Local Reasoning about a Copying Garbage Collector*

NOAH TORP-SMITH and LARS BIRKEDAL
IT University of Copenhagen
and
JOHN C. REYNOLDS
Carnegie Mellon University

We present a programming language, model, and logic appropriate for implementing and reasoning about a memory management system. We state semantically what is meant by correctness of a copying garbage collector, and employ a variant of the novel separation logics to formally specify partial correctness of Cheney's copying garbage collector in our program logic. Finally, we prove that our implementation of Cheney's algorithm meets its specification, using the logic we have given, and auxiliary variables.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance—*documentation*; D.2.8 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, Logics of programs, Specification techniques*

General Terms: Realiability, Theory, Verification

Additional Key Words and Phrases: Separation Logic, Copying Garbage Collector, Local Reasoning

1. INTRODUCTION

Formal reasoning about programs that manipulate imperative data structures involving pointers has proven to be very difficult, mainly due to a lack of reasoning principles that are adequate and simple at the same time. Recently, Reynolds, O'Hearn, and others have suggested *separation logic* as a tool for reasoning about programs involving pointers; see [Reynolds 2002] for a survey and historical remarks. In his dissertation, Yang showed that separation logic is a promising direction by giving an elegant proof of the non-trivial Schorr-Waite graph marking algorithm [Yang 2001]. One of the key features making separation logic a promising tool is that it supports *local reasoning*: when specifying and reasoning about program fragments involving pointers, one may restrict attention to the "footprint"

* An extended abstract of the present paper appeared in the proceedings of POPL'04 [Birkedal etal. 2004].

Lars Birkedal's and Noah Torp-Smith's research was partially supported by Danish Natural Science Research Council Grant 51–00–0315 and Danish Technical Research Council Grant 56–00–0309. John Reynolds's research was partially supported by an EPSRC Visiting Fellowship at Queen Mary, University of London, by National Science Foundation Grant CCR-0204242, and by the Basic Research in Computer Science Centre of the Danish National Research Foundation.

of the programs, that is, to just that part of memory that the program fragments read from or write to.

The aim of this paper is to further explore the idea of local reasoning and its realization in separation logic. To this end we prove correctness of Cheney's copying garbage collector [Cheney 1970] via local reasoning in separation logic. There are several reasons to focus on Cheney's algorithm:

— The algorithm involves interesting imperative manipulations of data; in particular, it not only updates an existing data structure as the Schorr-Waite algorithm does, but *relocates* the structure. Moreover, it simultaneously treats the same data as a set of records linked by pointers and as an array of records.

— Cheney's algorithm copies any kind of data, including cyclic data structures.

— Variants of the algorithm are used in practice, e.g., in runtime systems for implementations of functional programming languages.

There are two other motivating factors that we call attention to. First, our analysis answers a question in the literature and thus paves the way for important future work that so far has been out of reach: In [Calcagno etal. 2003], local reasoning and separation logic for a garbage collected language is analyzed. An underlying garbage collector is presumed in the operational semantics of the language, inasmuch as a *partial pruning* and α*-renaming* (i.e., relocation) of the current state is allowed at any time during execution of a program. In [Calcagno etal. 2003] it is not mentioned *how* this pruning and renaming should be done, let alone proven that it is done correctly. A remark at the end of the paper expresses the desirability of such a proof — we provide one here. The analyses in [Calcagno etal. 2003] and the present paper are at two distinct levels: the former is at the level of a user language using a runtime system (a garbage collector), the latter is at the level of a runtime system providing operations for the user language (memory allocation and garbage collection). We believe these analyses pave the way for an investigation of the correctness of *combinations* of user level programs and runtime systems. We present some preliminary ideas in this direction in Sec. 9.

The additional motivating factor is that our analysis of garbage collection should be of use in connection with foundational proof-carrying code [Appel 2001] and typed assembly language [Morrisett etal. 1999]. In these settings, a memory allocation (but no deallocation) construct is part of the instruction set and a memory management system is implicitly assumed. We believe that our correctness proof can contribute to mimicking the work of [Morrisett etal. 1999] in a more realistic setting, for more machine-like assembly languages.

1.1 Contributions and Methodology

In Sec. 3 we present our storage model and the syntax and semantics of assertions. As usual in separation logic, a *state* consists of a *stack* and a *heap*, where a *stack* is a finite map from variables to values, and a *heap* is a finite map from locations to integers. A new feature is that our values include finite sets and relations of pointers and integers, which are used to give the semantics of assertions and of auxiliary variables [Owicki and Gries 1976]. Our storage model is very concrete and close to real machines; it treats *locations* as multiples of four (A datum other than a location can be easily be encoded as a number that is not a multiple of

four. More precisely, record fields containing such a number will not be altered by the algorithm). This is similar to what is often used in real implementations of runtime systems for compilers. For simplicity we assume that heaps consist only of **cons**-cells, aligned such that the first field is always on a location divisible by eight; hence *pointers* (to **cons**-cells) are multiples of eight. These assumptions and definitions induce a semantic notion of what it means for a heap to be a garbage collected version of another. This is presented in Section 3.2. Our definitions are based on the analysis in [Calcagno etal. 2003] (already referred to above) and thus involve generalizations of pruning and α-renaming of program states.

Our assertion language, presented in Sec. 3.4, is an extension of separation logic [Reynolds 2002] with new assertion forms for finite sets and relations. These are crucially used to express part of the specification of Cheney's algorithm; in particular the existence of an isomorphism between pointers to old cells and pointers to copies of the old cells – which exist at different points in time (before and after execution of the algorithm) – is established using the new assertion forms. We believe the methodology of using sets and relations can be used more widely, to specify and prove correct other programs involving shared or cyclic structures. Indeed a somewhat similar approach is being used by Richard Bornat [Bornat 2003] to specify and verify an algorithm for copying directed acyclic graphs.

Moreover, we have extended the iterated separating conjunction [Reynolds 2002] of separation logic to arbitrary finite sets. The assertion $\forall_* x \in m.\ A$ holds in a state (s, h), if m denotes a finite set $\{p_1, \ldots, p_k\}$ and $A[p_1/x] * \cdots * A[p_k/x]$ holds in (s, h) (see Fig. 2 for a precise definition). As illustrated in Sec. 5, one can specify a program by separating the locations it manipulates into disjoint sets with different properties, and then use the iterated separating conjunction, together with expressions for finite sets and relations, to express the particular properties of each set.

In Sec. 4 we define the syntax and semantics of the programming language used to implement the garbage collector. It is a simple imperative programming language, with constructs for heap lookup and heap update (but no constructs for allocation or disposal of heap cells). The associated program logic is presented in Sec. 4.2. The program logic is mostly standard except for the new rules regarding sets and relations, and it includes the frame rule of separation logic which makes local reasoning possible, as explained in Sec. 4.2.

Cheney's algorithm and the specification of our implementation thereof are presented in Sec. 5; the implementation itself is included in Appendix A. In Sec. 5.1 we define a semantic condition on programs which implies that a program is a correct garbage collector and show that this condition ensures that the heap after execution of the program is a garbage collected version (in the sense of Sec. 3.2) of the heap before execution if the latter satisfies certain requirements mentioned in Sec. 3.2.

We present an informal analysis of the algorithm and use it to derive a formal specification of an invariant. At any point of execution, the pointers manipulated by the algorithm can be divided into disjoint sets in which the elements have the same property. Thus it is natural to use the method of sets and relations along with the iterated separating conjunction mentioned before. The sets and relations are also used in another crucial way, namely to record the initial contents of the heap (before garbage collection). This makes it possible to relate the final heap (after garbage

collection) to the initial heap and prove that the final heap is a garbage collected version of the initial heap. As mentioned, we use these to give a specification for our implementation formulated in the program logic from Sec. 4.2, and before we formally prove that implementation meets the specification (in Sec. 6), we show in Sec. 5.4 that the resulting specification ensures that the implementation is indeed a correct garbage collector.

We emphasize the following point. Cheney's algorithm assumes two contiguous "semi-heaps" of equal size, OLD and NEW, and works by copying all *reachable* data from OLD into NEW. One of the reasons for the popularity of Cheney's algorithm (and variants thereof) is that it runs in time proportional to the size of the reachable data; it never touches unreachable cells. This fact is reflected directly in our specification of the algorithm, which refers to the reachable part of OLD only. It is in the spirit of local reasoning to have such a direct correspondence between the intuitive understanding of an algorithm and its formal specification.

2. AN INTRODUCTION TO SEPARATION LOGIC

In this section, we give a brief introduction to separation logic. Formal definitions of the concepts used here will not be given, since it would clutter the presentation, and since we will extend traditional separation logic with a few constructs later. For formal definitions, we refer to Sections 3 and 4.

Separation logic is an extension of traditional Hoare logic [Hoare 1969]. The simple **while**-language is extended with commands for manipulating imperative data structures, stored in a *heap*, and if "dangling" pointers are dereferenced, the semantics for the language will get "stuck", or "abort". Accordingly, the assertion language is extended with basic predicates concerning the heap, and two new connectives: the *separating conjunction* $*$ and the *separating implication* $-\!\!*$ (in this paper we will not use the separating implication).

We have specifications $\{A\}\ C\ \{B\}$, stating that in any state in which A holds, no execution of C will abort, and if the execution terminates in a final state, then B will hold in that state. As a consequence, we have the slogan

> *"Well-specified programs do not go wrong."*

for separation logic.

2.1 An Example

As mentioned in Sec. 1, separation logic provides reasoning principles for proving programs that manipulate shared mutable data structures. We will demonstrate the advantage of separation logic by an example.

In traditional Hoare logic (without shared data structures), one has the *rule of constancy* (as usual, $\mathrm{Mod}(C)$ is the set of variables modified by the command C).

$$\frac{\{A\}\ C\ \{A'\}}{\{A \wedge B\}\ C\ \{A' \wedge B\}}\ \mathrm{Mod}(C) \cap \mathrm{FV}(B) = \emptyset$$

This rule has been useful, since it has allowed reasoning about only the parts of the store that are modified by the program fragment. In the presence of aliasing, however, the rule of constancy is *not* sound, as can be seen from the following

counterexample.

$$\frac{\{x \hookrightarrow 3\}\ [x] := 4\ \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 \wedge y \hookrightarrow 3\}\ [x] := 4\ \{x \hookrightarrow 4 \wedge y \hookrightarrow 3\}}\ ,$$

where the effect of $[x] := 4$ is that 4 is stored in the heap at the address denoted by x. The problem here is that x and y might be references to the same heap cell, and if we change the value stored in this heap cell, the assertion about what y points to does not remain true. In Hoare logic, we would have to add a premise like $x \neq y$, or some other non-interference predicate, for the conclusion to hold. This may seem like a small thing to do, but if there are many variables in play, the induced number of non-interference predicates in the involved assertions would quickly become intractable. Further, if a program deals with more realistic data structures, the assertions which prevent sharing also become unacceptably complex, as can be seen from the examples in [Reynolds 2002]. In contrast, the assertion

$$x \hookrightarrow 3 * y \hookrightarrow 3 \tag{1}$$

in separation logic *implicitly* states that x and y are disjoint, since the two assertions $x \hookrightarrow 3$ and $y \hookrightarrow 3$ must hold in *disjoint* parts of the heap in order for (1) to hold. Therefore, the derivation

$$\frac{\{x \hookrightarrow 3\}\ [x] := 4\ \{x \hookrightarrow 4\}}{\{x \hookrightarrow 3 * y \hookrightarrow 3\}\ [x] := 4\ \{x \hookrightarrow 4 * y \hookrightarrow 3\}}$$

is valid in separation logic. In fact, it is an instance of the important *frame rule*:

$$\frac{\{A\}\ C\ \{A'\}}{\{A * B\}\ C\ \{A' * B\}}\ \mathrm{Mod}(C) \cap \mathrm{FV}(B) = \emptyset$$

This rule allows *local reasoning*. Suppose we have a program with a **while**-loop which performs some manipulations on a data structure stored in the heap. When verifying the program, one has to exhibit an invariant for the **while** loop and prove that it is indeed an invariant. The invariant is typically an assertion about the full data structure, whereas each iteration of the loop manipulates a small portion of this structure only. The Frame Rule allows us to prove the invariant by proving a specification which mentions only the parts of the heap that are actually manipulated in one loop iteration (this has been called the *footprint* of the code fragment by O'Hearn [Reynolds 2002]), and then conclude the specification regarding the full structure from this. We will see several applications of the frame rule in subsequent sections. Also, we discuss benefits of separation logic in general terms in Section 7.1, after we have shown correctness of our garbage collector.

