Logic to reason about the amount of available space in the heap?

A simple idea, inspired by Hofmann [2000], is to extend Separation Logic with a *space credit* $\diamond 1$, an assertion that guarantees the existence of one word of free space in the heap, and can therefore be understood as the ownership of one word of free space, or as a permission to allocate one word. In a programming language *without garbage collection*, where memory allocation and deallocation instructions are explicit in the source code, one can express the idea that allocating a block b consumes $size(b)$ space credits, whereas, symmetrically, deallocating this block produces $size(b)$ credits, like so:

$$\{\diamond size(b)\}\ x := \mathsf{alloc}(b)\ \{x \mapsto b\} \qquad \textsc{(BasicAlloc)}$$
$$\{x \mapsto b\}\quad \mathsf{free}(x)\quad \{\diamond size(b)\} \qquad \textsc{(BasicFree)}$$

In such a variant of Separation Logic, the precondition and postcondition of a program (or program fragment) reveal how much free space it requires and how much free space it leaves in the heap. This is what we desire: a compositional program logic, where the space requirements of a (sub)program are explicit in its specification.

In the presence of garbage collection, what can be said? Certainly, ordinary Separation Logic can be and has been used in such a setting [Charguéraud 2020; Jung et al. 2018]: it is just a matter of removing "free" from the programming language and removing the corresponding reasoning rule from the program logic. However, by abandoning explicit control over memory deallocation, we seem to lose the ability to determine where memory deallocation takes place. Certainly, we lose the ability to recover space credits that is inherent in the rule BasicFree above.

To remedy this problem, a second idea comes to mind: abandon the instruction "free", but keep a variant of the reasoning rule BasicFree, along the following lines:

$$x \mapsto b \ \Rrightarrow\ \diamond size(b) \qquad \textsc{(FlawedLogicalFree)}$$

The logical connective $\Rrightarrow$ can be informally understood as a ghost update. Equipped with this reasoning rule, the programmer can *logically deallocate* a memory block, that is, highlight a point in the code where this block becomes unused. Applying the update consumes the assertion $x \mapsto b$ and produces the assertion $\diamond size(b)$ in its place. The point where the update is applied is not visible in the code, but is explicit in the Separation Logic derivation that the programmer builds while verifying the code. Thus, we envision a hybrid approach that marries *manual reasoning about memory management at program-verification time* and *automatic memory management at runtime*. Several questions spring to mind: is this approach practical? Is it sound?

Is this approach practical? The reader might fear that we get the worst of both worlds, that is, the complexity and mental burden that often accompany manual memory management, together with the latency issues sometimes caused at runtime by garbage collection. As we shall see, the need to reason about memory deallocation does indeed lead to reasoning rules, specifications, and proofs that are more complex than usual. This is a risk that we are running, and it is up to us to show that this added complication is tolerable and manageable at scale. On the upside, we get access to the simpler programming model and potentially superior performance afforded by garbage collection. We also get access, at program-verification time, to powerful mechanisms such as *bulk logical deallocation*, which consists in applying a variant of the reasoning rule FlawedLogicalFree to a group of memory blocks at once. Indeed, nothing forces us to logically deallocate only one block at a time; it is up to us to imagine a variety of fine-grained and coarse-grained policies for memory management at program-verification time.

Is this approach sound? It does inherit several valuable properties of Separation Logic. Because the reasoning rule FlawedLogicalFree consumes the "points-to" assertion $x \mapsto b$, a memory block cannot be logically deallocated twice, and cannot be accessed after it has been logically deallocated. Unfortunately, this is insufficient. The approach, as sketched so far, is not sound. Indeed, by introducing the idea of logical deallocation, we create a distinction between the *physical heap*, which exists at runtime and where memory is freed by the garbage collector, and the *logical heap*, which is the heap that the programmer has in mind at program-verification time and where memory is freed by applying the rule FlawedLogicalFree. The assertion $\diamond 1$ guarantees the existence of one word of free space in the logical heap. Does this imply the existence of one word of free space in the physical heap as well? In general, no. This is the case only if the programmer uses logical deallocation wisely: when a memory block is logically deallocated, it should be the case that this block is inaccessible, that is, unreachable. This guarantees that the garbage collector, too, can deallocate this block when it so wishes. This allows us to maintain the invariant that *every block that has been logically deallocated has been or can be physically deallocated.* This invariant implies that *if there are k words of free space in the logical heap, then the garbage collector can find or create k words of free space in the physical heap.* Then, the assertion $\diamond 1$ *can* be interpreted as a permission to allocate one word without risking to run out of memory.

The outstanding problem, thus, is to restrict the reasoning rule FlawedLogicalFree so as to require that the memory block b be inaccessible: there must exist no path from the roots, through the heap, to this block. The roots, roughly speaking, are the local variables of all ongoing procedure calls; they are found in the stack, or (if there are multiple threads) in each thread's stack.

This might seem a difficult problem, which Separation Logic might be ill-equipped to solve. Indeed, by design, Separation Logic is good at reasoning about *ownership*, while the challenge here seems to be reasoning about *reachability*. These notions are in general unrelated. It is possible to own a block that has become unreachable, and this is indeed the case when FlawedLogicalFree is about to be applied in a correct manner. Conversely, it is possible to retain an address for which there is no access permission. Furthermore, compositional reasoning about reachability seems difficult, because reachability is nonlocal: adding or deleting a single edge in the heap can cause many memory blocks to become reachable or unreachable.

Fortunately, there is a simple way out of this problem. To prove that a block is inaccessible, it suffices to prove that it has no incoming edges, that is, no predecessors. This leads us to *extending Separation Logic to keep track of the predecessors* of each block, an idea that has appeared in the literature [Kassios and Kritikos 2013] but has not been much exploited. In addition to the ordinary "points-to" assertion $\ell \mapsto b$, which means that at address ℓ there is a memory block b, let the new "pointed-by" assertion $\ell \leftarrowtail L$ mean that all pointers to address ℓ originate in blocks whose addresses inhabit the multiset L. Then, the assertion $\ell \leftarrowtail \emptyset$ guarantees that the block at address ℓ has no

$$
\begin{array}{lllll}
\ell, c, r, s \in \mathcal{L} & & & & \\
v ::= () \mid k \mid \ell \mid \lambda\vec{x}.i & & & & \\
b ::= \vec{v} \mid \langle v \rangle \mid \lightning & \mathit{size}(\vec{v}) = 1 + |\vec{v}| & \mathit{size}(\langle v \rangle) = 0 & \mathit{size}(\lightning) = 0 & \\
\varrho ::= x \mid c & & & & \\
i ::= \mathsf{skip} & \textit{no-op} & *\varrho = \mathsf{alloc}\ n & \textit{heap allocation} & \\
\quad i; i & \textit{sequencing} & *\varrho = [*\varrho + o] & \textit{heap load} & \\
\quad \mathsf{if}\ *\varrho\ \mathsf{then}\ i\ \mathsf{else}\ i & \textit{conditional} & [*\varrho + o] = *\varrho & \textit{heap store} & \\
\quad *\varrho(\vec{\varrho}) & \textit{procedure call} & *\varrho = (*\varrho == *\varrho) & \textit{address comparison} & \\
\quad *\varrho = v & \textit{constant load} & \mathsf{alloca}\ x\ \mathsf{in}\ i & \textit{stack allocation} & \\
\quad *\varrho = *\varrho & \textit{move} & \mathsf{alloca}\ c\ \mathsf{in}\ i & \textit{stack allocation (in progress)} & \\
 & & \mathsf{fork}\ *\varrho\ \mathsf{as}\ x\ \mathsf{in}\ i & \textit{thread creation} & \\
K ::= [] \mid K; i \mid \mathsf{alloca}\ c\ \mathsf{in}\ K & & & &
\end{array}
$$

Fig. 1. Syntax

predecessors, therefore is inaccessible. Logical deallocation can then be soundly expressed like so:

$$x \mapsto b \star x \hookleftarrow \emptyset \;\Rrightarrow\; \diamond \mathit{size}(b) \qquad \text{(BasicLogicalFree)}$$

Although we shall improve this rule later on, the main ideas are present already. A variant of this rule that allows bulk logical deallocation can also be given. There, it is not necessary to require every block in the group to have no predecessors: instead, it suffices to ensure that the group as a whole has no incoming edges, that is, that the group is closed under predecessors. This allows logically deallocating complex data structures, which may involve sharing and cycles.

The last point that remains to be stressed is that, for this approach to be sound, the predecessor-tracking logic that we build into the assertion $\ell \hookleftarrow L$ must be able to see not only the pointers from the heap into itself, but also the pointers from every stack into the heap. To this end, we choose to work with a *store* that encompasses both the heap and the stacks of all threads. A memory block in the store is either an object in the heap or a cell in some stack. In a "pointed-by" assertion $\ell \hookleftarrow L$, the elements of the multiset L are addresses of heap blocks and stack cells. This approach works extremely smoothly. The price to pay for it is that it leads us to the design of a rather low-level calculus, baptized SpaceLang, where a variable does not denote a value (as usual in a call-by-value λ-calculus), but denotes the address of a stack cell. SpaceLang features heap-allocated tuples with automatic garbage collection, mutable stack-allocated cells with explicit scoped allocation, first-class closed procedures, and shared-memory concurrency: a fork instruction spawns a new thread. It does not have procedures with free variables: adding support for closures, either directly or via a form of macro-expansion, is future work (§8).

We now present the syntax and semantics of SpaceLang (§2), followed with the reasoning rules of SL$\diamond$ (§3). We sketch the proof of soundness of SL$\diamond$ (§4), which is machine-checked [Madiot and Pottier 2021]. We present a certain style of use of SL$\diamond$, *ghost reference counting* (§5), and provide two examples that illustrate this style (§6). We review the related work (§7) and conclude (§8).

2 SYNTAX AND SEMANTICS OF SPACELANG

2.1 Values, Blocks, Stores

We fix an infinite set $\mathcal{L}$ of *memory locations*, also known as addresses. We view memory locations as abstract identifiers: one cannot convert a memory location to an integer, ask which of two locations is smaller, or ask whether two locations are adjacent. We let ℓ, c, r, s range over memory locations. By convention, we use ℓ for an address in the heap and c, r, s for addresses of stack cells.

A *store* σ is a finite map of memory locations to blocks, where a block can be a heap block, a stack cell, or a deallocated block. Thus, the store encompasses the heap and the stacks. (In a concurrent setting, there is one stack per thread.) Such a unified representation of the store is convenient, as it contains enough information to define the action of the garbage collector (§2.4). In particular, the roots are easily identified: they are the stack cells.

The syntax of *blocks* appears on the second line of Figure 1. A heap block $\vec{v}$ is a tuple: it stores a sequence $\vec{v}$ of values. A stack cell $\langle v \rangle$ stores a single value v. The special block $\lightning$ denotes a deallocated block. Our operational semantics does not recycle memory locations:[1] when a heap block or stack cell at address ℓ is deallocated, the store is updated with a mapping of ℓ to $\lightning$.

The syntax of *values* appears on the first line of Figure 1. A value is a piece of data that fits in one word of memory. The values include the unit value $()$, integer values k, memory locations ℓ, and procedures $\lambda\vec{x}.i$. A value of the form $\lambda\vec{x}.i$ is a closed procedure whose formal parameters are the variables $\vec{x}$ and whose body is the instruction i. It can be thought of as a code pointer.

Every memory block b has a size $\mathit{size}(b)$, also defined in Figure 1. The size of a heap block $\vec{v}$ is one (to account for a header word) plus the number of values stored in this block. The size of a stack cell is zero, because we are interested only in measuring the size of the heap. The size of a deallocated block is zero. The size of a store is the sum of the sizes of its blocks.

We fix a *maximum size* S, and we say that a store σ is *valid* if $\mathit{size}(\sigma) \leq S$ holds. The parameter S plays a role in the operational semantics (§2.3) and in the statement of soundness of SL$\diamond$ (§4). In the semantics, it is used to introduce the possibility of a runtime failure: an attempt to allocate a new heap block fails if this allocation would cause the store to grow too large and become invalid. Thus, by design, the semantics guarantees that the size of the store never exceeds S. The problem, therefore, is to rule out the possibility of a runtime failure. This is what the soundness statement is about: it guarantees that, if a program is correct according to SL$\diamond$, then this program is *safe*: it cannot fail at runtime. The parameter S appears in this statement, and also plays a role in its proof. Indeed, it is used at the most basic level of the proof, in the definition of the semantics of space credits. One space credit $\diamond 1$ is defined roughly as "the ownership of one unit of available space", where the available space in a store σ is defined as $S - \mathit{size}(\sigma)$. Even though S plays a role in the metatheory of SL$\diamond$, it does not appear in any of its reasoning rules. Thus, users need not know about S, and program specifications and proofs are independent of the value of S.