3. SYNTAX AND SEMANTICS

In this section we present our storage model, and define a semantic notion of garbage collection. We then proceed with the syntax and semantics of expressions and assertions which are part of our program logic. The basis of the system is the standard separation logic with pointer arithmetic [O'Hearn etal. 2001], but we extend the expression and assertion languages with finite sets and relations, new basic assertions about these, and an extension of the iterated separating conjunction to arbitrary finite sets.

3.1 Storage Model

We assume five countably infinite sets Var^{int}, Var^{fs}, Var^{frp}, Var^{fri} of variables, and let Var be the disjoint union of these sets. We let metavariables $x, y, \ldots$ range over Var and assume a type-function

$$\tau : Var \to \mathsf{types}, \text{ where } \mathsf{types} = \{\mathsf{int}, \mathsf{fs}, \mathsf{frp}, \mathsf{fri}\}$$

indicating which type a given variable has. The sets of *locations* and *pointers* are the sets of integers divisible by 4 and 8, respectively. More formally, we define:

$$\begin{array}{lrcl}
\text{Variables} & x, y, \ldots & \in & Var \\
\text{Locations} & l \in Loc & \equiv & \{4n \mid n \in \mathbf{Z}\} \\
\text{Pointers} & p \in Ptr & \equiv & \{8n \mid n \in \mathbf{Z}\} \\
\text{Finite sets} & FS & \equiv & \mathcal{P}_{fin}(Ptr) \\
\text{Pointer relations} & FRP & \equiv & \mathcal{P}_{fin}(Ptr \times Ptr) \\
\text{Integer relations} & FRI & \equiv & \mathcal{P}_{fin}(Ptr \times \mathbf{Z}) \\
\text{Values} & v \in Val & \equiv & \mathbf{Z} \cup FS \cup FRP \cup FRI \\
Heaps & \equiv & Loc \rightharpoonup_{fin} \mathbf{Z} & \\
Stacks & \equiv & \{s : Var \rightharpoonup_{fin} Val \mid \forall x \in Var.\ s(x) \in [\![\tau(x)]\!]\} & \\
States & \equiv & Stacks \times Heaps, &
\end{array}$$

where $[\![\mathsf{int}]\!] = \mathbf{Z}$, $[\![\mathsf{fs}]\!] = FS$, $[\![\mathsf{fri}]\!] = FRI$, and $[\![\mathsf{frp}]\!] = FRP$. We use the notation $Loc \rightharpoonup_{fin} \mathbf{Z}$ to denote the set of finite partial functions from Loc to $\mathbf{Z}$. As mentioned in Section 1.1, we will assume that data is arranged in **cons** cells that are aligned in such a way that the first field of each cell is located at a location that is divisible by 8, and this is why we define the subset Ptr of locations which point to such **cons** cells.

The finite sets and both kinds of relations in the table above are extensions of traditional separation logic. In the sense of [Reynolds 1981] and [Owicki and Gries 1976], variables of these types are used as auxiliary and ghost variables in the garbage collector and the proof of its correctness. This means that they are not necessary for the program to work, but they ease proofs of its properties.

Before we present the expression and assertion languages of our program logic, we define several concepts related to heaps; these are needed for the definition of a correct garbage collector.

3.2 What is a Garbage Collected Heap?

To implement and reason about a garbage collector, we must make some assumptions about heaps, and the assumptions will be part of our logic. For instance, if a heap contains dangling reachable pointers, then our garbage collector will malfunction, and hence we must make assumptions that prevent this.

In what follows, we therefore define the concept of a "garbage collected heap", and what it means for a heap to be a garbage collected version of another. First, we give a more precise definition of our requirement about aligned **cons**-cells. Let S be a finite set of pointers, let $f_1, f_2 : Ptr \rightharpoonup \mathbf{Z}$ be finite partial functions from pointers to integers, and let h be a heap. We then define the semantic condition

pairheap by

$$\begin{array}{rl} \mathit{pairheap}(S, f_1, f_2, h) \text{ iff } & \mathrm{dom}(f_1) = \mathrm{dom}(f_2) = S \\ & \text{and } \mathrm{dom}(h) = \{p + k \mid p \in S \text{ and } k \in \{0, 4\}\} \\ & \text{and } \forall p \in S.\ h(p) = f_1(p) \text{ and } h(p + 4) = f_2(p). \end{array}$$

Note that if $\mathit{pairheap}(S, f_1, f_2, h)$ holds, then h is determined by f_1, f_2, and S (and S, f_1, f_2 are determined by h).

The concept of garbage collection depends on a given *root set*. Here, we assume for simplicity that there is only one root cell. Thus, given a heap h and a pointer r, we call (r, h) a *rooted heap*. Given such a rooted heap (r, h), we formally define the sets $rp(r, h)$ and $rl(r, h)$ of pointers and locations that are *reachable* from r in h as follows:

$$\begin{array}{rcl} rp_0(r, h) & \equiv & \{r\} \\ rp_{n+1}(r, h) & \equiv & \{h(\ell) \mid \ell \in rl_n(r, h) \cap \mathrm{dom}(h) \text{ and } h(\ell) \in \mathit{Ptr}\} \\ rl_n(r, h) & \equiv & \{p + k \mid p \in rp_n(r, h) \text{ and } k \in \{0, 4\}\} \\ rp(r, h) & \equiv & \bigcup_{n=0}^{\infty} rp_n(r, h) \\ rl(r, h) & \equiv & \bigcup_{n=0}^{\infty} rl_n(r, h). \end{array}$$

With these definitions, we can define the condition that "no reachable location dangles":

$$rl(r, h) \subseteq \mathrm{dom}(h),$$

and that "all locations are reachable":

$$rl(r, h) \supseteq \mathrm{dom}(h).$$

If both these conditions hold, *i.e.*, if $rl(r, h) = \mathrm{dom}(h)$, we say that the rooted heap (r, h) is *exactly reachable*. The reachable pointers and locations satisfy a monotonicity property:

$$h_0 \subseteq h \text{ implies } rl(r, h_0) \subseteq rl(r, h). \tag{2}$$

Furthermore, once there are no dangling reachable locations in a heap, further enlargements of the heap does not increase reachability:

$$h_0 \subseteq h \text{ and } rl(r, h_0) \subseteq \mathrm{dom}(h_0) \text{ implies } rl(r, h_0) = rl(r, h). \tag{3}$$

It is not hard to see that if a rooted heap (r, h) has no dangling reachable locations, then h contains a unique subheap h_0 such that (r, h_0) is exactly reachable, and otherwise, h does not contain a subheap h_0 such that (r, h_0) is exactly reachable. The subheap (r, h_0) plays a role similar to that of $\mathit{prune}(h)$ from the article [Calcagno etal. 2003], but it is only well-behaved if there are no reachable dangling locations in (r, h).

For any heap h, we define

$$\mathrm{pdom}(h) \equiv \mathrm{dom}(h) \cap \mathit{Ptr}.$$

Now, assume (r, h) is exactly reachable. Then for all pointers p,

$$p \in \mathrm{pdom}(h) \text{ iff } p \in \mathrm{dom}(h) \text{ iff } p + 4 \in \mathrm{dom}(h), \tag{4}$$

and for all locations ℓ,

$$\ell \in \mathrm{dom}(h) \text{ and } h(\ell) \in \mathit{Ptr} \text{ implies } h(\ell) \in \mathrm{pdom}(h), \tag{5}$$

and

$$r \in \mathrm{pdom}(h). \tag{6}$$

Conversely, if the conditions (4) to (6) hold for a rooted heap (r, h), then one can show

$$rp(r, h) \subseteq \mathrm{pdom}(h) \quad \text{and} \quad rl(r, h) \subseteq \mathrm{dom}(h)$$

by induction on n to the corresponding formulas where rp and rl are subscripted with n.

The conditions (4) to (6) are just what is needed to define a *heap morphism* from (r, h) to an arbitrary rooted heap. Henceforth, assume (r, h) satisfies (4) to (6), (r', h') is an arbitrary rooted heap, and that β is a function from $\mathrm{pdom}(h)$ to $\mathrm{pdom}(h')$ such that for all $p \in \mathrm{pdom}(h)$ and $k \in \{0, 4\}$,

$$h'((\beta(p)) + k) = \beta^*(h(p + k)) \tag{7}$$

and

$$r' = \beta(r). \tag{8}$$

Here, β^* is the extension of β to $\mathbf{Z}$ that is the identity on numbers that are not in $\mathrm{pdom}(h)$. Then β is called a *heap morphism* from (r, h) to (r', h').

LEMMA 3.1. *Assume (r, h) satisfies (4) to (6), (r', h') is an arbitrary rooted heap, and β is a heap morphism from (r, h) to (r', h'). Then,*

—*for all pointers p, if $p \in rp(r, h)$, then $\beta(p) \in rp(r', h')$.*

—*If β is an isomorphism of functions, then for all pointers p', if $p' \in rp(r', h')$, then $p' \in \mathrm{pdom}(h')$ and $\beta^{-1}(p') \in rp(r, h)$.*

—*If β is an isomorphism of functions and (r, h) is exactly reachable, then (r', h') is exactly reachable.*

PROOF. For the first claim, we prove the slightly stronger statement that for all pointers p, if $p \in rp_n(r, h)$, then $\beta(p) \in rp_n(r', h')$. This is done by induction on n. If $p \in rp_0(r, h)$, then $p = r$ so that $\beta(p) = \beta(r) = r' \in rp_0(r', h')$. For the induction step, if $p \in rp_{n+1}(r, h)$, there is a pointer $q \in rp_n(r, h)$ and a $k \in \{0, 4\}$ with $p = h(q + k)$. Moreover, $p, q \in \mathrm{dom}(h)$, and by induction, $\beta(q) \in rp_n(r', h')$. This implies

$$\beta(p) = \beta(h(q + k)) = \beta^*(h(q + k)) = h'(\beta(q) + k),$$

and thus, $\beta(p) \in rp_{n+1}(r', h')$.

The second claim is also proved by induction on n. More precisely, we show that if β is an isomorphism of functions (*i.e.*, it is a bijection between $\mathrm{pdom}(h)$ and $\mathrm{pdom}(h')$), then for all natural numbers n and all pointers p', if $p' \in rp_n(r', h')$, then $p' \in \mathrm{pdom}(h')$ and $\beta^{-1}(p') \in rp_n(r, h)$. The base case is obvious, and for the induction step, suppose $p' \in rp_{n+1}(r', h')$. Then there is a pointer $q' \in rp_n(r', h')$ and a $k \in \{0, 4\}$ with $p' = h'(q' + k)$. For this q', the induction hypothesis yields $q' \in \mathrm{pdom}(h')$ and $\beta^{-1}(q') \in rp_n(r, h)$, and thus

$$p' = h'(q' + k) = h'((\beta(\beta^{-1}(q')) + k)) = \beta^*(h(\beta^{-1}(q') + k)) = \beta(h(\beta^{-1}(q') + k)),$$

where the last step holds since p' is a pointer. This means that $p' \in \mathrm{pdom}(h')$, and

$$\beta^{-1}(p') = \beta^{-1}(\beta(h(\beta^{-1}(q') + k))) = h(\beta^{-1}(q') + k).$$

Now, since $\beta^{-1}(q') \in rp_n(r,h)$, $\beta^{-1}(p') \in rp_{n+1}(r,h)$.

Finally, also suppose that (r,h) is exactly reachable. To show that (r',h') is exactly reachable, let $\ell' \in rl(r',h')$. By definition, there are a pointer $p' \in rp(r',h')$ and a $k \in \{0,4\}$ with $\ell' = p' + k$. By the previous proof, $p' \in \mathrm{pdom}(h')$ and $\ell' \in \mathrm{dom}(h')$. Conversely, if $\ell' \in \mathrm{dom}(h')$, then $\ell' = p' + k$ for a $k \in \{0,4\}$ and a $p' \in \mathrm{pdom}(h')$. Since β is a bijection, $\beta^{-1}(p') \in \mathrm{pdom}(h)$, and thus $\beta^{-1}(p') \in rp(r,h)$, since (r,h) is exactly reachable. We can now use the proof of the first item to conclude the desired result. □

The Lemma justifies the following semantic notion of one heap being a garbage collected version of another. This, of course, is central when verifying correctness of a garbage collector.

DEFINITION 3.2. *Let $(r,h),(r',h')$ be rooted heaps, and suppose $h_0 \subseteq h$ is such that (r,h_0) is exactly reachable. If there is a heap morphism β from (r,h_0) to (r',h'_0) for some subheap $h'_0 \subseteq h'$, then (r',h') is called a* garbage collected version *of (r,h).*

This notion of garbage collection, of course, depends on the fact that we collect **cons** cells only, and it does not imply that h and h' need have the same size. Indeed, a memory management system may shrink the heap after garbage collection, to diminish the heap usage of a program, or it may allocate more space for a program, to reduce the required number of rounds of garbage collection.

With this notion of garbage collection, we also note that the identity is an example of a heap morphism, and the trivial program **skip** is thus an example of a correct garbage collector. However, our interest lies with non-trivial garbage collectors.

Our notion of garbage collection is also quite conservative. Indeed, it has been noted in the literature [Aditya etal. 1994; Morrisett etal. 1995] that if the programming language that uses the garbage collector has a sufficiently rich type system, there can be a significant difference between the data that is reachable and data that can be reclaimed without affecting the program. This is because some objects in the heap might be reachable, but we might be able to infer information about reachable objects in the heap at run-time, (e.g. via type reconstruction [Aditya and Caro 1993]) which implies that these objects are not necessary for execution of the rest of the program. In this case we say that these objects are reachable, but not *live*, and it would, of course, give better memory usage to collect objects that are "dead" in this sense. Using region inference [Tofte and Talpin 1994; Tofte and Birkedal 1998] it is also possible to statically infer that some data is dead, although it is reachable, and combinations of region inference and garbage collection exist in current run-time systems for ML [Tofte etal. 2004; Hallenberg etal. 2002]. It might even be possible to employ such techniques as hash-consing [Goto 1974; Appel and Gonçalves 1993] or other dynamic updates that affect data representation [Stoyle etal. 2005]. However, these possibilities are not relevant to the goal of this paper, which is a specification and proof of the Cheney algorithm in separation logic.

3.3 Expressions

We define the syntax and semantics for expressions of each of the types $\mathsf{int}, \mathsf{fs}, \mathsf{frp}$, and fri. For expressions of type int we just present the syntax; the semantics is straightforward. For expressions of the remaining types, we just present the semantics as the syntax will be evident from the presentation of the semantics. In general, the semantics for an expression E (of each of the types just mentioned) is formally a map from stack s with $\mathrm{FV}(E) \subseteq \mathrm{dom}(s)$ to values of suitable kinds, *i.e.*, the semantics of an expression depends on a stack.

Expressions of type int are defined by the following grammar:

$$\begin{aligned} e \ ::= \ & n \mid x^{\mathsf{int}} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \mid e \bmod j \\ & \mid e_1 \leq e_2 \mid e_1 = e_2 \mid \neg e \mid e_1 \wedge e_2 \mid \#m^{\mathsf{fs}}, \end{aligned}$$

where $n \in \mathbf{Z}$ and $j \in \mathbf{N} \setminus \{0\}$. The semantics of $\#m^{\mathsf{fs}}$ is the number of elements in the finite set of pointers denoted by m^{fs} (see below). In order to avoid introducing an explicit type of boolean values, we use a standard encoding of truth values, where 0 denotes "false", and all other integers denote "true". Although the superscript that indicates the type is only meant to indicate the type of variables, we sometimes use a superscript to indicate the type of composite expressions. Most frequently, however, we will omit the superscripts, even on variables, if it causes no confusion.

We use m to range over expressions of type fs. The semantics of an expression of type fs is a set of pointers. For instance, the expression $Itv(e_1, e_2)$ denotes the set of pointers in the half-open interval from e_1 to e_2. In general, the semantics of an expression m of type fs is a map from stacks s with $\mathrm{FV}(m) \subseteq \mathrm{dom}(s)$ to the set FS of finite sets of pointers, and it is given by

$$\begin{array}{lll} [\![\emptyset^{\mathsf{fs}}]\!]s = \emptyset & & [\![Itv(e_1^{\mathsf{int}}, e_2^{\mathsf{int}})]\!]s = \{p \in Ptr \mid [\![e_1]\!]s \leq p \ \wedge \ p < [\![e_2]\!]s\} \\ [\![x^{\mathsf{fs}}]\!]s = s(x) & & [\![m_1^{\mathsf{fs}} \cup m_2^{\mathsf{fs}}]\!]s = [\![m_1]\!]s \cup [\![m_2]\!]s \\ [\![\{e\}]\!]s = \{[\![e]\!]s\} \cap Ptr & & [\![m_1^{\mathsf{fs}} \setminus m_2^{\mathsf{fs}}]\!]s = [\![m_1]\!]s \setminus [\![m_2]\!]s \end{array}$$

We can now formally define the semantics of expressions of form $\#m^{\mathsf{fs}}$:

$$[\![\#m^{\mathsf{fs}}]\!]s = k, \text{ where } [\![m]\!]s = \{p_1, \ldots, p_k\} \quad \text{(note that } k \text{ may be 0).}$$

We use r to range over expressions of type frp. The semantics of an expression r of type frp is a map from stacks s with $\mathrm{FV}(r) \subseteq \mathrm{dom}(s)$ to the set FRP of finite relations on pointers, and it is given by the following clauses.

$$\begin{aligned} [\![\emptyset^{\mathsf{frp}}]\!]s &= \emptyset \\ [\![x^{\mathsf{frp}}]\!]s &= s(x) \\ [\![r^{\dagger}]\!]s &= \{(p', p) \mid (p, p') \in [\![r]\!]s\} \\ [\![r_1^{\mathsf{frp}} \circ r_2^{\mathsf{frp}}]\!]s &= \{(p, p'') \mid \exists p'.\ (p, p') \in [\![r_2]\!]s \ \wedge \ (p', p'') \in [\![r_1]\!]s\} \\ [\![r^{\mathsf{frp}} \cup \{(e_1^{\mathsf{int}}, e_2^{\mathsf{int}})\}]\!]s &= [\![r]\!]s \cup (\{([\![e_1]\!]s, [\![e_2]\!]s)\} \cap Ptr \times Ptr) \\ [\![r^{\mathsf{frp}} \setminus \{(e_1^{\mathsf{int}}, e_2^{\mathsf{int}})\}]\!]s &= [\![r]\!]s \setminus (\{([\![e_1]\!]s, [\![e_2]\!]s)\}) \end{aligned}$$

Note that we use $r^{\dagger}$ for the inverse of the relation r.

To conclude our semantics for expressions, we give the semantics for expressions of type fri. We use ρ to range over such expressions. The $\odot$ operator will be used to model the structure-preserving property of our garbage collector, inasmuch as it extends a relation with the identity on non-pointers before composing it with

another relation (cf. the definition of β^* from Sec. 3.2). Hence, the semantics of an expression ρ of type fri is a map from stacks s with $\mathrm{FV}(\rho) \subseteq \mathrm{dom}(s)$ to the set FRI of relations between pointers and integers.

$$\begin{aligned}
[\![x^{\mathsf{fri}}]\!]s &= s(x) \\
[\![\rho^{\mathsf{fri}} \circ r^{\mathsf{frp}}]\!]s &= \{(p,n) \mid \exists p' \in Ptr.\ (p,p') \in [\![r]\!]s \ \wedge (p',n) \in [\![\rho]\!]s\} \\
[\![r^{\mathsf{frp}} \odot \rho^{\mathsf{fri}}]\!]s &= \{(p,n) \mid ((p,n) \in [\![\rho]\!]s \wedge n \notin Ptr) \vee \\
&\qquad (\exists p' \in Ptr.\ (p,p') \in [\![\rho]\!]s \wedge (p',n) \in [\![r]\!]s)\}
\end{aligned}$$

Note the connection between the $(-)^*$-construct and the semantics of the special relation composition $\odot$: if φ and ψ are functions denoting the relations r and ρ, respectively, then $\varphi^* \circ \psi$ is the denotation of $r \odot \rho$.

Substitution is defined in a standard way; there are no binders in the expression language. The following substitution lemma is easily proved by induction on expressions.

LEMMA 3.3. *Let s be a stack, and let $\delta, \delta' \in$ Types. Then, for all expressions $e^{\delta}, e'^{\delta'}$ of type δ and δ' respectively, and for all variables $x^{\delta'}$ of type δ',*

$$[\![e[e'/x]]\!]s = [\![e]\!](s[x \mapsto [\![e']\!]s]),$$

where $s[x \mapsto v]$ is the function that is like s, but with x mapped to v.

We use $\equiv$ to denote syntactic equality between expressions, and we sometimes write $e_1 = e_2$ to denote that $[\![e_1]\!]s = [\![e_2]\!]s$, for all stacks s with $\mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \subseteq \mathrm{dom}(s)$.

3.4 Assertions

Our assertion language is a variant of that of separation logic [Reynolds 2002] with assertion forms for finite sets and relations. We first present the syntax of assertions and give informal explanations of the most interesting assertion forms. Then we give the formal semantics and present some useful inference rules.