We define the multiset $\mathit{pointers}(v)$ as follows: if v is a memory location ℓ, then $\mathit{pointers}(v)$ is $\{\ell\}$; otherwise, it is $\emptyset$. The multiset $\mathit{pointers}(b)$ is the multiset sum of the multisets $\mathit{pointers}(v)$, where v ranges over the values stored in the block b. It represents the outgoing pointers of the block b.

2.2 References

Next in Figure 1 come the syntaxes of *references* ϱ and instructions i. A reference ϱ is either a variable x or a memory location c, which represents the address of a stack cell. SpaceLang departs from ordinary call-by-value λ-calculus, whose variables denote values, and whose syntax is stable under substitutions of values for variables. In SpaceLang, instead, a variable denotes the address of a stack cell, and the syntax of instructions is stable under substitutions of memory locations for variables. Put another way, whereas in ordinary λ-calculus the stack is entirely implicit, in SpaceLang, it is partly explicit: every stack cell has an address and exists in the store.

As a consequence of this design, there is no way for an expression to return a value. For this reason, SpaceLang has instructions instead of expressions. Similarly, it has procedures instead of

[1] In a realistic implementation, memory locations are recycled. However, we must work at a higher level of abstraction where memory locations are never reused. Indeed, the reasoning rules of Separation Logic guarantee that memory allocation always produces a fresh address. Ideally, one should prove that the semantics presented in this paper is equivalent (regarding both functional behavior and live heap space) to a semantics where memory is reused. We have not done so yet.

$$\textsc{StepSeqSkip}\quad \mathsf{skip}; i \;/\; \sigma \longrightarrow i \;/\; \sigma$$

$$\textsc{StepIf}\quad \frac{\sigma(r) = \langle k \rangle}{\mathsf{if} * r \;\mathsf{then}\; i_1 \;\mathsf{else}\; i_2 \;/\; \sigma \longrightarrow k \neq 0 \;?\; i_1 : i_2 \;/\; \sigma}$$

$$\textsc{StepCall}\quad \frac{\sigma(r) = \langle \lambda \vec{x}.i \rangle \qquad |\vec{x}| = |\vec{s}|}{*r(\vec{s}) \;/\; \sigma \longrightarrow [\vec{s}/\vec{x}]i \;/\; \sigma}$$

$$\textsc{StepConst}\quad \frac{\sigma' = \langle s := v \rangle \sigma \qquad \mathit{pointers}(v) = \emptyset}{*s = v \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepMove}\quad \frac{\sigma(r) = \langle v \rangle \qquad \sigma' = \langle s := v \rangle \sigma}{*s = *r \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepAlloc}\quad \frac{\sigma' = [\ell \mathrel{+}= ()^n]\sigma \qquad \mathit{size}(\sigma') \leq S \qquad \sigma'' = \langle s := \ell \rangle \sigma'}{*s = \mathsf{alloc}\ n \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma''}$$

$$\textsc{StepLoad}\quad \frac{\sigma(r) = \langle \ell \rangle \qquad \sigma(\ell) = \vec{v} \qquad 0 \leq o < |\vec{v}| \qquad \vec{v}(o) = v \qquad \sigma' = \langle s := v \rangle \sigma}{*s = [*r + o] \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepStore}\quad \frac{\sigma(r) = \langle v \rangle \qquad \sigma(s) = \langle \ell \rangle \qquad \sigma(\ell) = \vec{v} \qquad 0 \leq o < |\vec{v}| \qquad \sigma' = [\ell := [o := v]\vec{v}]\sigma}{[*s + o] = *r \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepLocEq}\quad \frac{\sigma(r_1) = \langle \ell_1 \rangle \qquad \sigma(r_2) = \langle \ell_2 \rangle \qquad \sigma' = \langle s := (\ell_1 = \ell_2 \;?\; 1 : 0) \rangle \sigma}{*s = (*r_1 == *r_2) \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepAllocaEntry}\quad \frac{\sigma' = [c \mathrel{+}= \langle () \rangle]\sigma}{\mathsf{alloca}\ x\ \mathsf{in}\ i \;/\; \sigma \longrightarrow \mathsf{alloca}\ c\ \mathsf{in}\ [c/x]i \;/\; \sigma'}$$

$$\textsc{StepAllocaExit}\quad \frac{\sigma(c) = \langle v \rangle \qquad \sigma' = [c := \lightning]\sigma}{\mathsf{alloca}\ c\ \mathsf{in}\ \mathsf{skip} \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma'}$$

$$\textsc{StepFork}\quad \frac{\sigma(r) = \langle v \rangle \qquad \sigma' = [r := ()][c \mathrel{+}= \langle v \rangle]\sigma}{\mathsf{fork} * r\ \mathsf{as}\ x\ \mathsf{in}\ i \;/\; \sigma \longrightarrow \mathsf{skip} \;/\; \sigma' \quad \mathit{spawning}\ \mathsf{alloca}\ c\ \mathsf{in}\ [c/x]i}$$

$$\textsc{StepContext}\quad \frac{i \;/\; \sigma \longrightarrow i' \;/\; \sigma' \qquad \mathit{spawning}\ \vec{i}}{K[i] \;/\; \sigma \longrightarrow K[i'] \;/\; \sigma' \quad \mathit{spawning}\ \vec{i}}$$

Fig. 2. Small-Step Operational Semantics, without GC

functions. Procedure calls have call-by-reference semantics: the formal parameters of a procedure denote references (addresses of stack cells), not values. This lets a procedure read and write stack cells in its caller's frame. Although this convention may seem baroque, it respects our decision that variables denote references, not values. Call-by-value can be simulated by letting the callee allocate its own local storage, copy parameters into it upon entry, and copy results out of it upon exit.

2.3 Instructions

The syntax of instructions is given in Figure 1. Their small-step operational semantics, a reduction relation of the form $i \;/\; \sigma \longrightarrow i' \;/\; \sigma'$ *spawning* $\vec{i}$, is defined in Figure 2. The last argument $\vec{i}$ is a list of newly-spawned threads; when it is empty, we omit it and write just $i \;/\; \sigma \longrightarrow i' \;/\; \sigma'$.

The no-op "skip" and the sequencing construct "$i_1; i_2$" are standard. The conditional construct "$\mathsf{if} * \varrho\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2$" tests whether the integer value stored in the stack cell ϱ is nonzero or zero. (We write $k \neq 0 \;?\; i_1 : i_2$ for a meta-level conditional.) The procedure call "$*\varrho(\vec{\varrho})$" invokes the procedure whose address is stored in the stack cell ϱ. The actual arguments $\vec{\varrho}$ are *addresses* of stack cells.

The instruction "$*\varrho = v$" writes v into the stack cell ϱ.[2] The side condition $\mathit{pointers}(v) = \emptyset$ in StepConst means that v must be a constant, that is, a unit value, an integer value, or a code pointer. It cannot be a memory location: a memory location is not a constant.

[2]We write $[\ell := b]\sigma$ for the store that maps ℓ to b and that maps every other location ℓ' to $\sigma(\ell')$. This represents either an extension or an update. For greater clarity, in Figure 2, we write $\sigma' = [\ell \mathrel{+}= b]\sigma$ as a short-hand for $\sigma' = [\ell := b]\sigma$ together with the side condition $\ell \notin \mathit{dom}(\sigma)$, which indicates that the store is extended. We write $\sigma' = \langle s := v \rangle \sigma$ as a short-hand for $\sigma' = [s := \langle v \rangle]\sigma$ together with the side condition $\sigma(s) = \langle \_ \rangle$. This indicates that an existing stack cell at address s is updated with v. The side condition prevents updating a nonexistent stack cell or turning a heap block into a stack cell.

The move instruction "$*\varrho_1 = *\varrho_2$" copies the content of the stack cell ϱ_2 into the stack cell ϱ_1.

The allocation instruction "$*\varrho$ = alloc n" allocates a new heap block with n fields, initializes them with unit values,[3] and writes its address into the stack cell ϱ. The side condition $size(\sigma') \leq S$ in StepAlloc means that this instruction fails at runtime (it is stuck) if, as a result of this allocation, the size of the heap exceeds S. The load instruction "$*\varrho_1 = [*\varrho_2 + o]$" fetches from the stack cell ϱ_2 the address of a heap block, reads this block at offset o, and stores the resulting value into the stack cell ϱ_1. The store instruction "$[*\varrho_1 + o] = *\varrho_2$" fetches from the stack cell ϱ_1 the address of a heap block, then writes into this block, at offset o, the value stored in the stack cell ϱ_2. The address comparison instruction "$*\varrho = (*\varrho_1 == *\varrho_2)$" compares the memory locations stored in the stack cells ϱ_1 and ϱ_2 and writes the result of the comparison (encoded as the integer value 0 or 1) into the stack cell ϱ.

The stack allocation construct "alloca x in i" allocates a fresh stack cell, binds the variable x to the address of this cell, and executes the instruction i. Once the execution of i terminates, the stack cell is deallocated. Technically, this process happens in three phases. First, StepAllocaEntry lets "alloca x in i" reduce to "alloca c in $[c/x]i$", where c is the address of the newly-allocated cell. Then, since "alloca c in []" is an evaluation context, StepContext allows reducing the instruction $[c/x]i$, in zero, one or more steps, to skip. Finally, StepAllocaExit deallocates the stack cell at address c. Thus, stack cells are indeed allocated and deallocated according to a stack discipline. If one wishes for a stack cell to be considered dead (that is, no longer a root) before it is deallocated, then one must explicitly kill this cell, that is, write a unit value into it.

The construct "fork $*\varrho$ as x in i" reads a value v from the stack cell ϱ, overwrites this cell with a unit value, initializes a fresh stack cell with v, and spawns a new thread that executes i under a binding of the variable x to the new stack cell. This seems the simplest way of allowing a value v to be transmitted from the original thread to the new thread. The use of a destructive read ensures that there is no point in time where both threads hold a pointer to v. This makes "fork" better-behaved and easier to reason about in terms of space usage.

2.4 Garbage Collection and Concurrency

The relation $i \mid \sigma \longrightarrow i' \mid \sigma'$ *spawning* $\vec{i}$ describes one step of computation taken by one thread. To fully describe the semantics of SpaceLang, there remains to introduce garbage collection and to allow the interleaving of steps taken by multiple threads.

We model garbage collection as a nondeterministic relation between stores $\sigma \text{ 🗑 } \sigma'$. Its definition is standard: in short, garbage collection *may* deallocate any block that is not reachable from a root.

Definition 2.1 (Terminology). Fix a store σ. With respect to this store, there is an *edge* from ℓ to ℓ' if $\sigma(\ell) = b$ and $pointers(b) \ni \ell'$. A *path* from ℓ to ℓ' is a succession of zero or more edges leading from ℓ to ℓ'. A location c is a *root* if $\sigma(c) = \langle v \rangle$. A location is *reachable* if there is a path from some root to this location.

Definition 2.2 (Garbage Collection). The relation $\sigma \text{ 🗑 } \sigma'$ holds if (1) the stores σ and σ' have the same domain and (2) for every location ℓ in this domain, either $\sigma'(\ell) = \sigma(\ell)$, or ℓ is unreachable (with respect to σ) and $\sigma'(\ell) = \lightning$.