We use A, B, and D to range over assertions, which are generated by the following grammar:

$$\begin{array}{lll}
A, B, D ::= & e_1 \le e_2 & \mid e_1 = e_2 \\
& \mid \neg e & \mid \mathsf{T} \\
& \mid \mathsf{F} & \mid \neg A \\
& \mid A \to B & \mid A \wedge B \\
& \mid A \vee B & \mid \forall x^{\delta}.\ A \\
& \mid \exists x^{\delta}.\ A & \\
\hline
& \mid \mathsf{emp} & \mid e_1 \mapsto e_2 \\
& \mid A * B & \mid A \mathrel{-\!\!*} B \\
& \mid \forall_* x^{\mathsf{int}} \in m.\ A & \\
\hline
& \mid e \in m & \mid m_1 \perp m_2 \\
& \mid m_1 = m_2 & \mid m_1 \subseteq m_2 \\
& \mid (e_1, e_2) \in \rho & \mid (e_1, e_2) \in r \\
& \mid \mathsf{Ptr}(e) & \mid \mathsf{PtrRg}(\rho, m) \\
& \mid \mathsf{Tfun}(r, m) & \mid \mathsf{Tfun}(\rho, m) \\
& \mid \mathsf{iso}(r, m_1, m_2) & \\
\hline
& \mid \mathsf{Reachable}(\rho_1, \rho_2, m, e), &
\end{array}$$

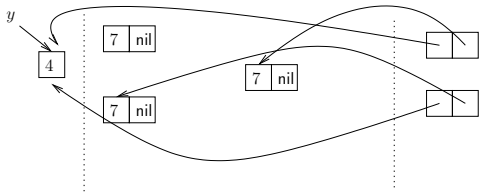

Fig. 1. A sample partition of a heap

where δ ranges over Types. We have grouped the assertion forms by horizontal lines. The assertion forms above the first horizontal line are the usual assertion forms from classical logic with equality.

The second group are assertions that describe the heap. Apart from the iterated separating conjunction, these are all part of standard separation logic. The assertion emp states that the heap is empty, and $e_1 \mapsto e_2$ states that there is precisely one location in the domain of the heap. $A * B$ means that A and B hold in *disjoint* subheaps of the current heap, and $A \mathrel{-\!\!*} B$ means that for all heaps h' that are disjoint from the current heap h and in which A hold, the combination of the extension and the current heap will satisfy B. Finally, $\forall_*$ is an *iterated separating conjunction*. Informally, if $s, h \models \forall_* x \in m.\ A$, and if $[\![m]\!]s = \{p_1, \ldots, p_k\}$, then h can be split into disjoint heaps $h = h_1 \cdot \ldots \cdot h_k$ with $s, h_i \models A[p_i/x]$. This assertion form plays a crucial role in our specification of our garbage collector. This is evident in Sections 5.2 and 5.3, but to illustrate this important new assertion form, we provide a simple example of using $\forall_*$ to describe heaps here.

Consider the heap depicted in Figure 1. The heap cells here can be split into three disjoint sets in each of which all the cells have similar properties. This partition is illustrated with dotted lines in the figure. The first set simply consists of the cell pointed to by y, whereas the second set (called m_1) consists of three cells which clearly have a similar property. The cells in the last group m_2 have the property that their first fields contain the pointer y, and the second fields all contain a pointer to a cell in m_1. Hence, a description of this heap is given by the assertion

$$\begin{aligned}\exists m_1, m_2.\ ((y \mapsto 4) * \\ (\forall_* x \in m_1.\ x \mapsto 7, \mathsf{nil}) * \\ (\forall_* x \in m_2.\ (\exists z.\ z \in m_1 \wedge x \mapsto y, z))).\end{aligned}$$

At any state of execution of the garbage collection algorithm, the pointers involved in the algorithm can be split into disjoint sets as in this example, and thus the iterated separating conjunction can be used to obtain an assertion which describes the heap, in a way similar to the example above. Also note that the sets involved in the assertion form might change throughout execution of a program; thus this assertion form describes a "dynamic" heap. An example of this can be found in Section 6.1.

The next group contains assertion forms that are related to our extension of standard separation logic. They are assertions about sets and relations, and most of them are self-explanatory. The assertion $\mathsf{PtrRg}(\rho, m)$ loosely says that m is the pointers in the range of the relation ρ; Tfun says that a relation is a total function on a set, and iso says that a relation is a bijection between two sets. Finally,

the assertion Reachable says that an exactly reachable heap is described by two (functional) relations on a set, and a "root pointer".

The set $\mathrm{FV}(A)$ of free variables for an assertion A is defined as usual. Note that x (and not m) is bound in $\forall_* x \in m.\ A$. Substitution $A[e/x]$ of the expression e for the variable x in the assertion A is defined in the standard way. We sometimes write $A(x)$ to indicate that the variable x may occur free in A.

The semantics for propositions is formally given by a judgment of the form

$$(s, h) \models A,$$

the intended meaning of which is that the proposition A holds in the state (s, h). We require $\mathrm{FV}(A) \subseteq \mathrm{dom}(s)$. Before the definition of this judgment, we need to introduce a partial commutative monoid structure on the set of heaps: Write $h_1 \# h_2$ to indicate $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ (we call such heaps *disjoint*), and if $h_1 \# h_2$, we define the combined heap $h_1 \cdot h_2$ by

$$n \mapsto \begin{cases} h_1(n) & \text{if } n \in \mathrm{dom}(h_1) \\ h_2(n) & \text{if } n \in \mathrm{dom}(h_2) \end{cases}.$$

The semantics is given in Fig. 2. We let b range over the boolean expressions $e_1 \leq e_2, e_1 = e_2, \neg e$, and δ range over Types.

Note that the semantics is classical for the standard first-order logic fragment, and that the parameters for the assertion form Reachable determine the heap that is required to exist unambiguously.

As for expressions, we have the expected substitution lemma for assertions.

LEMMA 3.4. *Let e be an expression of type δ, let x be a variable of type δ, and let A be an assertion. Then, for all states s, h,*

$$(s, h) \models A[e/x] \quad \text{iff} \quad (s[x \mapsto [\![e]\!]s], h) \models A$$

DEFINITION 3.5. *We call an assertion A* valid *if, for all states (s, h) with $\mathrm{FV}(A) \subseteq \mathrm{dom}(s)$, $(s, h) \models A$. We use $\Rightarrow$ to denote semantic validity,* i.e.*, $A \Rightarrow B$ if $(s, h) \models A$ implies $(s, h) \models B$, for all states (s, h).*

The following shorthand notations are standard in separation logic, and we shall also use them in this paper.

$$\begin{aligned} e \mapsto e_1, e_2 &\equiv (e \mapsto e_1) * (e + 4 \mapsto e_2) \\ e_1 \hookrightarrow e_2 &\equiv e_1 \mapsto e_2 * \mathsf{T} \\ e \hookrightarrow e_1, e_2 &\equiv e \mapsto e_1, e_2 * \mathsf{T} \\ e \mapsto - &\equiv \exists x^{\mathsf{int}}.\ e \mapsto x \\ e \mapsto -, - &\equiv \exists x^{\mathsf{int}}, y^{\mathsf{int}}.\ e \mapsto x, y \end{aligned}$$

These notations make sense for all locations, but we shall only use them when e denotes a pointer. We also write $e_1 \neq e_2$ for $\neg(e_1 = e_2)$.

There are certain special classes of assertions [Yang 2001],[Reynolds 2002], which we will use later. These are defined here.

DEFINITION 3.6.

—*An assertion A is called* pure *if its validity does not depend on the heap, i.e., if $(s, h) \models A$ if and only if $(s, h') \models A$, for all stacks s and heaps h, h'.*

A	$(s,h) \models A$ if and only if
b	$[\![b]\!]s \neq 0$
T	T
F	F
$\neg A$	$s,h \not\models A$
$A \to B$	$s,h \models A$ implies $s,h \models B$
$A \wedge B$	$s,h \models A$ and $s,h \models B$
$A \vee B$	$s,h \models A$ or $s,h \models B$
$\forall x^{\delta}.\ A$	for all $v \in [\![\delta]\!], s[x \mapsto v], h \models A$
$\exists x^{\delta}.\ A$	for some $v \in [\![\delta]\!],\ s[x \mapsto v], h \models A$
emp	$\mathrm{dom}(h) = \emptyset$
$e_1 \mapsto e_2$	$h = \{([\![e_1]\!]s, [\![e_2]\!]s)\}$
$A * B$	there are heaps h_1, h_2 such that $h_1 \# h_2$, $h_1 \cdot h_2 = h, s, h_1 \models A,$ and $s, h_2 \models B$
$A \mathrel{-\!\!*} B$	$s, h \cdot h' \models B$ for all h' with $h \# h'$ and $s, h' \models A$
$\forall_* p \in m.\ A$	$\begin{cases} s,h \models A[p_1/p] * \cdots * A[p_k/p] & \text{if } [\![m]\!]s = \{p_1, \ldots, p_k\} \\ s,h \models \mathsf{emp} & \text{if } [\![m]\!]s = \emptyset \end{cases}$
$e \in m$	$[\![e]\!]s \in [\![m]\!]s$
$m \perp m'$	$[\![m]\!]s \cap [\![m']\!]s = \emptyset$
$m_1 = m_2$	$[\![m_1]\!]s = [\![m_2]\!]s$
$m_1 \subseteq m_2$	$[\![m_1]\!]s \subseteq [\![m_2]\!]s$
$(e_1, e_2) \in r$	$([\![e_1]\!]s, [\![e_2]\!]s) \in [\![r]\!]s$
$(e_1, e_2) \in \rho$	$([\![e_1]\!]s, [\![e_2]\!]s) \in [\![\rho]\!]s$
$\mathsf{Ptr}(e)$	$[\![e]\!]s \in Ptr$
$\mathsf{PtrRg}(\rho, m)$	$\forall\ (p,q) \in [\![\rho]\!]s.\ q \in Ptr \Rightarrow q \in [\![m]\!]s$
$\mathsf{Tfun}(r, m)$	$\forall p \in [\![m]\!]s.\ \exists! n \in \mathbf{Z}.\ (p,n) \in [\![r]\!]s$
$\mathsf{Tfun}(\rho, m)$	$\forall p \in [\![m]\!]s.\ \exists! n \in \mathbf{Z}.\ (p,n) \in [\![\rho]\!]s$
$\mathsf{iso}(r, m_1, m_2)$	$\forall p_1 \in M_1.\ \exists! p_2 \in M_2.\ (p_1,p_2) \in \varphi \wedge$ $\forall p_2 \in M_2.\ \exists! p_1 \in M_1.\ (p_1,p_2) \in \varphi \wedge$ $\forall (p_1,p_2) \in \varphi.\ p_1 \in M_1 \wedge p_2 \in M_2,$ where $M_1 = [\![m_1]\!]s, M_2 = [\![m_2]\!]s, \varphi = [\![r]\!]s$
$\mathsf{Reachable}(\rho, \rho', m, e)$	$s,h \models \mathsf{Tfun}(\rho, m) \wedge \mathsf{Tfun}(\rho', m)$ and $\exists h'.\ ([\![e]\!]s, h')$ is exactly reachable, and $\mathit{pairheap}([\![m]\!]s, [\![\rho]\!]s, [\![\rho']\!]s, h')$

Fig. 2. Semantics of Assertions

—*We call an assertion* A intuitionistic *if, for all stacks* s *and heaps* h, h',

$$(s,h) \models A \text{ and } h \subseteq h' \text{ imply } (s,h') \models A.$$

Here, $\subseteq$ *is just set-theoretic inclusion of graphs.*

REMARK 3.7. *Special rules apply for certain of the newly introduced classes of assertions. Here are some of these rules.*

—*When* A *is a pure assertion, the rule*

$$A \wedge (B * C) \longleftrightarrow (A \wedge B) * C \tag{9}$$

is valid for any assertions B *and* C.

—*Pure assertions are intuitionistic.*

—*Syntactically, an assertion is pure, if it does not contain any occurrences of* emp, $\forall_*$, $\mapsto$, *or the shorthand notation* $\hookrightarrow$.

3.5 Some Useful Rules

A number of axiom schemata are used in proofs later. We believe they could be part of a small theory of finite sets and relations that could be used in proofs of other programs where the goal is to establish an isomorphism between data structures before and after execution. The schemata most relevant to separation logic are listed below, and more can be found in the survey paper [Reynolds 2002]. Beside these, there are a number of obvious rules regarding sets and relations which can easily be verified semantically. For completeness, these are listed in Appendix B.
Rules for $\forall_*$:

$$(\forall_* x \in m.\ A)\ \wedge\ m = m' \to \forall_* x \in m'.\ A \tag{10}$$

$$m = \emptyset \to ((\forall_* x \in m.\ A) \longleftrightarrow \mathsf{emp}) \tag{11}$$

$$\begin{array}{l}(\forall_* x \in m.\ x \mapsto -\ \wedge\ A) \wedge y \in m \to \\ \qquad (\forall_* x \in m.\ x \mapsto -\ \wedge\ A) \wedge (y \hookrightarrow -)\end{array} \tag{12}$$

When m and m' are disjoint,

$$(\forall_* x \in m.\ A) * (\forall_* x \in m'.\ A) \longleftrightarrow (\forall_* x \in m \cup m'.\ A) \tag{13}$$

As a special case, we get

$$\begin{array}{l}e \in m \to \\ ((\forall_* x \in m.\ A) \longleftrightarrow ((\forall_* x \in (m \setminus \{e\}).\ A) * A[e/x])).\end{array} \tag{14}$$

General / Structural rules: When $x \notin \mathrm{FV}(e)$,

$$\begin{array}{l}(e \hookrightarrow e') \wedge ((\exists x.\ e \mapsto x \wedge A) * B) \longleftrightarrow \\ (e \mapsto e' \wedge A[e'/x])) * B\end{array} \tag{15}$$

If B is pure and B' is intuitionistic,

$$\frac{A \wedge B \to B'}{(A * A') \wedge B \to B'}\ . \tag{16}$$

We also use the commutativity and associativity of the separating conjunction (the same rules for the traditional conjunction are also used, but we refrain from stating them here):

$$A * B \longleftrightarrow B * A \qquad \text{and} \qquad A * (B * C) \longleftrightarrow (A * B) * C \tag{17}$$

THEOREM 3.8. *The rules (10) - (17), and the rules (34) - (83) from Appendix B are all valid.*

We shall use the commutativity and associativity rules implicitly in proofs, and we shall often just say "by purity", when we apply the distributive law (9) for pure assertions from Remark 3.7.

The following lemma is useful when reasoning about assertions involving $\forall_*$.

LEMMA 3.9. *Suppose $A, B, s,$ and h are such that $(s,h) \models \forall_* x \in m.\ A$, and $\forall x'.\ x' \in m \wedge A[x'/x] \to B[x'/x]$ is valid. Then $(s,h) \models \forall_* x \in m.\ B$.*

PROOF. We do a case analysis on the cardinality of $[\![m]\!]s$. If $[\![m]\!]s = \emptyset$, then

$$(s,h) \models \forall_* x \in m.\ A \iff (s,h) \models \mathsf{emp} \iff (s,h) \models \forall_* x \in m.\ B$$

If $[\![m]\!]s = \{p_1, \ldots, p_k\}$, we have

$$(s,h) \models A[p_1/x] * \cdots * A[p_k/x],$$

so there is a partition $h = h_1 \cdot \ldots \cdot h_k$ such that for $i = 1 \ldots k$, $s, h_i \models A[p_i/x]$. Since we have $s, h_i \models A[p_i/x] \wedge p_i \in m$, we get $(s, h_i) \models B[p_i/x]$, and this means that

$$(s,h) \models \forall_* x \in m.\ B,$$

as desired. □

This shows that we can exploit the information about $x' \in m$ to do "implication under $\forall_*$". Another useful rule comes from the following lemma for which we omit the proof.

LEMMA 3.10. *If D is a pure assertion, and if $D \wedge A \to A'$ and $D \wedge B \to B'$ are valid, then $D \wedge (A * B) \to D \wedge (A' * B')$ is valid.*

By induction, this means that $D \wedge (A_1 * \cdots * A_k) \to (A'_1 * \cdots * A'_k)$ can be inferred from $D \wedge A_1 \to A'_1$, and $\cdots$,and $D \wedge A_k \to A'_k$.

Note that Lemma 3.10 is not valid if D is not pure — as a counterexample, set A to $4 \mapsto 2$, B to $8 \mapsto 3$, A' to $4 \mapsto 10$, B' to $8 \mapsto 17$, and D to $(4 \mapsto 2 * 8 \mapsto 3)$.

As an example of a rule that can be derived from the rules above, we get the following from (15) and purity. Here, A must be a pure assertion; otherwise the second step below is not valid.

$$\begin{array}{l} (e \hookrightarrow e') \wedge ((\exists x.\ (e \mapsto x \wedge A)) * B) \\ \Downarrow \\ (e \mapsto e' \wedge A[e'/x]) * B \\ \Downarrow \\ ((e \mapsto e') * B) \wedge A[e'/x] \\ \Downarrow \\ A[e'/x] \end{array} \tag{18}$$

In addition to the rules above, we have the standard rules of classical logic. We will sometimes implicitly substitute equals for equals; for example, we will infer $e \in m_2$ from $e \in m_1 \wedge m_1 = m_2$. Also, the most basic arithmetic will be performed implicitly, so we will for example infer $x - 8 \leq y$ from $x \leq y$.

4. PROGRAMMING LANGUAGE

In this section we first define the syntax and semantics of the programming language used for implementing our garbage collector. Next, we use the assertion language from Sec. 3.4 to give a program logic in the style of Hoare for the language. This enables us to specify and prove correct the collector in Sec. 5 and 6.

4.1 Syntax and Semantics

DEFINITION 4.1. *The syntax of the implementation language is given by the following grammar:*

$$C ::= \mathbf{skip} \mid x^{\mathsf{int}} := e \mid x^{\mathsf{fs}} := m \mid x^{\mathsf{frp}} := r \mid x^{\mathsf{int}} := [e] \mid [e] := e \\ \mid C; C \mid \mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{od} \mid \mathbf{if}\ e\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi}$$

Note that there are no constructs for allocating or deallocating locations on the heap. In our implementation and specification of Cheney's algorithm we simply assume that the domain of the heap contains the necessary locations. It would be straightforward to add these features to the language, implement a version of the algorithm that allocates the necessary space before collection and disposes garbage afterwards, and modify our proof of its correctness. We discuss this further in Section 5.

The operational semantics is given by a (small-step) relation $\rightsquigarrow$ on *configurations* (ranged over by K). Configurations are either of the form s, h (these are called *terminal*) or of the form C, s, h (these are called *non-terminal*).

DEFINITION 4.2. *The relation $\rightsquigarrow$ on configurations is defined by the following inference rules:*

$$\frac{}{\mathbf{skip}, s, h \rightsquigarrow s, h} \qquad \frac{[\![e]\!]s = n}{x^{\mathsf{int}} := e, s, h \rightsquigarrow s[x \mapsto n], h}$$

$$\frac{[\![m]\!]s = M}{x^{\mathsf{fs}} := m, s, h \rightsquigarrow s[x \mapsto M], h} \qquad \frac{[\![r]\!]s = \varphi}{x^{\mathsf{frp}} := r, s, h \rightsquigarrow s[x \mapsto \varphi], h}$$

$$\frac{[\![e]\!]s = p \quad p \in \mathrm{dom}(h) \quad h(p) = n}{x^{\mathsf{int}} := [e], s, h \rightsquigarrow s[x \mapsto n], h} \qquad \frac{[\![e_1]\!]s = p \quad [\![e_2]\!]s = n \quad p \in \mathrm{dom}(h)}{[e_1] := e_2, s, h \rightsquigarrow s, h[p \mapsto n]}$$

$$\frac{C_1, s, h \rightsquigarrow C', s', h'}{C_1; C_2, s, h \rightsquigarrow C'; C_2, s', h'} \qquad \frac{C_1, s, h \rightsquigarrow s', h'}{C_1; C_2, s, h \rightsquigarrow C_2, s', h'}$$

$$\frac{[\![e]\!]s = 0}{\mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{od}, s, h \rightsquigarrow s, h} \qquad \frac{[\![e]\!]s \neq 0 \quad C; \mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{od}, s, h \rightsquigarrow K}{\mathbf{while}\ e\ \mathbf{do}\ C\ \mathbf{od}, s, h \rightsquigarrow K}$$

$$\frac{[\![e]\!]s = 0 \quad C_2, s, h \rightsquigarrow K}{\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s, h \rightsquigarrow K} \qquad \frac{[\![e]\!]s \neq 0 \quad C_1, s, h \rightsquigarrow K}{\mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}, s, h \rightsquigarrow K}$$

The semantics is easily seen to be deterministic. We introduce the following terminology.

DEFINITION 4.3. *We say that*

—C, s, h *is* stuck *if there is no configuration K such that $C, s, h \rightsquigarrow K$.*

—C, s, h goes wrong *if there is a non-terminal configuration K with $C, s, h \rightsquigarrow^* K$ and K is stuck.*

—C, s, h terminates normally *if there is a terminal configuration s', h' such that $C, s, h \rightsquigarrow^* s', h'$.*

Other published definitions of the programming language used in separation logic use a special configuration called "abort" or "fault" instead of the concept

of "stuck"; we are able to avoid this complication because we have restricted the programming language to a deterministic sublanguage.

As is standard, we define $\mathrm{Mod}(C)$ to be the set of variables that are modified by the command C, i.e., those that occur on the left hand side of the forms $x^{\delta} := v$ and $x^{\mathsf{int}} := [e]$ (but *not* $[x] := e$). The set $\mathrm{FV}(C)$ for a command is just the set of variables that occur in C.

4.2 Partial Correctness Specifications and Program Logic

DEFINITION 4.4. *Let A and B be assertions, and let C be a command. The* partial correctness specification *(pcs) $\{A\}\ C\ \{B\}$ is said to hold if, for all states (s,h) with $\mathrm{FV}(A,C,B) \subseteq \mathrm{dom}(s)$, $(s,h) \models A$ implies*

—C, s, h does not go wrong, and
—if $C, s, h \rightsquigarrow^* s', h'$, then $(s', h') \models B$.

We refer to A and B as the pre- *and* postcondition *of the specification, respectively.*

We present a set of proof rules that are sound with respect to Def. 4.4. First, we give rules regarding the different constructs in the programming language and then we give some structural rules.