The reduction relation is combined with garbage collection by sequential composition:

Definition 2.3 (Reduction Step with Garbage Collection). The relation $\text{🗑}\!\!\longrightarrow$ is defined as follows:

$$\frac{\sigma \text{ 🗑 } \sigma' \qquad i \mid \sigma' \longrightarrow i' \mid \sigma'' \text{ spawning } \vec{i}}{i \mid \sigma \ \text{🗑}\!\!\longrightarrow\ i' \mid \sigma'' \text{ spawning } \vec{i}}$$

[3] We write $()^n$ for a tuple of n unit values.

$$\ell \mapsto_{q_1+q_2} b \;\equiv\; \ell \mapsto_{q_1} b \star \ell \mapsto_{q_2} b \qquad \textsc{Join}{\mapsto}$$

$$v \hookleftarrow_{q_1+q_2} L_1 \uplus L_2 \;\equiv\; v \hookleftarrow_{q_1} L_1 \star v \hookleftarrow_{q_2} L_2 \qquad \textsc{Join}{\hookleftarrow}$$

$$v \hookleftarrow_q L \;\twoheadrightarrow\; v \hookleftarrow_q L' \quad \text{if } L \subseteq L' \qquad \textsc{Covariance}{\hookleftarrow}$$

$$\ell \mapsto_q b \star \ell' \hookleftarrow_1 L \;\Rrightarrow_I\; \begin{array}{l} \ell \mapsto_q b \star \ell' \hookleftarrow_1 L \star \\ \ell' \,\$\, \mathit{pointers}(b) \leq \ell \,\$\, L \end{array} \qquad \textsc{Confront}{\mapsto}{\hookleftarrow}$$

$$\mathit{True} \Rrightarrow_I \diamond 0 \qquad \textsc{Zero}\diamond$$

$$\diamond(n_1 + n_2) \;\equiv\; \diamond n_1 \star \diamond n_2 \qquad \textsc{Join}\diamond$$

$$\mathit{True} \;\equiv\; ††\emptyset \qquad \textsc{Zero}††$$

$$††(D_1 \cup D_2) \;\equiv\; ††D_1 \star ††D_2 \qquad \textsc{Join}††$$

$$\mathit{True} \;\twoheadrightarrow\; \emptyset \,☁^0\, P \qquad \textsc{CloudEmpty}$$

$$D \,☁^n\, P \star \ell \mapsto_1 \vec{v} \star \ell \hookleftarrow_1 L \;\twoheadrightarrow\; (\{\ell\} \cup D) \,☁^{n+\mathit{size}(\vec{v})}\, P \quad \text{if } \mathit{dom}(L) \subseteq P \qquad \textsc{CloudCons}$$

$$D \,☁^n\, D \Rrightarrow_I ††D \star \diamond n \qquad \textsc{Free}$$

$$v \hookleftarrow_q L \star ††D \Rrightarrow_I v \hookleftarrow_q L' \quad \text{if } \mathit{dom}(L \setminus L') \subseteq D \qquad \textsc{Cleanup}$$

Fig. 3. Selected Logical Implications, Equivalences, and Updates

Garbage collection takes place immediately before every reduction step. This confers an angelic character to the nondeterminism that is inherent in garbage collection. Indeed, the relation 🗑 is nondeterministic, while the relation $\longrightarrow$ is partial: some machine configurations $i \mid \sigma$ are unable to make a step. In particular, according to the relation $\longrightarrow$, memory allocation fails if there is not enough space in the heap. According to the relation $🗑\!\!\longrightarrow$, however, allocation fails *only if there is no way for the garbage collector to free up enough space.*

The last step in the definition of the semantics of SpaceLang is to keep track of the existence of multiple threads, sharing a single store, and to allow arbitrary interleavings of their execution steps. We omit the required definitions, which are standard: Iris provides them [Jung et al. 2018, §6.1].

3 SEPARATION LOGIC WITH SPACE CREDITS FOR SPACELANG

We now describe SL$\diamond$ at an intuitive level. A more technical soundness argument is deferred to §4.

3.1 Points-To Assertions

Following a long-established tradition [Bornat et al. 2005; Boyland 2003], we describe memory blocks via fractional points-to assertions. The assertion $\ell \mapsto_p b$ guarantees that the memory block b exists at address ℓ. If the fraction p is 1, then this assertion represents the unique ownership of this memory block, and grants read-write permission; otherwise, it represents shared ownership, and grants read permission only. Points-to assertions are split and joined in the usual way; this is expressed by the law Join$\mapsto$ in Figure 3. As usual, $\ell \mapsto b$ is short for $\ell \mapsto_1 b$.

Points-to assertions describe both heap blocks and stack cells. Thus, $\ell \mapsto \vec{v}$ describes a heap block, while $c \mapsto \langle v \rangle$ describes a stack cell. The assertion $\ell \mapsto \lightning$, which represents the unique ownership of a deallocated block, is technically well-formed, but is not used. Instead, we use a persistent assertion $††D$ (§3.5) to witness the fact that the locations in the set D have been deallocated.

3.2 Space Credits

To reason about space, we use *space credits*. The assertion $\diamond n$ guarantees that there exist n words of available space in the heap. More precisely, it guarantees that the garbage collector can reclaim

enough space so as to make n words of memory available. Space credits are not duplicable: $\diamond 1$ does not entail $\diamond 1 \star \diamond 1$. They are split and joined in an additive way (Zero$\diamond$ and Join$\diamond$, Figure 3), so $\diamond 1 \star \diamond 1$ is in fact equivalent to $\diamond 2$.

The assertion $\diamond n$ can be interpreted as the unique ownership of n words of available space, or as a permission to allocate an object of size n. Indeed, the reasoning rule for dynamic memory allocation, Alloc (Figure 4), consumes $size(b)$ space credits, where b is the newly-allocated block.

Conversely, the reasoning rule for deallocation, Free (Figure 3), produces n space credits when a group of objects whose total size is n is logically deallocated. To explain this rule, we must first introduce more concepts. Indeed, to prove that a group of objects can be deallocated, the user must argue that these objects are unreachable. To do so, the user must reason about the pointers to these objects that exist in the store. To enable such reasoning, we use *pointed-by* assertions.

3.3 Pointed-By Assertions

Points-to assertions yield a forward view of the heap as a graph. The points-to assertion $\ell \mapsto b$ describes the outgoing edges of the location ℓ: for every location $\ell' \in pointers(b)$, there is an edge from ℓ to ℓ'. We introduce *pointed-by* assertions in order to maintain a view of the heap as a graph in the reverse direction. The assertion $\ell' \hookleftarrow L$ means that the predecessors of the location ℓ' inhabit the multiset L. Thus, for every $\ell \in L$, there may be an edge from ℓ to ℓ'. More crucially, for every $\ell \notin L$, *there definitely exists no edge* from ℓ to ℓ'. In particular, the assertion $\ell' \hookleftarrow \emptyset$ guarantees that the block at address ℓ' has no predecessors.

Several design choices come up regarding pointed-by assertions. One such choice is whether to keep track of predecessors using a set or a multiset. We choose multisets because they lead to simpler reasoning rules. Another choice is whether, in the assertion $\ell' \hookleftarrow L$, the multiset L should represent exactly the predecessors of ℓ', or possibly an over-approximation. We choose the latter possibility, and admit the reasoning rule Covariance$\hookleftarrow$ in Figure 3, because this makes the system slightly more flexible and convenient. A third design choice is whether one should be able to split pointed-by assertions. It is technically easy to allow this, by annotating them with fractions, and we believe that this could be important in practice, although we do not yet have evidence of this. Thus, we introduce fractional pointed-by assertions of the form $\ell' \hookleftarrow_q L$, and allow them to be split and joined via the rule Join$\hookleftarrow$ in Figure 3. This rule involves multiset sum. Thus, while the assertion $\ell' \hookleftarrow_1 L$ guarantees that the predecessors of ℓ' form a subset of L, the fractional assertion $\ell' \hookleftarrow_q L$, alone, does not allow making such a claim. One might therefore conclude that such a fractional assertion is of little use by itself. That is not the case: indeed, this assertion still allows recording the creation or deletion of a predecessor of ℓ'. This is exploited in several of the reasoning rules in Figure 4, such as Load and Store. These rules are explained later on (§3.6).

The "forward" and "reverse" views of the heap represented by points-to and pointed-by assertions are always consistent with one another. This is reflected by Confront$\mapsto\hookleftarrow$ in Figure 3. There, the points-to assertion $\ell \mapsto_q b$ indicates how many edges from ℓ to ℓ' exist: their number is the multiplicity of ℓ' in $pointers(b)$. (We write $\ell \,\$\, L$ for the multiplicity of ℓ in the multiset L.) The pointed-by assertion $\ell' \hookleftarrow_1 L$ guarantees that the incoming edges of ℓ' are contained (and counted) in the multiset L. Thus, the multiplicity of ℓ in L must be at least the multiplicity of ℓ' in $pointers(b)$.

In many of the reasoning rules, we need pointed-by assertions of the form $v \hookleftarrow_q L$, where v is a value. This assertion is defined in a straightforward way, as follows: if v is a memory location ℓ', then it is synonymous with $\ell' \hookleftarrow_q L$; otherwise, it is synonymous with $True$.

3.4 Memory Deallocation as a Ghost Operation

SpaceLang does not have a memory deallocation instruction. Nevertheless, the user must reason about the points where an object (or a data structure) becomes eligible for deallocation. For instance,

the user might reason, "at this point in the code, this list of length n becomes unreachable, therefore $4 + 4n$ words of memory can be reclaimed by the garbage collector".

In the formal setting of SL$\diamond$, this suggests viewing memory deallocation as a *ghost operation*, also known as a *ghost state update*. Such an operation has no runtime effect, but transforms an assertion into a different assertion. For instance, it may consume a permission to access a certain group of objects and produce a number of space credits. This operation takes the form of a reasoning rule, namely rule Free in Figure 3, which is explicitly used by the person who verifies a program.

There is no connection between the point where this ghost operation is used and the point where the garbage collector actually reclaims memory. An object can be reclaimed by the garbage collector either before or after the point where the user reasons that this object has become unreachable. This creates a difference between the *physical store* that exists at runtime and the *logical store* that the user has in mind. In the physical store, memory is deallocated by the garbage collector at somewhat unpredictable times. In the logical store, memory is explicitly deallocated by the user who applies Free. Still, the program logic is designed in such a way that the physical and logical stores remain closely related: *their reachable fragments coincide* (§4), so the differences between them concern unreachable objects only.

Let us now explain the reasoning rule Free, whose statement is $D \mathrel{☁^n} D \Rrightarrow_I ⨅ D \star \diamond n$. Although perhaps cryptic at first sight, this statement is actually fairly simple. It can be read as follows:

> *If a set of heap objects D is closed under predecessors and has total size n, then, by abandoning the ownership of these objects, one obtains a witness $⨅ D$ that these objects have been deallocated, together with n space credits.*

We explain deallocation witnesses in the next subsection (§3.5). The logical connective $\Rrightarrow_I$ is a variant of Iris's ghost state update [Jung et al. 2018, §7.2]; it allows transforming its left-hand side into its right-hand side.[4] Let us now explain the *cloud*, whose general form is $D \mathrel{☁^n} P$, where D and P are sets of locations. In short, this assertion: (1) guarantees that every object in D is a heap object and represents the full ownership of this object, including points-to and pointed-by assertions; (2) guarantees that all predecessors of the objects in D lie in the set P; and (3) indicates that the total size of the objects in D is n.

The case where the sets D and P coincide is of particular significance. The assertion $D \mathrel{☁^n} D$ guarantees that the set D is closed under predecessors. It also guarantees that no object in D is a stack cell, that is, no object in D is a root. Therefore, every object in D must be unreachable. This explains why it is sound to *logically deallocate* these objects and obtain n space credits in return.

The predicate $D \mathrel{☁^n} P$ is inductively defined. The reasoning rules CloudEmpty and CloudCons in Figure 3 correspond to the two constructors, and allow constructing clouds of arbitrary size.[5] By combining CloudEmpty, CloudCons, and Free, one obtains a simplified reasoning rule that allows deallocating a single heap object:

$$\ell \mapsto_1 \vec{v} \star \ell \hookleftarrow_1 L \star dom(L) \subseteq \{\ell\} \Rrightarrow_I ⨅ \{\ell\} \star \diamond size(\vec{v}) \qquad \textsc{FreeSingleton}$$

That is, if we have full ownership of the memory block $\vec{v}$ at address ℓ, including points-to and pointed-by assertions, and if this block has no predecessors (except possibly itself), then this block can be deallocated, producing n space credits, where n is the size of the block.

A careful reader may wonder why, when an object at address ℓ is deallocated, we do not immediately remove ℓ from the predecessors of every successor ℓ' of ℓ. The answer is, we do allow such a removal, but, for greater simplicity and flexibility, we allow it to be deferred. This is made possible by the deallocation witness $⨅ \{\ell\}$, or in the general case, $⨅ D$. We explain this next.