Rule for **skip**: $\{A\}\ \mathbf{skip}\ \{A\}$

Rules for assignment

$$\begin{array}{l} \{B[e/x]\}\ x^{\mathsf{int}} := e\ \{B\} \\ \{B[m/x]\}\ x^{\mathsf{fs}} := m\ \{B\} \\ \{B[r/x]\}\ x^{\mathsf{frp}} := r\ \{B\} \end{array} \tag{19}$$

Rules for heap lookup. When $x \notin \mathrm{FV}(e', A)$ and $y \notin \mathrm{FV}(e)$,

$$\{(\exists y.\ e \mapsto y \wedge A) \wedge x = e'\}\ x := [e]\ \{e[e'/x] \mapsto x \wedge A[x/y]\} \tag{20}$$

When $x \notin \mathrm{FV}(e, A)$ and $y \notin \mathrm{FV}(e)$,

$$\{\exists y.\ e \mapsto y\ \wedge\ A\}\ x := [e]\ \{e \mapsto x \wedge A[x/y]\} \tag{21}$$

Rule for heap update : $$\{e_1 \mapsto -\}\ [e_1] := e_2\ \{e_1 \mapsto e_2\} \tag{22}$$

Rule for sequencing

$$\frac{\{A\}\ C_1\ \{B'\} \quad \{B'\}\ C_2\ \{B\}}{\{A\}\ C_1; C_2\ \{B\}}$$

Rule for conditionals

$$\frac{\{A\ \wedge\ b\}\ C_1\ \{B\} \quad \{A\ \wedge\ \neg b\}\ C_2\ \{B\}}{\{A\}\ \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ \{B\}}$$

Rule for **while** loops

$$\frac{\{A\ \wedge\ b\}\ C\ \{A\}}{\{A\}\ \mathbf{while}\ b\ \mathbf{do}\ C\ \mathbf{od}\ \{A\ \wedge\ \neg b\}}$$

Structural Rules:

Rule of consequence

$$\frac{A \Rightarrow A' \quad \{A'\}\ C\ \{B'\} \quad B' \Rightarrow B}{\{A\}\ C\ \{B\}}$$

Rule of Conjunction

$$\frac{\{A\}\ C\ \{A'\} \quad \{B\}\ C\ \{B'\}}{\{A \wedge B\}\ C\ \{A' \wedge B'\}}$$

$\forall$ introduction rule

$$\frac{\{A\}\ C\ \{B\}}{\{\forall x.\ A\}\ C\ \{\forall x.\ B\}}\ x \notin \mathrm{Modifies}(C)$$

The Frame Rule

$$\frac{\{A\}\ C\ \{B\}}{\{A * A'\}\ C\ \{B * A'\}}\ \mathrm{Modifies}(C) \cap \mathrm{FV}(A') = \emptyset$$

The frame rule makes local reasoning possible: suppose the assertion $A * A'$ describes a state in which we are to execute C, but that the "footprint" of C, i.e., those locations read or written by C, is described by A and B. Then from a local specification $\{A\}\ C\ \{B\}$ for C, involving only this footprint, one can infer a global specification $\{A * A'\}\ C\ \{B * A'\}$, which also mentions locations not in the footprint of C. It is simpler to state and reason about local specifications, and the frame rule says that it is adequate to do so. We shall see several applications of the frame rule later.

We have the expected soundness result.

THEOREM 4.5. *If a specification* $\{A\}\ C\ \{B\}$ *is derivable by the rules in this section, then* $\{A\}\ C\ \{B\}$ *holds.*

The proof of this theorem follows from proofs in earlier literature on separation logic [Yang 2001; Reynolds 2002].

5. THE GARBAGE COLLECTION ALGORITHM

We implement and reason about *Cheney's Algorithm* [Cheney 1970]. The implementation of the algorithm and the associated memory allocator is given in Appendix A. It assumes two disjoint contiguous "semi-heaps" which have as domains the intervals $Itv(\mathsf{startOld}, \mathsf{endOld})$ and $Itv(\mathsf{startNew}, \mathsf{endNew})$ of equal size. We use the abbreviations

$$\mathsf{OLD} \equiv Itv(\mathsf{startOld}, \mathsf{endOld}) \text{ and } \mathsf{NEW} \equiv Itv(\mathsf{startNew}, \mathsf{endNew})$$

for these two intervals in the rest of the paper. The memory allocator attempts to allocate a **cons**-cell in OLD; if there is no space available in OLD, the garbage collector copies all cells in OLD reachable from root into NEW, and then the allocation resumes in NEW. The garbage collector GC^* is delimited by comments in the code in Appendix A. Notice that the algorithm is aware of the locations of the reachable cells only. In the spirit of local reasoning, our specification will therefore only involve the reachable pointers in OLD, called RCH, and not the remaining (unreachable) part of OLD. The implementation starts by initializing the variables

offset, scan, free, and maxFree, according to which of the semi-heaps mentioned above that contains the data to be copied. We consider the case where offset is initialized to startNew.

The set $\{\varphi, \mathsf{FWD}, \mathsf{UFWD}\}$ is an *auxiliary variable set* for the implementation, in the sense of [Owicki and Gries 1976; Reynolds 1981]. Thus, assignments to these variables are not necessary for the program to work; rather they ease the job of proving properties about the program – hence these assignments need not be executed. We have marked assignments to these variables with vertical bars in the margin of the code. We could have chosen to existentially quantify these variables and omit them from the program, but the reasoning becomes clearer when the program modifies the auxiliary variables explicitly.

Had we chosen to include allocation and disposal of blocks of memory in our programming language, a different implementation might have allocated a new semi-heap initially, then performed the garbage collection, and finally disposed the old semi-heap after collection. We stick to the current implementation, since this other implementation would not add any features of significant interest to our proof.

In the rest of this section, we first present a semantic condition in Sec. 5.1 which ensures that our program is a correct garbage collector. This condition is tightly connected to Cheney's algorithm (and our implementation of it), inasmuch as it explicitly mentions the input and output variables such as offset and scan, and even auxiliary variables such as φ, head, and tail. We then present the precondition for the algorithm and the invariant for the **while** loop in the implementation, both formulated in the assertion language from Sec. 3.4. Before we formally prove that our implementation meets the corresponding specification in Sec. 6, we show in Sec. 5.4 that the specification entails that our implementation meets the requirement from Sec. 5.1.

5.1 Semantic Specification of a Copying Garbage Collector

The semantic definition of a garbage collector presented here is quite specific to our implementation, since it mentions the variables we use. It can be generalized by quantifying over these, but it is clearer to use the same variables as in the implementation. Thus, we have the following definition:

DEFINITION 5.1. *We call a command GC^* a* correct copying garbage collector *provided that if*

(a) $[\![\mathsf{head}]\!]s, [\![\mathsf{tail}]\!]s$ *are total functions on* $[\![\mathsf{RCH}]\!]s$*, and* $[\![\mathsf{offset}]\!]s, [\![\mathsf{maxFree}]\!]s$ *are pointers,*

(b) the heaps $h_{rch}, h_{new}, h_{extra}$ *are disjoint,*

(c) the rooted heap $([\![\mathsf{root}]\!], h_{rch})$ *is exactly reachable,*

(d) $pairheap([\![\mathsf{RCH}]\!]s, [\![\mathsf{head}]\!]s, [\![\mathsf{tail}]\!]s, h_{rch})$*,*

(e) $\mathrm{dom}(h_{new}) = [\![\mathsf{NEW}]\!]s$*,*

(f) $\#[\![\mathsf{RCH}]\!]s \leq \#[\![\mathsf{NEW}]\!]s$*, and*

(g) $GC^*, s, h_{rch} \cdot h_{new} \cdot h_{extra} \rightsquigarrow^* s', h'$*,*

then there exists a disjoint split $h' = h'_{rch} \cdot h'_{fin} \cdot h'_{free}$ *and a variable* FIN *such that*

(a') the heaps $h'_{rch}, h'_{fin}, h'_{free}$*, and* h_{extra} *are pairwise disjoint,*

(b') $\mathrm{dom}(h'_{rch}) = \mathrm{dom}(h_{rch})$,
(c') $\mathrm{dom}(h'_{fin}) = [\![\mathsf{FIN}]\!]s'$,
(d') $\mathrm{dom}(h'_{fin} \cdot h'_{free}) = \mathrm{dom}(h_{new})$, *and*
(e') $[\![\varphi]\!]s'$ *is a heap morphism from* $([\![\mathsf{root}]\!]s, h_{rch})$ *to* $([\![\mathsf{offset}]\!]s', h'_{fin})$ *that is an isomorphism of functions.*

Clearly, if $([\![\mathsf{root}]\!]s, h_{rch} \cdot h_{new} \cdot h_{extra})$ contains an exactly reachable subheap, then that must be $([\![\mathsf{root}]\!]s, h_{rch})$, and $([\![\mathsf{offset}]\!]s', h'_{fin})$ will be an isomorphic exactly reachable subheap, due to Lemma 3.1. Thus, this condition clearly ensures that GC^* works as we would expect of a garbage collector on a heap with no dangling reachable locations, *i.e.*, it makes $([\![\mathsf{offset}]\!]s', h')$ a garbage collected version of $([\![\mathsf{root}]\!]s, h_{rch} \cdot h_{new} \cdot h_{extra})$ in the sense of Definition 3.2 (but it does not say what GC^* does if this is not the case). Furthermore, the definition specifies properties specific to a *copying* garbage collector; for example, it assumes that there is enough new space in the heap to make a new copy of the reachable data.

In the rest of this section, we first present a specification for our implementation and then show that this specification is strong enough to infer that our implementation of Cheney's algorithm meets the requirements in Def. 5.1.

5.2 The Precondition

Before execution of GC^*, we assume that an assertion $\mathsf{InitAss}$ holds. $\mathsf{InitAss}$ can be split into two parts:

$$\mathsf{InitAss} \equiv \mathsf{I}_c \wedge \mathsf{I}_h,$$

where I_c is pure and I_h describes the heap unambiguously. These assertions are given by

$$\begin{aligned}\mathsf{I}_c \;\equiv\; & \mathsf{RCH} \perp \mathsf{NEW} \wedge \#\mathsf{RCH} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{RCH} \wedge \\ & \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{RCH}) \wedge \mathsf{Tfun}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{RCH}) \wedge \\ & \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{RCH}, \mathsf{root}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{maxFree})\end{aligned}$$

and

$$\begin{aligned}\mathsf{I}_h \;\equiv\; & (\forall_* x \in \mathsf{RCH}.\ ((\exists y.\ (x, y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x, y') \in \mathsf{tail} \wedge x + 4 \mapsto y'))) * \\ & (\forall_* x \in \mathsf{NEW}. x \mapsto -, -).\end{aligned}$$

We use the convention that variables in upper case sans serif (like OLD) always have type fs. The variables head and tail are of type fri, and the rest of the variables are of type int.

Note that I_c is "constant" in the sense that it trivially remains true throughout execution of GC^*, since it is pure and only contains variables that are not modified. This means that it is also part of the invariant of the **while** loop.