[4] $P \Rrightarrow_I Q$ is defined as $\forall \sigma.\ P \star I\ \sigma \Rrightarrow Q \star I\ \sigma$, where $I\ \sigma$ is the state interpretation invariant (Definition 4.8).

[5] In CloudCons and Cleanup, $dom(\cdot)$ maps a multiset to its domain, a set. L denotes a multiset; D and P denote sets.

3.5 Deallocation Witnesses and Deferred Predecessor Deletion

Deallocating a set D of memory locations produces a witness $\dagger\dagger D$ that the locations in this set have been deallocated. This is a persistent assertion: once it holds, it holds forever. (As noted in §2.1, memory locations are not recycled.) Deallocation witnesses can be split and joined, if necessary, per the rules Zero†† and Join†† in Figure 3.

Deallocation witnesses are exploited by the reasoning rule Cleanup, which states that deallocated locations can be removed from predecessor multisets. This rule transforms $v \hookleftarrow_q L$ into $v \hookleftarrow_q L'$, where the multiset L' is contained in the multiset L, provided every location in $L \setminus L'$ has been deallocated.[5] This allows deferred deletion of predecessors: deallocating a location ℓ and removing ℓ from the predecessors of some other location ℓ' are two separate steps.

Removing deallocated locations from predecessor multisets is beneficial because it helps satisfy the requirements of rule Free. Indeed, Free ostensibly requires proving that "every predecessor of an object in D is in D". Thanks to Cleanup, what is really required by Free is to check that "every predecessor of an object in D either has been deallocated already or is in D".

3.6 Reasoning Rules for Instructions

As is usual in Separation Logic, the reasoning rules of the program logic allow deriving Hoare triples of the form $\{\Phi\}\ i\ \{\Psi\}$, where the precondition Φ and the postcondition Ψ are assertions, and where i is an instruction. Such a triple receives a standard interpretation as a statement of partial correctness: the assertion $\{\Phi\}\ i\ \{\Psi\}$ means that, in a state where Φ holds, it is safe to execute the instruction i, and if this execution terminates, then it leads to a final state where Ψ holds. A more precise statement is given in the next section (§4).

In light of the description given in the previous subsections (§3.1–§3.5), the reasoning rules, which appear in Figure 4, should seem relatively straightforward, if somewhat verbose. This verbosity stems from two main causes: (1) the fact that every instruction takes its parameters in stack cells means that points-to assertions $r \mapsto \langle v \rangle$ must appear in pre- and postconditions; (2) keeping track of predecessors requires pointed-by assertions $v \hookleftarrow_q L$ to appear in pre- and postconditions. Let us now review the reasoning rules.

Control Constructs. The reasoning rules for sequences (Skip, Seq), conditionals (If), and procedure calls (Call) are straightforward. In If, the stack cell r is required to contain an integer value k, and the condition $k \neq 0$ determines which branch is executed. In Call, the stack cell r is required to contain a procedure $\lambda\vec{x}.i$ whose arity matches the arity of the call.

Memory Allocation in the Heap. The reasoning rule for dynamic memory allocation, Alloc, is more difficult to read; several aspects must be disentangled. The most interesting aspect is that allocation consumes a number of space credits: $\diamond size(b)$ appears in the precondition, where b is the newly-allocated block. This new block is allocated at a fresh address ℓ, for which a points-to assertion $\ell \mapsto b$ and a pointed-by assertion $\ell \hookleftarrow \{s\}$ appear in the postcondition. The latter assertion reflects the fact that the new block has exactly one predecessor, namely the stack cell s. The rest is administrative boilerplate. The points-to assertions $s \mapsto \langle v \rangle$ and $s \mapsto \langle \ell \rangle$ in the pre- and postcondition reflect the manner in which the stack cell s is updated. The pointed-by assertions $v \hookleftarrow_q L$ and $v \hookleftarrow_q L \setminus \{s\}$ in the pre- and postcondition reflect the fact that the value v that was *previously* stored in the stack cell loses one predecessor, namely s: this is an *edge deletion.*

Edge Deletion. Several remarks about edge deletion are in order. First, in the potentially common case where the value v is not a memory location, both $v \hookleftarrow_q L$ and $v \hookleftarrow_q L \setminus \{s\}$ are equivalent to *True* (§3.3), so these assertions vanish: no edge deletion takes place.

$$\textsc{Skip}\quad \{\Psi\}\ \mathsf{skip}\ \{\Psi\}$$

$$\textsc{Seq}\quad \frac{\{\Phi\}\ i_1\ \{\chi\} \qquad \{\chi\}\ i_2\ \{\Psi\}}{\{\Phi\}\ (i_1; i_2)\ \{\Psi\}}$$

$$\textsc{If}\quad \frac{\{r \mapsto \langle k\rangle \star \Phi\}\ k \neq 0\ ?\ i_1 : i_2\ \{\Psi\}}{\{r \mapsto \langle k\rangle \star \Phi\}\ \mathsf{if}\ *r\ \mathsf{then}\ i_1\ \mathsf{else}\ i_2\ \{\Psi\}}$$

$$\textsc{Call}\quad \frac{|\vec{x}| = |\vec{s}| \qquad \{r \mapsto \langle \lambda\vec{x}.i\rangle \star \Phi\}\ [\vec{s}/\vec{x}]i\ \{\Psi\}}{\{r \mapsto \langle \lambda\vec{x}.i\rangle \star \Phi\}\ *r(\vec{s})\ \{\Psi\}}$$

$$\textsc{Const}\quad \frac{\mathit{pointers}(v') = \emptyset}{\begin{Bmatrix} s \mapsto \langle v\rangle \\ v \hookleftarrow_q L \end{Bmatrix}\ *s = v'\ \begin{Bmatrix} s \mapsto \langle v'\rangle \\ v \hookleftarrow_q L \setminus \{s\} \end{Bmatrix}}$$

$$\textsc{Move}\quad \begin{Bmatrix} s \mapsto \langle v\rangle \\ r \mapsto \langle v'\rangle \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{Bmatrix} *s = *r \begin{Bmatrix} s \mapsto \langle v'\rangle \\ r \mapsto \langle v'\rangle \\ v \hookleftarrow_q L \setminus \{s\} \\ v' \hookleftarrow_{q'} L' \uplus \{s\} \end{Bmatrix}$$

$$\textsc{Alloc}\quad \begin{Bmatrix} \diamond \mathit{size}(()^n) \\ s \mapsto \langle v\rangle \\ v \hookleftarrow_q L \end{Bmatrix} *s = \mathsf{alloc}\ n \begin{Bmatrix} \exists \ell. & \ell \mapsto ()^n \\ & \ell \hookleftarrow \{s\} \\ & s \mapsto \langle \ell\rangle \\ & v \hookleftarrow_q L \setminus \{s\} \end{Bmatrix}$$

$$\textsc{Load}\quad \frac{\vec{v}(o) = v}{\begin{Bmatrix} r \mapsto \langle \ell\rangle \\ \ell \mapsto_p \vec{v} \\ s \mapsto \langle v'\rangle \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{Bmatrix} *s = [*r + o] \begin{Bmatrix} r \mapsto \langle \ell\rangle \\ \ell \mapsto_p \vec{v} \\ s \mapsto \langle v\rangle \\ v \hookleftarrow_q L \uplus \{s\} \\ v' \hookleftarrow_{q'} L' \setminus \{s\} \end{Bmatrix}}$$

$$\textsc{Store}\quad \frac{\vec{v}(o) = v}{\begin{Bmatrix} s \mapsto \langle \ell\rangle \\ r \mapsto \langle v'\rangle \\ \ell \mapsto_1 \vec{v} \\ v \hookleftarrow_q L \\ v' \hookleftarrow_{q'} L' \end{Bmatrix} [*s + o] = *r \begin{Bmatrix} s \mapsto \langle \ell\rangle \\ r \mapsto \langle v'\rangle \\ \ell \mapsto_1 [o := v']\vec{v} \\ v \hookleftarrow_q L \setminus \{\ell\} \\ v' \hookleftarrow_{q'} L' \uplus \{\ell\} \end{Bmatrix}}$$

$$\textsc{LocEq}\quad \begin{Bmatrix} r_1 \mapsto \langle \ell_1\rangle \\ r_2 \mapsto \langle \ell_2\rangle \\ s \mapsto \langle v\rangle \\ v \hookleftarrow_q L \end{Bmatrix} *s = (*r_1 == *r_2) \begin{Bmatrix} r_1 \mapsto \langle \ell_1\rangle \\ r_2 \mapsto \langle \ell_2\rangle \\ s \mapsto \langle \ell_1 = \ell_2\ ?\ 1 : 0\rangle \\ v \hookleftarrow_q L \setminus \{s\} \end{Bmatrix}$$

$$\textsc{Alloca}\quad \frac{\forall c.\ \{\Phi \star c \mapsto \langle()\rangle\}\ [c/x]i\ \{c \mapsto \langle()\rangle \star \Psi\}}{\{\Phi\}\ \mathsf{alloca}\ x\ \mathsf{in}\ i\ \{\Psi\}}$$

$$\textsc{Fork}\quad \frac{\forall c.\ \begin{Bmatrix} c \mapsto \langle v\rangle \\ \Phi \star v \hookleftarrow_q L \setminus \{r\} \uplus \{c\} \end{Bmatrix}\ [c/x]i\ \{c \mapsto \langle()\rangle\}}{\begin{Bmatrix} r \mapsto \langle v\rangle \\ \Phi \star v \hookleftarrow_q L \end{Bmatrix}\ \mathsf{fork}\ *r\ \mathsf{as}\ x\ \mathsf{in}\ i\ \{r \mapsto \langle()\rangle\}}$$

Fig. 4. Reasoning Rules

Second, the fraction q is *not* required to be 1. That is, a fractional pointed-by assertion suffices for edge deletion. In other words, the permission to (add or) delete pointers to an object can be shared.

Third, a potential pitfall is that if the fraction q is less than 1, then the system does not guarantee that s is a member of L. In practice, it should most often be the case that $s \in L$ holds, because the user is usually careful to split v's pointed-by assertion in such a way that this property holds. Still, it could happen that s is not a member of L. Then, $L \setminus \{s\}$ is equal to L, which means that the deletion of the edge of s to v is *not* recorded at the logical level. This does not compromise the soundness of the program logic; it simply means that the predecessor multiset of the value v becomes over-approximated. The rule Covariance↤ in Figure 3, already discussed in §3.3, is another way of introducing such an over-approximation. Over-approximation is acceptable if it can be later eliminated. The reasoning rule Cleanup offers one way of doing so: once a memory location ℓ has been deallocated, all occurrences of ℓ in predecessor multisets, whether they are "real"

or "spurious", can be removed. The law ConfrontMapsTo$\mapsto\!\leftarrow\!\dashv$ offers another way: it allows re-discovering the identity of a predecessor.

Finally, one can establish simplified variants of the reasoning rules Const, Move, Alloc, Load, Store, and LocEq, where edge deletion is not recorded. In the simplified Store rule (not shown), for instance, the assertions $v \leftarrow\!\dashv_q \ldots$ are removed from the pre- and postcondition: thus, the deletion of the edge that leads from ℓ to v is not recorded. As explained above, it is sometimes acceptable to over-approximate predecessor multisets; then, these more lightweight rules can be used.

Loads and Stores. Read and write access to heap blocks are described by the rules Load and Store in Figure 4. Although these rules are unfortunately quite verbose, all of the ideas required to understand them have been presented already. As usual in Separation Logic, read access requires a fractional points-to assertion $\ell \mapsto_p \vec{v}$, whereas write access requires a full points-to assertion $\ell \mapsto_1 \vec{v}$, which in the postcondition is updated to $\ell \mapsto_1 [o := v']\vec{v}$. The rest is boilerplate. Two points-to assertions describe the stack cells r and s before and after the operation. Two pointed-by assertions describe the predecessors of the values v and v' before and after the operation; one edge deletion and one edge addition take place.

Memory Allocation on the Stack. The reasoning rule Alloca states that a fresh stack cell c exists during the execution of the instruction i. The points-to assertion $c \mapsto \langle () \rangle$ appears upon entry and disappears upon exit. Although we could, we do not make the pointed-by assertion $c \leftarrow\!\dashv \emptyset$ available to the user, so she cannot create pointers to the stack cell c. Because the rule requires the stack cell to contain a unit value upon exit, the programmer may need to explicitly kill this cell by writing a unit value into it. We impose this requirement for the sake of simplicity. We have also proved a more permissive (and more complex) version of the reasoning rule, which does not have this requirement, and performs an edge deletion upon exit.