We describe part of these assertions in detail. For I_c, note in particular the conjunct

$$\mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{RCH}, \mathsf{root}) \tag{23}$$

expressing that $\mathsf{head}, \mathsf{tail}, \mathsf{RCH}$, and root describe an exactly reachable subheap. We shall see that RCH is the set of intially reachable pointers from root. Since the variables head, tail, root, and RCH are not modified by the algorithm, this holds at any step of execution. The rest of I_c simply records basic facts about the

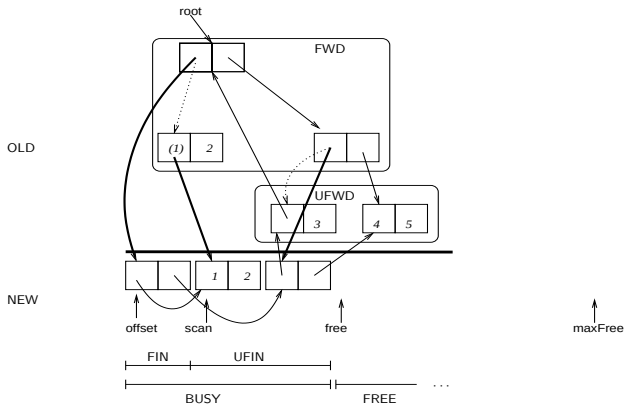

Fig. 3. A state of execution

relationship between the various sets and pointers. To understand I_h, suppose x denotes a pointer and consider the assertion

$$(\exists y.\ (x, y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x, y') \in \mathsf{tail} \wedge x + 4 \mapsto y').$$

It asserts that there are values v, v' that are related to (the denotation of) x by the relations head and tail, and at the same time, x and $x + 4$ point to these values in the heap. In this way, head and tail "record" the contents of the cell pointed to by x. By the definition of $\forall_*$, the assertion

$$((\forall_* x \in \mathsf{RCH}.\ ((\exists y.\ (x, y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x, y') \in \mathsf{tail} \wedge x + 4 \mapsto y')))$$

therefore asserts that head and tail record the contents of all cells pointed to be pointers in RCH, (and those pointers are in the domain of the heap). It is easy to see from (23) and this assertion that RCH is the domain of an exactly reachable subheap of the initial heap. Also, I_h asserts that the set NEW is in the domain of the heap, so we can safely place copies of cells from RCH here.

5.3 The Invariant

To exhibit an invariant of the **while**-loop, we consider Fig. 3, which is a snapshot of a state during execution. Only the reachable cells in OLD (the part of the heap above the bold horizontal line) are shown. Three of the cells in RCH have been modified at this stage: their first fields have been updated with *forwarding pointers*; these appear bolder in the figure. The original contents of these first fields are indicated with dotted lines and parenthesized numbers. The pointers in RCH naturally divide into two sets:

—UFWD: The pointers in RCH that point to cells not yet modified by the algorithm.

—FWD: The pointers in RCH that point to cells that have their first fields overwritten with a pointer in NEW.

The algorithm proceeds by traversing all the cells in between the scan and free pointers, that is, scan always points to the next cell to be traversed. A cell is traversed by *scanning* its two fields. If the field being scanned contains a non-pointer,

the traversal simply proceeds to the next field or cell; if the field being scanned contains a pointer p in UFWD, the cell pointed to by p is copied and a forwarding pointer is placed in the original field; if the field being scanned contains a pointer in FWD, then the cell pointed to has already been copied and we simply update the scanned field to point to the copy. We use the auxiliary variables φ (which is of type frp), FWD, and UFWD to keep track of the forwarding pointers, and to record the reachable cells that have been already copied into NEW. When a cell is copied from RCH to NEW, the corresponding pointer is moved from UFWD to FWD, and φ is updated. To simplify certain aspects of the proof, each execution of the body of the **while** command uses the subprograms ScanCar and ScanCdr to scan the two fields of a cell that lie at the locations scan and scan + 4. Thus the actual value of scan always addresses the first field of a cell, and is therefore a pointer.

The pointers in NEW divide into the following three sets:

—FIN (which is an abbreviation of $Itv(\mathsf{offset}, \mathsf{scan})$): The pointers in NEW that have been scanned. These are not modified further by the algorithm.

—UFIN (which is an abbreviation of $Itv(\mathsf{scan}, \mathsf{free})$): The pointers in NEW that have not been scanned. These point to the original contents of cells pointed to by pointers in RCH.

—FREE (which is an abbreviation of $Itv(\mathsf{free}, \mathsf{maxFree})$): The pointers in NEW that are available for allocation.

The five sets are illustrated in Fig. 3. Note that FIN, UFIN and FREE are intervals, whereas this is not the case for FWD and UFWD in general. Also, observe that φ is a one-to-one correspondence between the pointers in FWD and those in $\mathsf{BUSY} \equiv \mathsf{FIN} \cup \mathsf{UFIN} = Itv(\mathsf{offset}, \mathsf{free})$. This bijection will turn out to be the heap morphism we are looking for.

The invariant of the algorithm has a pure and an impure part; the latter describes the heap. The pure part is

$$\begin{aligned} \mathsf{I}_{pure} \;\equiv\; & \mathsf{I}_c \wedge (\mathsf{root}, \mathsf{offset}) \in \varphi \wedge (\mathsf{root} \in \mathsf{FWD}) \wedge \\ & \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge (\mathsf{RCH} = \mathsf{FWD} \cup \mathsf{UFWD}) \wedge \\ & \mathsf{scan} \leq \mathsf{free} \wedge \mathsf{offset} \leq \mathsf{scan} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \end{aligned}$$

Note in particular the conjunct $\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY})$ expressing that φ is a bijection. The rest of I_{pure} simply records basic facts about the relationship between the various sets and pointers.

We now describe the impure part of the invariant; we use the partitioning of pointers into sets from before.

The cells pointed to by pointers in UFWD have not been modified by the algorithm; hence they are described by head and tail in the same way as the pointers in RCH in InitAss. We thus define

$$\begin{aligned} & \mathsf{A}_{\mathsf{UFWD}} \equiv \forall_* x \in \mathsf{UFWD}. \\ & ((\exists y.\ (x, y) \in \mathsf{head} \wedge x \mapsto y) * \ (\exists y'.\ (x, y') \in \mathsf{tail} \wedge x + 4 \mapsto y')). \end{aligned}$$

Each of the cells pointed to by a pointer in FWD has a forwarding pointer in its first field. Recalling that φ records the forwarding pointers, we define

$$\mathsf{A}_{\mathsf{FWD}} \equiv \forall_* x \in \mathsf{FWD}.\ (\exists y.\ (x, y) \in \varphi \wedge x \mapsto y, -).$$

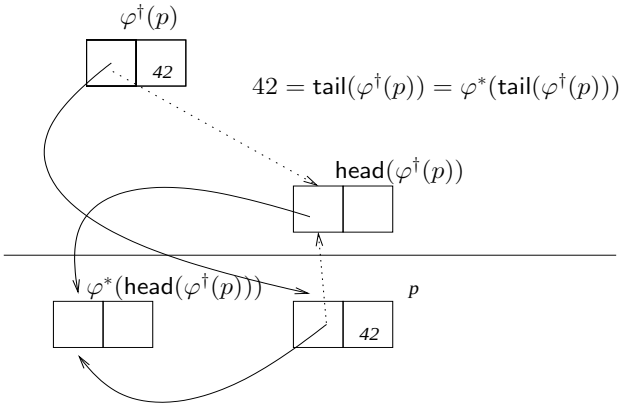

Fig. 4. The situation for a pointer p in FIN

A cell pointed to by a pointer in UFIN contains the original contents of a cell pointed to by a pointer in FWD. The latter pointer is recorded by the inverse of φ, and hence we define

$$\begin{aligned}\mathsf{A}_{\mathsf{UFIN}} \equiv \forall_* x \in \mathsf{UFIN}.\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\ (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')).\end{aligned}$$

The cells in FIN have been scanned. The case-distinction between pointers and non-pointers during scanning is captured by the operator $\odot$. We define

$$\begin{aligned}&\mathsf{A}_{\mathsf{FIN}} \equiv \\ &\forall_* x \in \mathsf{FIN}.\ ((\exists y.\ (x,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)\ \wedge x \mapsto y) * \\ &\qquad (\exists y'.\ (x,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x+4 \mapsto y')).\end{aligned}$$

To understand $\mathsf{A}_{\mathsf{UFIN}}$ and $\mathsf{A}_{\mathsf{FIN}}$, it is helpful to consider Fig. 4, in which we use a functional notation for the functional relations head, tail, and φ. The pointer $p \in \mathsf{FIN}$ is the address of the rightmost bottom cell. Before p was scanned, it held the original contents of a cell pointed to by a pointer $q \in \mathsf{RCH}$. After that cell was copied, it had its first field overwritten with the forwarding pointer p; this is recorded by φ, hence $(q,p) \in \varphi$. The original contents of the cell pointed to by q is recorded by head and tail, so letting q' denote the address of the rightmost upper cell in Fig. 4, we have $(q,q') \in \mathsf{head}$, hence $(p,q') \in \mathsf{head} \circ \varphi^\dagger$. Before the field pointed to by p was scanned, it had q' in its first field. Now, by scanning the first field in the cell pointed to by p, we copy the cell pointed to by q' (if necessary), and update the field we are scanning to point to the address of the copy of that cell. Denoting the address of the copy by q'', we then have $(q',q'') \in \varphi$, by the definition of φ, and hence, $(p,q'') \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)$.

For the pointers in FREE, we need only know that they are in the domain of the heap, to allow us to safely copy cells into FREE. We therefore define

$$\mathsf{A}_{\mathsf{FREE}} \equiv \forall_* x \in \mathsf{FREE}.\ x \mapsto -,-.$$

In summary, the invariant of the algorithm is

$$\mathsf{I} \equiv \mathsf{I}_{pure} \wedge (\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FREE}}),$$

	Unchanging	Changing
Abbreviations	$\mathsf{NEW} \equiv Itv(\mathsf{startNew}, \mathsf{endNew})$	$\mathsf{FIN} \equiv Itv(\mathsf{offset}, \mathsf{scan})$ $\mathsf{UFIN} \equiv Itv(\mathsf{scan}, \mathsf{free})$ $\mathsf{FREE} \equiv Itv(\mathsf{free}, \mathsf{maxFree})$ $\mathsf{BUSY} \equiv Itv(\mathsf{offset}, \mathsf{free})$
Auxiliary	RCH head tail	FWD UFWD φ
Program	startOld endOld startNew endNew	root scan free offset maxFree

Table I. Variables involved in the proof of the garbage collector.

and hence, we aim to prove the specification

$$\{\mathsf{InitAss}\}\ GC^*\ \{\mathsf{I} \wedge \mathsf{scan} = \mathsf{free}\}. \tag{24}$$

Notice that the sets $\mathsf{UFWD}, \mathsf{FWD}$, etc. change throughout execution of the algorithm. Therefore, each of the corresponding assertions $\mathsf{A}_{\mathsf{UFWD}}, \mathsf{A}_{\mathsf{FWD}}, \ldots$ describe *dynamic* portions of the heap.

We prove that the specification (24) holds in Section 6, but prior to that, we prove that this implies correctness of the algorithm in Section 5.4.