Fork. The reasoning rule Fork reflects the operational semantics of fork. The content of the stack cell r changes from v to $()$. In the new thread, a fresh stack cell c appears, and is initialized with v. The treatment of this stack cell in the premise of Fork is analogous to what can be seen in the premise of Alloca. Finally, the assertions Φ and $v \leftarrow\!\dashv_q L$ are transmitted from the original thread to the new thread. The choice of a suitable assertion Φ is made by the user. The pointed-by assertion is updated to $v \leftarrow\!\dashv_q L \setminus \{r\} \uplus \{c\}$ in the new thread, reflecting the fact that the pointer from r to v is replaced with a pointer from c to v.

4 SOUNDNESS OF SL⋄

SL⋄ is an instance of Iris [Jung et al. 2018]. This means that SL⋄ is set up via the following steps:

(1) define a certain amount of *ghost state* that is needed in the next two steps;
(2) define a central *invariant* that ties the machine's physical state to the ghost state;
(3) define the *assertions* that an end user of SL⋄ exploits, namely points-to assertions, deallocation witnesses, space credits, and pointed-by assertions; all of these assertions are defined as the ownership of a fragment of the ghost state.

Once these steps have been carried out, Iris provides a semantic definition of the Hoare triple $\{\Phi\}\ i\ \{\Psi\}$. We can then use Hoare triples to state the reasoning rules of SL⋄ (Figure 4), to prove that these rules are valid, and to state and prove the fact that SL⋄ is sound. This is the topic of the next subsection (§4.1). After presenting these results, in §4.2, we come back to the three steps listed above and describe them at an informal level, without diving into the concepts of Iris, which would be difficult to explain in a thorough and concise way.

4.1 The Soundness Statement

The operational semantics of SpaceLang guarantees that the size of the store never exceeds S (§2.1). A program that attempts to allocate memory beyond this limit becomes stuck (§2.3). Thus, in order to prove that SL$\diamond$ is sound, all we have to do is prove that it rules out the possibility of a runtime failure. This is guaranteed by the following theorem:

Theorem 4.1 (Soundness of the Logic SL$\diamond$). *Suppose $\{\diamond S\}\ i\ \{True\}$ holds. Then, the execution of the instruction i, beginning in an empty store, cannot result in a configuration where a thread is stuck.*

This theorem states that if the program i satisfies a semantic Hoare triple whose precondition is $\diamond S$, then this program cannot crash. In particular, it cannot reach a situation where a memory allocation request cannot be satisfied because the heap is full. The intuitive reason why this holds is that $\diamond S$ represents a permission to allocate at most S words of memory, and that is precisely the amount of memory that is initially available. The proof of this theorem is short: it is a direct consequence of Iris's adequacy theorem [Jung et al. 2018, §6.4].

Separately, we prove that the reasoning rules of SL$\diamond$ are valid, that is, each rule is a valid lemma:

Theorem 4.2 (Validity of the Reasoning Rules). *Each of the rules in Figures 3 and 4 is valid.*

The proof of this theorem represents more work: one lemma per reasoning rule is required, not to mention a large number of auxiliary lemmas. The proofs of these lemmas are fairly uninteresting; the key insights lie in the definitions presented in the next subsection (§4.2).

Together, Theorems 4.1 and 4.2 guarantee that, if the Hoare triple $\{\diamond S\}\ i\ \{True\}$ can be obtained by applying the reasoning rules of SL$\diamond$, then the program i is safe, that is, it cannot crash and the size of its live heap data cannot exceed S.

4.2 Key Definitions and Invariants

For the definitions that follow, we need some more terminology about stores.

Definition 4.3 (More Terminology). We write $successors(\sigma, \ell)$ for the multiset of the locations ℓ' such that there is an edge in the store σ from ℓ to ℓ'. A store σ is *closed* if the existence of an edge from ℓ to ℓ' implies $\ell' \in dom(\sigma)$. The *available space* in a store σ is $available(\sigma) = S - size(\sigma)$. The set $freed(\sigma)$ is the set $\{\ell \mid \sigma(\ell) = \lightning\}$ of the locations that have been allocated, then freed.

To give meaning to pointed-by assertions, we introduce the concept of a *predecessor map* π, and define an invariant that relates a store σ and a predecessor map π. Our intention is to exploit Iris's invariants and ghost state to express the idea that, at every point in time, there exists a global predecessor map, which is consistent with the current logical store, and that every pointed-by assertion corresponds to a fragment of this predecessor map.

Definition 4.4 (Predecessor Map). A *predecessor map* π is a finite map of memory locations to finite multisets of memory locations. A store σ and a predecessor map π are *consistent* with one another, which we write $\sigma \bowtie \pi$, if the following three properties hold:

(1) $dom(\sigma) \setminus freed(\sigma) = dom(\pi)$;
(2) for all locations ℓ, ℓ', the inequality $\ell' \,\$\, successors(\sigma, \ell) \leq \ell \,\$\, predecessors(\pi, \ell')$ holds;
(3) for all locations ℓ, ℓ', $\ell \in predecessors(\pi, \ell')$ implies $\ell \in dom(\sigma)$.

Property (1) states that the domain of π is the domain of σ, deprived of the locations that have been freed. This reflects the fact that we do not need (and do not allow) a pointed-by assertion about a deallocated location.

Property (2) reflects the fact that, for every forward edge in the store σ, there is a backward edge in the predecessor map π. The property is stated so as to tolerate multiple edges from ℓ to ℓ',

which exist when the memory block at location ℓ contains several pointers to ℓ'. Property (2) is an inequality; the reverse inequality is not required. This reflects the fact that we allow predecessor multisets to be over-approximated. The validity of the law CONFRONT$\mapsto\leftarrow$ in Figure 3 is a direct consequence of this property.

Property (3) is a simple sanity requirement: the locations in the multiset $\mathit{predecessors}(\pi, \ell')$ must have been allocated in the past. (They may or may not have been deallocated since.) In other words, a predecessor multiset cannot contain a location that has never been allocated at all.

Next, we must define the relation that ties the physical store to the logical store, two concepts that we have informally described earlier (§3.4). From here on, we write σ for the physical store and θ for the logical store.

Definition 4.5 (Relation between Physical and Logical Stores). Two stores σ and θ are *related*, which we write $\sigma \approx \theta$, when they have the same domain, the same reachable locations, and the same content at every reachable location.

Equipped with the above definitions, we are now prepared to enter the realm of Iris and perform the three steps described at the beginning of §4, that is, define the ghost state, the central invariant, and the assertions of SL$\diamond$. We give the definitions first and explain them afterwards.

Assumption 4.6. Iris defines a certain piece of ghost state, defines a predicate *Heap* σ that ties a store σ to this ghost state, and defines the points-to assertion $\ell \mapsto_q b$ in terms of this ghost state. This is visible in the paper [Jung et al. 2018, §6.3.2] and in Iris's `gen_heap` library [Iris 2021]. We re-use this machinery without change, so we do not repeat these definitions. We mention the predicate *Heap* in our own invariant (Definition 4.8), where it is applied to the logical store θ.

Definition 4.7 (Ghost State). Two *ghost cells* γ and δ are allocated at the beginning of a program's execution. The cell γ stores an element of the monoid $\textsc{Auth}(\textsc{Nat}, +)$, while the cell δ stores an element of the monoid $\textsc{Auth}(\mathcal{L} \rightarrow_{\text{fin}} (\textsc{Frac}, +) \times (\textsc{Multiset}(\mathcal{L}), \uplus))$ [Jung et al. 2018, §6.3.3].

Definition 4.8 (Central Invariant). The central invariant of SL$\diamond$, or *state interpretation invariant* [Jung et al. 2018, §7.3], is the following Iris assertion:

$$I\,\sigma \triangleq \exists\theta. \begin{cases} \sigma \text{ is closed} & \star & \mathit{size}(\sigma) \le S & \star \\ \theta \text{ is closed} & \star & \mathit{size}(\theta) \le S & \star \\ \sigma \approx \theta & \star & \mathit{Heap}\ \theta & \star \\ \boxed{\bullet\, \mathit{available}(\theta)}^{\gamma} & \star & \exists\pi.\ \theta \bowtie \pi & \star\ \boxed{\bullet\,(1.\pi)}^{\delta} \end{cases}$$

Definition 4.9 (Assertions). The assertions of SL$\diamond$ are defined as follows:

(1) The *points-to assertion* $\ell \mapsto_q b$ is inherited from Iris (Assumption 4.6).
(2) The *deallocation witness* for a single location, $\dagger\dagger\{\ell\}$, is defined as $\ell \mapsto_\square \lightning$, where $\square$ is a *discarded fraction* [Vindum and Birkedal 2021]. The deallocation witness for a set of locations, $\dagger\dagger D$, is defined as the iterated conjunction $*_{\ell \in D} \dagger\dagger\{\ell\}$.
(3) The *space credits* $\diamond n$ are defined as $\boxed{\circ\, n}^{\gamma}$.
(4) The *pointed-by assertion* $\ell \leftarrow\!\!\mapsto_q L$ is defined as $\exists L'.\ \boxed{\circ\,[\ell := (q, L')]}^{\delta} \star L' \subseteq L$.

Let us try and explain the most important aspects of these definitions.

In Iris, the state interpretation invariant $I\,\sigma$ is an assertion that holds of the *physical store* σ in between every two steps of computation. It is used by Iris in the definition of Hoare triples; we inherit this definition from Iris.

Our definition of $I\,\sigma$ begins with an existential quantification over a *logical store* θ. The invariant records the fact that both stores are closed (Definition 4.3) and valid (§2.1). It is a simple property

of SpaceLang's semantics that the physical store always remains closed and valid; we require the same property of the logical store. The two stores are related by the relation $\sigma \approx \theta$ (Definition 4.5), which means that their reachable fragments coincide.

As noted earlier (Assumption 4.6), the predicate *Heap* and the points-to assertion are inherited from Iris. What this means, as far as we are concerned, is that the points-to assertion is defined in a standard manner. One unusual aspect is that *Heap* is applied to the logical store θ, not to the physical store σ, as is usually the case [Jung et al. 2018, §6.3.2]. Thus, the points-to assertion $\ell \mapsto_q b$ guarantees the existence of the memory block b at address ℓ *in the logical store*. If this address is reachable, then the relation $\sigma \approx \theta$ guarantees that the block b also exists at address ℓ in the physical store. If the address ℓ has become unreachable, however, then it may be the case that ℓ has been deallocated already in the physical store.

The deallocation witness $\dagger\dagger\{\ell\}$ is in fact a special case of the points-to assertion: it is defined as $\ell \mapsto_\square \lightning$, where $\square$ is a *discarded fraction* [Vindum and Birkedal 2021]. This persistent assertion guarantees that the address ℓ has been deallocated and will remain so forever.

While the central invariant involves the *authoritative assertion* $\boxed{\bullet\, \mathit{available}(\theta)}^{\gamma}$, the assertion $\diamond n$ expands to the *fragmentary assertion* $\boxed{\circ\, n}^{\gamma}$ [Jung et al. 2018, §6.3.3]. Furthermore, the ghost cell γ ranges over the monoid $\textsc{Auth}(\textsc{Nat}, +)$. This means intuitively that the total number of space credits in circulation is at most $\mathit{available}(\theta)$ and that to own n space credits is to own n units out of this number. In particular, according to Iris's laws for ghost state, from the conjunction $\boxed{\bullet\, \mathit{available}(\theta)}^{\gamma} \star \boxed{\circ\, n}^{\gamma}$, one can deduce $\exists n'.\ \mathit{available}(\theta) = n + n'$, that is, $\mathit{available}(\theta) \geq n$. Thus, the ownership of n space credits guarantees that there exist n words of available space in the logical store. From this fact and from the relation $\sigma \approx \theta$, one cannot deduce that there exist n words of available space in the physical store, because the physical store may be partly filled with garbage. However, one *can* deduce that, *by letting the garbage collector run*, it is possible to reach a situation where n words of memory are available in the physical store.

The central invariant states that there exists a predecessor map π and that this map is related to the logical store by the relation $\theta \bowtie \pi$ (Definition 4.4). Again, the invariant involves the authoritative assertion $\boxed{\bullet\, (1.\pi)}^{\delta}$, while the definition of the pointed-by assertion $\ell \hookleftarrow_q L$ involves a fragmentary assertion $\boxed{\circ\, [\ell := (q, L')]}^{\delta}$.[6] This means that a pointed-by assertion represents a fragment of the central predecessor map π. The ghost cell δ ranges over a monoid that is chosen so as to validate the law $\textsc{Join}{\hookleftarrow}$ in Figure 3. The existential quantification over L' in Definition 4.9, together with the inequality $L' \subseteq L$, allows predecessor multisets to be over-approximated; this validates the law $\textsc{Covariance}{\hookleftarrow}$ in Figure 3.

In summary, one idea that is worth taking away is that, in SL$\diamond$, all assertions (points-to assertions; deallocation witnesses; space credits; pointed-by assertions) are really about the logical store. However, because the physical store and the logical store are tightly related, these assertions can also be used to reason about the physical store.

The complete definition and proof of soundness of SL$\diamond$ can be found in our repository [Madiot and Pottier 2021].

5 REFERENCE-COUNTING STYLE

On many occasions, our pointed-by assertions are too precise. The assertion $v \hookleftarrow_q L$ keeps track of a multiset of predecessors L, where the location and the multiplicity of every predecessor are explicitly recorded. This information can be needlessly precise; what is worse, it can be inconvenient

[6] We write $1.\pi$ for the finite map $\mathit{map}\ (\lambda L.\ (1, L))\ \pi$. That is, whereas π maps every location to a multiset of predecessors, $1.\pi$ maps every location to a pair of the fraction 1 and a multiset of predecessors. We write $[\ell := (q, L')]$ for the singleton map that maps ℓ to the pair (q, L').

or impossible to maintain. A typical example is that of a container data structure, such as a stack or a priority queue. When an element is inserted into the container, a pointer from the container to the element is created. So, the pointed-by assertion for this element must be updated: the existence of a new predecessor must be recorded. Unfortunately, this predecessor is usually an object that is part of the internal representation of the container data structure: for instance, it could be an internal node in a binary tree. Thus, it is an object whose address cannot be named by a user of the container abstraction. As a result, it can be unclear at first how to write natural and useful specifications for the operations that insert and extract elements into and out of the container.

To solve this problem, the predecessor information must be made less precise, while remaining useful. We envision several ways of making this information more abstract, while retaining a sufficient level of precision. One approach would be to keep track of predecessors in terms of abstract entities, instead of concrete locations. At an intuitive level, the specification of "insert" would state that "the container" becomes a predecessor of the newly inserted element, while the specification of "extract" or "remove" would state that "the container" is no longer a predecessor of the element that has been extracted. Another approach is to count predecessors, without keeping track of their identity. At an intuitive level, the specification of "insert" would state that the newly inserted element gains one predecessor, while the specification of "extract" would state that the element that has been extracted loses one predecessor. In either approach, it is desirable that the sequential composition of insertion and extraction cause no loss of precision: if an element is inserted into the container and immediately extracted, then the static information that is available about its predecessors should be unaffected.

We believe that this abstraction of the predecessor information can be programmed by the user in a library, providing an additional layer of abstraction on top of SL$\diamond$. As an example, in this section, we present a simple way of keeping track of predecessors in *reference-counting style*. We leave the exploration of other styles to future work.

A user who wishes to keep track of the number of predecessors, while ignoring their identity, is naturally led to define a new pointed-by assertion $v \hookleftarrow n$ where n is a natural number. The simplest way of defining such an assertion is as follows:

$$v \hookleftarrow n \triangleq \exists L.\ (v \hookleftarrow_1 L \ \star\ |L| = n)$$

The existential quantification on L indicates that it is not known where there are pointers to v, but it is known that there exist n such pointers: indeed, n must be the cardinality of the multiset L. We require the fraction 1; the next paragraph explains why.

For each reasoning rule in Figure 4, it is possible to derive a variant of the rule in reference-counting style. The addition of a predecessor, which was expressed by transforming $v \hookleftarrow L$ into $v \hookleftarrow L \uplus \{\ell\}$, is now expressed by transforming $v \hookleftarrow n$ into $v \hookleftarrow n + 1$. Clearly, this is sound because the function "cardinality" maps multiset sum to addition. The deletion of a predecessor, which was expressed by transforming $v \hookleftarrow L$ into $v \hookleftarrow L \setminus \{\ell\}$, is now expressed by transforming $v \hookleftarrow n$ into $v \hookleftarrow n - 1$. The reason why this is sound is more subtle. The law $|L \setminus \{\ell\}| = |L| - 1$ is not true in general: it is true only under the side condition $\ell \in L$. Fortunately, when a predecessor is deleted, a points-to assertion is at hand, stating the existence of a pointer of ℓ to v; the law Confront$\mapsto\hookleftarrow$ can then be used to obtain $\ell \in L$.[7]

As an example, from Move, one can derive RC-Move (Figure 5). By following a similar pattern, one gives a version of almost every reasoning rule in reference-counting style. We omit these rules; the reader can find them in our repository [Madiot and Pottier 2021]. One exception is Cleanup, which

[7] The use of Confront$\mapsto\hookleftarrow$ is the reason why the deletion of a predecessor requires the fraction 1. It may be possible to remove this limitation by adopting a more complex definition of $v \hookleftarrow_q n$. We leave it to future work to determine whether this is possible and desirable.

RC-Move
$$\left\{\begin{array}{c} s \mapsto \langle v \rangle \\ r \mapsto \langle v' \rangle \\ v \hookleftarrow n \\ v' \hookleftarrow n' \end{array}\right\} \; *s = *r \; \left\{\begin{array}{c} s \mapsto \langle v' \rangle \\ r \mapsto \langle v' \rangle \\ v \hookleftarrow n-1 \\ v' \hookleftarrow n'+1 \end{array}\right\}$$

RC-Free-Singleton
$$\left\{\begin{array}{c} \ell \mapsto \vec{v} \;\star\; \ell \hookleftarrow 0 \\ \circledast_{(v,n)\in vns} \; v \hookleftarrow n \end{array}\right\} \Rrightarrow_I \left\{\begin{array}{c} \diamond size(\vec{v}) \\ \circledast_{(v,n)\in vns} \; v \hookleftarrow n - (v \,\$\, \vec{v}) \end{array}\right\}$$

Fig. 5. Selected Reasoning Rules in Reference-Counting Style

$$\begin{array}{rcl} isList\ \ell\ [] & \triangleq & \ell \mapsto [0] \\ isList\ \ell\ (v :: vs) & \triangleq & \exists \ell'.\ \ell \mapsto [1; v; \ell'] \star \ell' \hookleftarrow 1 \star isList\ \ell'\ vs \end{array}$$

Fig. 6. The Predicate *isList*, in Reference-Counting Style

has no sensible counterpart in this style. To address this problem, we combine Confront$\mapsto\hookleftarrow$, Free, and Cleanup, and obtain a reasoning rule that does admit a useful counterpart in reference-counting style, namely RC-Free-Singleton (Figure 5). This rule states that, if there is a tuple $\vec{v}$ at address ℓ in the heap, and if there are no pointers to this memory block (as witnessed by the assertion $\ell \hookleftarrow 0$), then this block can be logically freed. The reference count of every successor can then be decremented. This is expressed by the last part of the precondition and postcondition: for an arbitrary value v and natural number n, the assertion $v \hookleftarrow n$ can be updated to $v \hookleftarrow n - (v \,\$\, \vec{v})$, where $v \,\$\, \vec{v}$ denotes the multiplicity of the value v in the tuple $\vec{v}$.[8] The user chooses the list vns, a list of pairs (v, n), and is expected to choose a list that covers every value v in the tuple $\vec{v}$. In the common case where $\vec{v}$ is a tuple of distinct values, the user is expected to provide a pointed-by assertion $v \hookleftarrow n$ for each of these values, and this assertion is updated to $v \hookleftarrow n - 1$.

The above rules express the familiar discipline of reference counting [Collins 1960]. In the present case, they implement a form of *ghost reference counting*. Indeed, no counters exist at runtime. The counting discipline is purely static; its rules are applied as part of the verification of a program.

As suggested at the beginning of this section, we believe that reference counting is just one of several possible styles, or approaches to working with abstract predecessor information in SL$\diamond$. In future work, we wish to investigate other styles, including one where predecessors are not identified by their address, but are instead grouped in regions. We also plan to investigate whether and how several styles can be jointly used in the verification of a single program.

6 EXAMPLES

We have verified in SL$\diamond$ a few basic functions, including a function that copies a linked list (§6.1), and an implementation of a stack as a mutable reference to a linked list (§6.2). We now present the specifications of the list copy operation and of the stack data structure.

6.1 Linked Lists and List Copy

We represent linked lists in memory as follows: the empty list is represented as a tuple of one field, containing the tag 0; a nonempty list is represented as a tuple of three fields, containing the tag 1, the list head, and a pointer to the tail of the list. This convention is reflected in the definition of the predicate *isList* (Figure 6). It is the traditional definition [Reynolds 2002], with the addition of the

[8] In reality, we adopt a slightly more complex definition of $v \,\$\, \vec{v}$, which does not rely on decidable equality of values. For memory locations, we define $\ell \,\$\, \vec{v}$ as the multiplicity of the value ℓ in the tuple $\vec{v}$. For a value v other than a memory location, we define $v \,\$\, \vec{v}$ as $|\vec{v}|$, the length of the tuple $\vec{v}$. This is arbitrary; in this case, $v \hookleftarrow n$ is equivalent to *True* anyway.

```
copy ≜ λ(self, dst, src).
  alloca tag in *tag = [*src + 0];          – read the list's tag
  if *tag then                              – if this is a cons cell, then
    alloca head in *head = [*src + 1];      – read the list's head
    alloca tail in *tail = [*src + 2];      – read the list's tail
    *src = ();                              – clobber this root
    alloca dst' in *self(self, dst', tail); – copy the list's tail
    *dst = alloc 3;                         – allocate a new cons cell
    [*dst + 0] = *tag;                      – and initialize it
    [*dst + 1] = *head;
    [*dst + 2] = *dst'
  else                                      – this must be a nil cell
    *src = ();                              – clobber this root
    *dst = alloc 1;                         – allocate a new nil cell
    [*dst + 0] = *tag                       – and initialize it
```

Fig. 7. A List Copy Procedure in SpaceLang

$$\forall v \in vs.\ \exists n.\ (v, n) \in vns$$

$$\left\{\begin{array}{c} f \mapsto \langle copy \rangle \\ dst \mapsto \langle () \rangle \\ src \mapsto \langle \ell \rangle \\ isList\ \ell\ vs \star \ell \hookleftarrow m \\ m = 1\ ?\ True : \diamond(2 + 4 \times |vs|) \\ \circledast_{(v,n) \in vns}\ v \hookleftarrow n \end{array}\right\}\ *f(f, dst, src)\ \left\{\begin{array}{c} f \mapsto \langle copy \rangle \\ dst \mapsto \langle \ell' \rangle \\ src \mapsto \langle () \rangle \\ \exists \ell'.\ m = 1\ ?\ True : (isList\ \ell\ vs \star \ell \hookleftarrow m - 1) \\ isList\ \ell'\ vs \star \ell' \hookleftarrow 1 \\ \circledast_{(v,n) \in vns}\ v \hookleftarrow n + (m = 1\ ?\ 0 : v\ \$\ vs) \end{array}\right\}$$

Fig. 8. A Specification of List Copy, in Reference-Counting Style

assertion $\ell' \hookleftarrow 1$ in the second line, which indicates that the pointer from the first cell at address ℓ to the list tail at address ℓ' is the sole pointer to ℓ': that is, the list tail is not shared. Assuming that one wishes to express the absence of sharing (as we do, here), this assertion is necessary: indeed, the traditional chain of points-to assertions alone expresses the unique ownership of every cell but does not rule out the existence of pointers from the outside into the list.

The code of a list copy procedure, *copy*, appears in Figure 7. It is a function of three parameters. The stack cell *self* is expected to hold a pointer to *copy* itself: this pointer allows the recursive call. The stack cell *dst*, an "out parameter", is used to return the address of the new list. The stack cell *src* holds the address of the original list. One could define a version of *copy* where a single stack cell plays the role of both *dst* and *src*, an "in-out parameter". For simplicity, we keep these cells separate. Due to the low-level character of SpaceLang, the code is somewhat verbose; it is however fairly straightforward. Up to three new stack cells, *tag*, *head*, and *tail*, are allocated in order to load the three fields of the first list cell. Another stack cell, *dst'*, is allocated in order to receive the result of the recursive call.

A specification of *copy* appears in Figure 8. This is a rich specification, which provides precise heap-space-usage information and keeps track of reference counts both for the list and for its elements. This explains why the specification may seem intimidating at first. Let us review its key

aspects. The first three lines in the pre- and postcondition describe the initial and final contents of the stack cells f, *dst*, and *src*. The next line describes the original list at address ℓ. The precondition requires the unique ownership of this linked list and assumes that its reference count is m; this is expressed by the assertion $\mathit{isList}\ \ell\ \mathit{vs} \star \ell \hookleftarrow m$. In the postcondition, the reference count is decremented, because the pointer from *src* to ℓ is destroyed; thus, $\mathit{isList}\ \ell\ \mathit{vs} \star \ell \hookleftarrow m-1$ is produced. In the special case where m is 1, the list at address ℓ becomes inaccessible and is logically deallocated by *copy*: in that case, nothing is produced instead of $\mathit{isList}\ \ell\ \mathit{vs} \star \ell \hookleftarrow 0$. This is a design choice. We could have instead chosen to not exploit RC-Free-Singleton in the verification of *copy* and to produce $\mathit{isList}\ \ell\ \mathit{vs} \star \ell \hookleftarrow m-1$ in all cases. One positive effect of this choice is the tight space bound that we are able to express, which is visible in the next line of the precondition. When m is greater than 1, we require $2+4\times|\mathit{vs}|$ space credits, that is, enough free space to allocate a new list. When m is 1, however, we require no space credits, that is, we guarantee that *copy* runs in constant heap space! Indeed, in this case, the first list cell becomes inaccessible when the pointer stored in the stack cell *src* is destroyed by the instruction $*\mathit{src} = ()$. Thus, we can apply RC-Free-Singleton and obtain enough space credits to allocate a new list cell. This is reminiscent of Hofmann's use of space credits in LFPL [2000], except in the present case, we have garbage collection instead of an explicit deallocation instruction.

The assertion $\mathit{isList}\ \ell'\ \mathit{vs} \star \ell' \hookleftarrow 1$ in the postcondition guarantees that the newly-allocated list is uniquely owned and is not shared.

There remains to explain the last line of the pre- and postcondition. Its purpose is to update the reference counts of the list elements. This is done in the same way as in RC-Free-Singleton. In the general case where m is greater than 1, the reference count of each element v is incremented by the number of times this element occurs in the list, that is, by $v\ \$\ \mathit{vs}$. This reflects the fact that this element is now accessible both via the original list and via the copy. However, in the special case where m is 1, the reference count of the element does not change, because the original list is logically deallocated and no longer contributes to the reference count of the elements. The side condition $\forall v \in \mathit{vs}.\ \exists n.\ (v,n) \in \mathit{vns}$ ensures that the reference count of every list element is properly updated.

6.2 A Stack

Figure 9 shows the specification of an abstract data structure that behaves like a stack. We omit its code; it is implemented as a mutable reference to a linked list of elements. This stack offers three operations, namely *create*, *push* and *pop*, plus the ghost operation of logically deallocating a (possibly nonempty) stack. We believe that the specification is fairly easy to read. *create* consumes 4 space credits and produces a new stack. The abstract predicate $\mathit{isStack}\ \ell\ []$ indicates that the stack is empty; the pointed-by assertion $\ell \hookleftarrow 1$ indicates that it is not shared. *push* consumes 4 space credits and pushes an element v onto the stack; the reference count of this element is incremented by one. *pop* pops an element off the stack and releases 4 space credits. The reference count of this element is unchanged because a new pointer from the stack cell *elem* to this element is created. The ghost operation of deallocating a stack is applicable as soon as this stack has reference count 0. The assertions $\mathit{isStack}\ \ell\ \mathit{vs}$ and $\ell \hookleftarrow 0$ are consumed. The space occupied by the stack, that is $4+4\times|\mathit{vs}|$ words, is released, and the reference count of every stack element is updated. This shows that a ghost operation, which has no runtime cost, can produce a linear amount of space credits. It is not necessary to write an explicit loop (or recursive function) to obtain this effect.

7 RELATED WORK

There is a rich literature on measuring or controlling the heap space consumption of programs, via type systems, static and symbolic analyses, program logics, and so on. We review some of the most

$$\left\{\begin{array}{c} f \mapsto \langle create \rangle \\ stack \mapsto \langle () \rangle \\ \diamond 4 \end{array}\right\} \quad *f(stack) \quad \left\{\begin{array}{c} f \mapsto \langle create \rangle \\ \exists \ell. \quad stack \mapsto \langle \ell \rangle \\ isStack\ \ell\ [] \star \ell \hookleftarrow 1 \end{array}\right\}$$

$$\left\{\begin{array}{c} f \mapsto \langle push \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ \diamond 4 \star isStack\ \ell\ vs \\ v \hookleftarrow n \end{array}\right\} \quad *f(stack, elem) \quad \left\{\begin{array}{c} f \mapsto \langle push \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ isStack\ \ell\ (v :: vs) \\ v \hookleftarrow n+1 \end{array}\right\}$$

$$\left\{\begin{array}{c} f \mapsto \langle pop \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle () \rangle \\ isStack\ \ell\ (v :: vs) \\ v \hookleftarrow n \end{array}\right\} \quad *f(stack, elem) \quad \left\{\begin{array}{c} f \mapsto \langle pop \rangle \\ stack \mapsto \langle \ell \rangle \\ elem \mapsto \langle v \rangle \\ \diamond 4 \star isStack\ \ell\ vs \\ v \hookleftarrow n \end{array}\right\}$$

$$\left\{\begin{array}{c} isStack\ \ell\ vs\ \star\ \ell \hookleftarrow 0 \\ \ast \quad v \hookleftarrow n \\ {\scriptstyle (v,n) \in vns} \end{array}\right\} \quad \Rrightarrow_I \quad \left\{\begin{array}{c} \diamond(4 + 4 \times |vs|) \\ \ast \quad v \hookleftarrow n - (v\ \$\ vs) \\ {\scriptstyle (v,n) \in vns} \end{array}\right\}$$

Fig. 9. A Specification of Stacks, in Reference-Counting Style

closely related papers, and refer the reader to Niu and Hoffmann [2018] for more. On a spectrum that ranges from most automated (and least expressive) to most expressive (and least automated) methods, our work lies near the latter end. We believe that it is the first program logic that allows reasoning about heap space in the presence of garbage collection.

Operational Semantics for Garbage Collection. Reasoning about space usage in the presence of garbage collection requires defining a suitable *cost model*, that is, typically, an operational semantics where the action of the garbage collector is explicitly modeled. Felleisen and Hieb [1992], followed by Morrisett et al. [1995], present and study several such semantics expressed in a small-step style. The roots are the memory locations that appear in the current expression: this is known as the *free-variable rule*.[9] Garbage collection can also be modeled in big-step style [Blelloch and Greiner 1996; Minamide 1999; Niu and Hoffmann 2018; Spoonhower et al. 2010]. Then, the evaluation judgement is typically parameterized with a set of roots. When descending under an evaluation context, this set is augmented with the memory locations that appear in the context. The effect of such a setup is analogous to that of the free-variable rule.

We follow a different route and consider a low-level calculus where a variable denotes the address of a stack cell. This contrasts with ordinary call-by-value λ-calculus, where a variable denotes a value. As a result, we have no need for a free-variable rule, that is, no need to extract the memory locations that appear in the current expression: it suffices instead to view every stack cell as a root. This allows us to model garbage collection as a relation between stores, something that was not possible in the calculi cited above, and to use the Iris framework without modification.[10]

[9] In these papers, memory locations and variables are conflated.

[10] Iris lets the user choose the definition of *thread-local* reduction, but does not allow the user to customize *thread-pool* reduction, which is rigidly defined as the interleaving of thread-local reduction steps. Thus, unless one is willing to modify Iris, one must build garbage collection into thread-local reduction. However, thread-local reduction sees only the current thread, not every thread, so it cannot scan every thread for its roots; this precludes the use of the free-variable rule.

As noted by Morrisett et al. [1995], the free-variable rule offers a conservative answer to an undecidable question, namely: which objects can be safely reclaimed? This rule regards every memory location that appears in the current expression as a root. In a substitution-based semantics, where values are substituted for variables at runtime, this amounts to *regarding a variable as a root as long as there is at least one occurrence of this variable ahead of the current program point.* This implies, in particular, that a variable can cease to be a root before it goes out of scope: it is a fairly precise approximation. In SpaceLang, in contrast, a stack cell continues to be viewed as a root until it is deallocated. This is a less precise approximation; fortunately, by explicitly writing a unit value into a stack cell, the programmer can *kill* it, that is, indicate that this cell should no longer be viewed as a root.

Type-based Analyses. Hofmann [1999, 2003] introduces space credits in the setting of an affine type system for the λ-calculus. Constructing a value consumes a credit $\diamond$; deconstructing a value produces a credit. This is exploited to obtain a result about time complexity: the first-order functions that can be defined in this system are exactly the polynomial-time-computable functions. In a contemporary paper [2000], Hofmann introduces space credits into a functional programming language, LFPL, whose programs can be compiled to C code that uses neither malloc nor free and therefore runs in constant space. There, a value of type $\diamond$ exists at runtime and can be understood as a pointer to a free block in the heap. Aspinall and Hofmann [2002] relax LFPL's affine type discipline so as to make it more expressive. Aspinall and Compagnoni [2003] present a typed assembly language that can serve as a compilation target for LFPL.

Hofmann and Jost [2003] propose an affine type system where types carry space credits, that is, where types are annotated with information about available heap space. For instance, the fact that a list xs has type $\mathsf{L}(\mathsf{B}, 4), 5$ guarantees that xs is a list of Boolean values and that there exist $4|xs| + 5$ words of free space in the heap. The type system is affine, but allows sharing a pointer, provided the associated space credits are split. There is no garbage collection: memory deallocation is explicit, and takes the form of a *destructive pattern matching* construct, which can be applied only to an object that is *not* shared. It is up to the programmer to use deallocation in a safe way: the type system does not enforce this property, which is viewed as an orthogonal problem. In contrast, in the present paper, the reasoning rules guarantee that deallocation (which, in SL$\diamond$, is a ghost operation) is used in a safe way, so that a verified program cannot run out of space.

Hofmann and Jost's paper marks the beginning of a long line of work known as *automatic amortized resource analysis* (AARA). In particular, Hofmann and co-authors develop RAJA [Hofmann and Jost 2006; Hofmann and Rodriguez 2009, 2013], an analysis of a variant of Java where garbage collection has been replaced with explicit deallocation, and RaML [Hoffmann et al. 2012a,b, 2017], an analysis of a variant of OCaml where garbage collection has been replaced with explicit destructive pattern matching. There are other papers of the AARA school where deallocation either must be explicit [Jost et al. 2010, 2009] or is not supported at all [Jost et al. 2017; Simões et al. 2012]. One paper in this line of work stands out by attempting to handle garbage collection. Niu and Hoffmann [2018] propose a big-step operational semantics that models garbage collection. Then, they present a variant of AARA for a pure, first-order programming language, where destructive pattern matching *can* be applied to shared objects. In other words, the cost analysis considers that if there exist two pointers to a shared object, then each of them can be deallocated, and the space occupied by this object is freed twice! This may seem surprising. The reason why it is sound is that the analysis treats the operation of *sharing* a data structure exactly as if it were a *copy*, and considers that one must pay for the size of the copy. As far as we understand, although Niu and Hoffmann's analysis is sound, the high-water mark that it infers can be widely over-approximated, for two main reasons: (1) it is really an analysis of a more costly semantics where data structures

are copied instead of shared; (2) deallocation is tied to pattern matching; when a pointer to a data structure is abandoned, the space occupied by this data structure is not reclaimed in the analysis. Kahn and Hoffmann [2021] present a system that is equipped with more flexible typing rules than its predecessors and therefore can derive tighter resource consumption bounds.

Hughes and Pareto [1999] present Embedded ML (MML), a pure, strict, first-order programming language without garbage collection. MML is the combination of a static region discipline [Tofte and Talpin 1994], which determines where deallocation can take place, and a type system that keeps track of the size of data structures and regions. Region management is explicit: the letreg construct introduces a new region, whose lifetime coincides with its lexical scope. The size of a region is explicitly specified when this region is allocated, and remains fixed. The programmer can allocate memory blocks in this region as long as there is space in it. The type system ensures that this is the case. To achieve this, every typing judgement and every function type carries a *put effect* p, a finite map of regions to natural integers, which indicates how much space this function allocates out of each region. It also carries a *store effect* ϕ that keeps track of the maximum size that the store can reach. The size of the store, at each point in time, is the sum of the sizes of the regions that exist at this point. MML and SL$\diamond$ have incomparable power: while a program logic is in many ways more expressive than a type system, SL$\diamond$ is designed to reason about garbage collection, which cannot deallocate a reachable object, whereas MML's region discipline can give rise to dangling pointers, which the type discipline guarantees are never dereferenced.

Chin et al. [2005] present a type system that 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. The specification of a method includes a precondition and a postcondition, which express constraints on the sizes of the method's arguments and result. It also indicates how much memory the method may need (a high-water mark) and how much memory it releases. As an example, the specification of a stack is provided. As expected, the specification of *push* indicates that this method allocates one list cell and increments the size of the stack; symmetrically, *pop* frees up one list cell and decrements the size of the stack. One limitation of the system is that the stack's elements are considered shared, so they can never be regarded as deallocated by the analysis. In contrast, our specification of a stack (Figure 9) maintains precise reference counts for elements, so, after extracting an element out of a stack, it is possible to recognize that this element is not shared and to logically deallocate it. In later work, Chin et al. [2008] present an analysis that is entirely automated and requires no user annotations.

Static and Symbolic Analyses. Nguyen et al. [2007] propose an automated verification system based on Separation Logic. They allow user-defined inductive predicates, which can be indexed with sizes. There is no reasoning about deallocation. He et al. [2009] re-use an existing Separation Logic-based program verifier, Hip/Sleek, to reason about stack and heap space. They consider a C-like imperative language with explicit deallocation instructions. To reason about space, they instrument the program with two global variables `stk` and `heap` of type *int*, which represent the available space in the stack and in the heap. The special variable `heap` is decremented when a block is allocated and incremented when a block is allocated.

In the setting of Java bytecode, Albert et al. [2007, 2009, 2010, 2013] infer recurrence equations that describe the heap space consumption of a method, expressed as a function of the sizes of its arguments. The system relies on an external analysis that infers object lifetimes and determines when objects can be deallocated. For each method, several quantities are characterized via recurrence equations: these include total memory allocation, active memory (memory that is allocated by a method and cannot yet be deallocated when the method exits), and peak heap space consumption. Also in the setting of Java, Braberman et al. [2008, 2006] and Garbervetsky et al. [2011] synthesize a

formula that bounds the amount of memory allocated by a method, as a function of its parameters. The placement of deallocation operations is inferred, but is scope-based. The focus seems to be on loop nests; recursion is not supported. In the setting of a pure, first-order functional language with garbage collection, Unnikrishnan and Stoller [2009] infer recurrence equations that describe the heap space consumption of a function, expressed as a function of the sizes of its arguments. They rely on the fact that, when a function terminates, the objects that it has allocated are either unreachable or reachable through its result.

Program Logics. Several program logics that keep track of time or of a user-defined notion of cost have been proposed. For instance, 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] extends 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. These systems can be used to count memory allocations, but do not model garbage collection.

A remarkable property of Separation Logic is that it is applicable both to low-level programming languages where memory deallocation is explicit and to high-level languages equipped with garbage collection. An interesting middle ground is studied by Hur et al. [2011], who wish to reason about low-level code that must interact with the garbage collector of a high-level language. This requires stating exactly what invariant the garbage collector relies upon and when this invariant must hold. Hur *et al.* introduce a distinction between the physical heap and the logical heap, and require that their reachable parts be isomorphic. We adopt a very similar setup. This resemblance is partly coincidental, as the two setups in fact serve slightly different purposes. Hur *et al.* aim to bridge the gap between a physical heap where the address of a block may change over time, and where a block may be deallocated, and a logical heap where a block is never moved or deallocated. We aim to bridge the smaller gap between a physical heap where blocks are deallocated (never moved) by the garbage collector and a logical heap where blocks are manually deallocated (never moved) by the user as part of her reasoning about the program.

Incorrectness Separation Logic [Raad et al. 2020] introduces a negative heap assertion $\ell \not\mapsto$, which means that the location ℓ has been deallocated (and not reallocated). This assertion is used to detect use-after-free bugs. It is not duplicable, therefore somewhat different from our persistent assertion $\dagger\dagger\{\ell\}$. Indeed, our locations are never recycled, whereas Raad *et al.* work in a lower-level setting where a freed location can be reallocated.

Kassios and Kritikos [2013] pioneer the use of pointed-by assertions in Separation Logic. The manner in which pointed-by assertions are used in the reasoning rule for assignment is the same in our paper as in theirs. There are minor design differences: their pointed-by assertions involve sets, whereas we use multisets; their pointed-by assertions cannot be split, whereas ours can; their pointed-by assertions are per-field and optional, whereas ours are mandatory and not per-field. Kassios and Kritikos use pointed-by assertions to verify concurrent copy-on-write lists, a data structure where reference counts are used to determine whether an object should be copied before it is updated. It would be interesting to transport the specification and proof of this data structure to SL$\diamond$ and to enrich its specification with space usage guarantees.

8 CONCLUSION

We have presented SL$\diamond$, a Separation Logic that allows reasoning about the heap space usage of a program (or program component) in the presence of tracing garbage collection. We believe that it is the first logic of this kind.

Space credits, which can be understood as permissions to allocate memory or as witnesses that memory is available, are consumed at allocation sites and produced at deallocation sites. Their use

in pre- and postconditions allows describing how much space a function needs, consumes, or frees up. Because garbage collection is automatic, *deallocation is a ghost operation*: it is not explicit in the source code. The person who verifies the code may use this operation wherever desired, but must prove that the objects that are about to be *logically deallocated* are unreachable. For this purpose, the logic keeps track of the predecessors of each object via *pointed-by assertions*. By default, these assertions keep track of a multiset of predecessors. However, in some cases, one can get away with less precise, therefore more tractable, predecessor information. For instance, we propose a set of rules that keep track of the *number* of predecessors, a *ghost reference count*. In future work, we would like to investigate a way of recording what *regions* the predecessors inhabit, without necessarily keeping track of the number or identity of the predecessors.

Although we give the user a programming language with garbage collection, we require her to be explicit about memory deallocation while verifying a program, and we impose on her the burden of keeping track of predecessors. Our work could be criticized on this basis: the reader may wonder, why not use a programming language with manual memory management? Our answer is, there are good reasons why many programmers choose a language with garbage collection: simplicity, safety, efficiency, guaranteed collection of all unreachable objects come to mind. Not every component in a program is verified. One pays the cost of reasoning about predecessors and deallocation only at verification time, and one chooses which program components are most in need of verification.

We have proved the soundness of SL$\diamond$ using the Coq proof assistant [Madiot and Pottier 2021]. Because we have defined SL$\diamond$ as an instance of Iris [Jung et al. 2018], it inherits all of the power of Iris, including support for shared-memory concurrency, shared invariants, ghost state, and more.

Although we believe that SL$\diamond$ is an important step forward, much remains to be done. First, SpaceLang is low-level. It requires values to be explicitly stored in mutable stack cells; it sometimes requires writing a unit value into a stack cell so as to indicate that this cell is no longer a root; it uses a somewhat odd call-by-reference discipline; and it does not have local functions with free variables. In future work, we would like to propose a program logic for a higher-level language, such as call-by-value λ-calculus, either via a direct definition or via a translation down to SpaceLang. Second, SL$\diamond$, too, is low-level: in particular, keeping track of predecessors can be cumbersome. We believe that introducing abstractions, such as the number of predecessors, or the regions that they inhabit, is a promising way of making this task more tractable. Still, more experience is needed in order to identify the most lightweight and most modular way of keeping track of predecessors in specifications and proofs. Third, although we have verified a few very small functions, we lack experience with SL$\diamond$. We would like to verify more challenging examples, including sequential and concurrent data structures.

SL$\diamond$ allows verifying that the size of a program's live data cannot exceed a certain bound S. However, from this fact, one cannot immediately conclude that a certain heap size N is sufficient for the program to be safely executed. Indeed, the relationship between the size of the live data and the heap size depends on which garbage collection algorithm is chosen. For instance, if a copying collector [Cheney 1970] is used, then setting $N = 2S$ suffices. Ideally, it seems desirable to use an implementation of garbage collection whose correctness and memory requirement have been verified: several such implementations are described in the literature [Hawblitzel and Petrank 2010; McCreight et al. 2010, 2007; Sandberg Ericsson et al. 2019; Wang et al. 2019]. It would be particularly interesting to adapt SL$\diamond$ to DataLang, an intermediate language of the CakeML compiler. Gómez-Londoño et al. [2020] have equipped DataLang with a formal cost semantics and have proved that the CakeML compiler and garbage collector respect this semantics. Developing a variant of SL$\diamond$ for DataLang would offer CakeML programmers a fully trustworthy methodology for verifying the space usage of their programs.

REFERENCES

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2007. Heap space analysis for Java bytecode. In *International Symposium on Memory Management*. 105–116.

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2009. Live heap space analysis for languages with garbage collection. In *International Symposium on Memory Management*. 129–138.

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2010. Parametric inference of memory requirements for garbage collected languages. In *International Symposium on Memory Management*. 121–130.

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

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 Adriana B. Compagnoni. 2003. Heap-Bounded Assembly Language. *Journal of Automated Reasoning* 31, 3-4 (2003), 261–302.

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).

Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *International Conference on Functional Programming (ICFP)*. 213–225.

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

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

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

Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. 2006. A Static Analysis for Synthesizing Parametric Specifications of Dynamic Memory Consumption. *Journal of Object Technology* 5, 5 (2006), 31–58.

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.

Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 116:1–116:34.

Chris J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (1970), 677–678.

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.

George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657.

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.

Diego Garbervetsky, Sergio Yovine, Víctor A. Braberman, Martín Rouaux, and Alejandro Taboada. 2011. Quantitative dynamic-memory analysis for Java. *Concurrency and Computation Practice and Experience* 23, 14 (2011), 1665–1678.

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.

Chris Hawblitzel and Erez Petrank. 2010. Automated Verification of Practical Garbage Collectors. *Logical Methods in Computer Science* 6, 3 (2010).

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.

John Hughes and Lars Pareto. 1999. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *International Conference on Functional Programming (ICFP)*. 70–81.

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.

Iris. 2021. `iris.base_logic.lib.gen_heap`. https://plv.mpi-sws.org/coqdoc/iris/iris.base_logic.lib.gen_heap.html.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Principles of Programming Languages (POPL)*. 223–236.

Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *Formal Methods (FM) (Lecture Notes in Computer Science)*, Vol. 5850. Springer, 354–369.

Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *Journal of Automated Reasoning* 59, 1 (2017), 87–120.

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.

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

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.

Jean-Marie Madiot and François Pottier. 2021. A Separation Logic for Heap Space under Garbage Collection (Repository). https://gitlab.inria.fr/fpottier/diamonds, DOI: https://doi.org/10.5281/zenodo.5549765.

Andrew McCreight, Tim Chevalier, and Andrew P. Tolmach. 2010. A certified framework for compiling and executing garbage-collected languages. In *International Conference on Functional Programming (ICFP)*. 273–284.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. 2007. A general framework for certifying garbage collectors and their mutators. In *Programming Language Design and Implementation (PLDI)*. 468–479.

Yasuhiko Minamide. 1999. Space-Profiling Semantics of the Call-by-Value Lambda Calculus and the CPS Transformation. *Electronic Notes in Theoretical Computer Science* 26 (1999), 105–120.

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

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. 2007. Automated Verification of Shape and Size Properties Via Separation Logic. In *Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science)*, Vol. 4349. Springer, 251–266.

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.

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

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

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning* 63, 2 (2019), 463–488.

Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *International Conference on Functional Programming*

(ICFP). 165–176.

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2010. Space profiling for parallel functional programs. *Journal of Functional Programming* 20, 5-6 (2010), 417–461.

Mads Tofte and Jean-Pierre Talpin. 1994. Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Principles of Programming Languages (POPL)*. 188–201.

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

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

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor. 2019. Certifying graph-manipulating C programs via localizations within data structures. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 171:1–171:30.