For the assertions $\mathsf{A}_{\_}$ defined above, we will abuse notation slightly. It will sometimes be practical to consider an iterated separating conjunction over over one of the above mentioned sets, except for one element, which needs to be considered separately. Therefore, for example, we will let $\mathsf{A}_{\mathsf{FWD}-x}$ denote the following assertion (compare with the definition of $\mathsf{A}_{\mathsf{FWD}}$)

$$\forall_* z \in (\mathsf{FWD} \setminus \{x\}).\ (\exists y.\ (z, y) \in \varphi \wedge z \mapsto y, -).$$

To get a better overview, we list the different variables (along with what they abbreviate) that are involved in the program and the proof in Table I. Recall that auxiliary variables are not needed for the program to work, but are included to ease the proof of the program. Auxiliary variables that are not modified by programs are called "ghost variables" or "logical variables" in the literature [Reynolds 1981; Gries and Levin 1980].

5.4 Sufficiency of the Specification

Before we formally show that our implementation meets the specification (24), we show that the specification entails that the implementation meets the requirements from Def. 5.1.

First, since heaps are finite, there is a variable-free assertion A_{h_0} which is satisfied only by h_0, for every heap h_0. For instance, if h is the two-element heap $[0 : 4 \mid 8 : 67]$, then A_h might be $0 \mapsto 4 * 8 \mapsto 67$. For any heap h_{extra}, the following specification therefore follows from (24) and the frame rule.

$$\{\mathsf{InitAss} * A_{h_{extra}}\}\ GC^*\ \{(\mathsf{I} \wedge \mathsf{scan} = \mathsf{free}) * A_{h_{extra}}\}. \tag{25}$$

Thus, to show that our specification implies that our implementation meets the requirements of Def. 5.1, it suffices to show that if the conditions (a) to (f) in Def. 5.1 are met, then

$$(s, h_{rch} \cdot h_{new} \cdot h_{extra}) \models \mathsf{InitAss} * A_{h_{extra}},$$

and if $GC^*, s, h_{rch} \cdot h_{new} \cdot h_{extra} \rightsquigarrow^* s', h'$ and

$$(s', h') \models \mathsf{I} * A_{h_{extra}} \wedge \mathsf{scan} = \mathsf{free},$$

then conditions (a') to (e') from Def. 5.1 are met. The first of these claims is relatively straightforward, so we focus on the latter. For brevity, we first set $h = h_{rch} \cdot h_{new} \cdot h_{extra}$.

Thus, assume $GC^*, s, h \rightsquigarrow^* s', h'$ for a state (s, h) in which the conditions (a) to (f) hold, and that the specification (25) holds. We then need to show that the conditions (a') to (e') from Def. 5.1 hold. The desired split of h' is the obvious one, so we only focus on showing the condition (e').

In our proof, we occasionally use names of variables instead of their denotations. Also, since $\mathsf{head}, \mathsf{tail}$, and φ denote functional relations (this clearly follows from I), we use functional notation for them, taking care to ensure definedness. For example, we write "there exists a q such that $\varphi(p) = q$" instead of "there exists a q such that $(p, q) \in \varphi$".

First note that the variables $\mathsf{RCH}, \mathsf{head}$, and tail are not modified by GC^*, so the values of these before and after execution of the algorithm are equal (for example, $[\![\mathsf{RCH}]\!]s = [\![\mathsf{RCH}]\!]s'$). The following lemma is important to establish that (the denotation of) φ is indeed a bijection that has the right domain and codomain.

LEMMA 5.2. *Under the assumptions above,*

$$[\![\mathsf{RCH}]\!]s = [\![\mathsf{RCH}]\!]s' = [\![\mathsf{FWD}]\!]s'.$$

PROOF. The invariant I contains the assertion $\mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{RCH}, \mathsf{root})$, and therefore there is a heap h_0 such that $([\![\mathsf{root}]\!]s', h_0)$ is exactly reachable and $pairheap(\mathsf{RCH}, \mathsf{head}, \mathsf{tail}, h_0)$. As mentioned in Section 3.2, h_0 is determined by the values of $\mathsf{RCH}, \mathsf{head}$, and tail. These variables are not altered by the algorithm, so h_0 is the same heap that is determined by $\mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{RCH}, \mathsf{root})$ in the state (s, h), which implies $h_0 = h_{rch}$. Hence $([\![\mathsf{root}]\!]s', h_{rch})$ is exactly reachable. Let p be a pointer in $[\![\mathsf{RCH}]\!]s' = \mathrm{pdom}(h_{rch})$. This implies $p \in rp([\![\mathsf{root}]\!]s', h_{rch})$, and we show by induction on n, that $p \in rp_n([\![\mathsf{root}]\!]s', h_{rch})$ implies $p \in [\![\mathsf{FWD}]\!]s'$.

If $n = 0$, then $p = \mathsf{root}$, and $\mathsf{root} \in \mathsf{FWD}$ is part of I.

Assume, for the induction step, that $rp_n([\![\mathsf{root}]\!]s', h_{rch}) \subseteq [\![\mathsf{FWD}]\!]s'$, and let $p \in rp_{n+1}([\![\mathsf{root}]\!]s', h_{rch})$. We need to show that $p \in [\![\mathsf{FWD}]\!]s'$. By assumption, there is $\ell \in rl_n([\![\mathsf{root}]\!]s', h_{rch})$ with $h_{rch}(\ell) = p$. By definition, $\ell = p_0 + k$ for some $p_0 \in rp_n([\![\mathsf{root}]\!]s', h_{rch})$ and $k \in \{0, 4\}$. By the induction hypothesis, $p_0 \in s'(\mathsf{FWD})$. We argue by cases on whether k is 0 or 4.

If $k = 0$, then $h_{rch}(p_0) = p$. Because we have $pairheap(\mathsf{RCH}, \mathsf{head}, \mathsf{tail}, h_{rch})$, this implies $\mathsf{head}(p_0) = p$. Since p_0 is in FWD, I implies that there is a q_0 such that $(p_0, q_0) \in \varphi$ (*i.e.*, $p_0 = \varphi^{-1}(q_0)$), and this means that q_0 is in FIN. The iterated separating conjunction over the set FIN that is part of I implies that there is a q_1 such that q_0 points to q_1, and

$$\begin{aligned} q_1 &= \varphi^*(\mathsf{head} \circ \varphi^{-1}(q_0)) \\ &= \varphi^*(\mathsf{head}(p_0)) \\ &= \varphi^*(p) \\ &= \varphi(p), \end{aligned}$$

since p is a pointer. This means that p is in the domain of φ, and thus $p \in \mathsf{FWD}$.

If $k = 4$, we use the same argument, but with tail instead of head. For example, $p = h_{rch}(p_0 + 4) = \mathsf{tail}(p_0)$.

This completes the proof of Lemma 5.2. □

LEMMA 5.3. *The equation*

$$h'_{fin}((\varphi(p)) + k) = \varphi^*(h_{rch}(p + k))$$

holds for all pointers $p \in \mathrm{dom}(h_{rch}) = s'(\mathsf{RCH}) = s'(\mathsf{FWD})$ *and* $k \in \{0, 4\}$.

PROOF. Let p be such a pointer. We show the desired equation by cases on whether k is 0 or 4.

If $k = 0$, we show $h'_{fin}(\varphi(p)) = \varphi^*(h_{rch}(p))$. Note first that $\mathsf{head}(p) = h_{rch}(p)$ because of $pairheap(\mathsf{RCH}, \mathsf{head}, \mathsf{tail}, h_{rch})$.

The fact that $p \in s'(\mathsf{FWD})$ implies (by I), that there is a $q \in \mathsf{FIN}$ with $q = \varphi(p)$. For this q, there is a q_0 such that q points to q_0, so that $q_0 = h'_{fin}(q) = h'_{fin}(\varphi(p))$, and

$$\begin{aligned} q_0 &= \varphi^*(\mathsf{head} \circ (\varphi^{-1}(q))) \\ &= \varphi^*(\mathsf{head} \circ (\varphi^{-1}(\varphi(p)))) \\ &= \varphi^*(\mathsf{head}(p)) \\ &= \varphi^*(h_{rch}(p)), \end{aligned}$$

as desired.

If k = 4, we show $h'_{fin}(\varphi(p) + 4) = \varphi^*(h_{rch}(p + 4))$. Again, since $p \in \mathsf{FWD}$, there is a $q \in \mathsf{FIN}$ such that $q = \varphi(p)$. For this q, there is a q_1 such that h'_{fin} maps the location $q + 4$ to q_1 (*i.e.*, $h'_{fin}(\varphi(p) + 4) = q_1$). At the same time,

$$\begin{aligned} q_1 &= \varphi^*(\mathsf{tail} \circ \varphi^{-1}(q)) \\ &= \varphi^*(\mathsf{tail} \circ \varphi^{-1}(\varphi(p))) \\ &= \varphi^*(\mathsf{tail}(p)) \\ &= \varphi^*(h_{rch}(p)), \end{aligned}$$

as desired. This completes the proof of Lemma 5.3. □

Now, since I implies that φ is a bijection between $[\![\mathsf{FWD}]\!]s' = [\![\mathsf{RCH}]\!]s' = [\![\mathsf{RCH}]\!]s$ and $[\![\mathsf{FIN}]\!]s' = [\![\mathsf{BUSY}]\!]s'$, and the equation in Lemma 5.3 holds, we only need to show that $\varphi([\![\mathsf{root}]\!]s) = [\![\mathsf{offset}]\!]s'$ to conclude that the condition (e') holds for the state (s', h') after execution of GC^*. But this holds since $(\mathsf{root}, \mathsf{offset}) \in \varphi$ is part of I and root is not modified by GC^*.

6. PROOFS

In this section, we show the two specifications:

$$\begin{array}{l}\{\mathsf{InitAss}\}\\ \quad \mathsf{INIT}^*\\ \{\mathsf{I}\}\end{array} \quad \text{and} \quad \begin{array}{l}\{\mathsf{I} \wedge \neg(\mathsf{scan} = \mathsf{free})\}\\ \quad \mathsf{BODY}\\ \{\mathsf{I}\}\end{array} \quad ,$$

where INIT^* is the code before the **while** loop, and BODY is the body of the loop. The assertions InitAss and I are those formulated in Sections 5.2 and 5.3.

Both of these proofs use *local reasoning*. Recall that the idea is to infer a local specification for program fragments that manipulate the heap. These local specifications mention exactly the parts of the heap that are manipulated by the fragments, and the frame rule is then applied to obtain a global specification. Without the separating conjunction $*$ and the frame rule, we would have to assert complicated non-interference claims each time we manipulate the heap. This is discussed further in Section 7.1.

6.1 Establishing the Invariant

We show that INIT^* establishes I when run in a state satisfying the precondition InitAss from Section 5.2. Therefore, let INIT and INIT^* be the code fragments

$$\begin{array}{rl}\mathsf{INIT} \equiv & t_1 := [\mathsf{root}];\\ & t_2 := [\mathsf{root} + 4];\\ & [\mathsf{free}] := t_1;\\ & [\mathsf{free} + 4] := t_2;\\ & [\mathsf{root}] := \mathsf{free}\end{array}$$

and

$$\begin{array}{rl}\mathsf{INIT}^* \equiv & \mathsf{scan} := \mathsf{offset};\\ & \mathsf{free} := \mathsf{offset};\\ & \mathsf{FWD} := \emptyset;\\ & \mathsf{UFWD} := \mathsf{RCH};\\ & \varphi := \emptyset;\\ & \mathsf{INIT};\\ & \mathsf{FORW} := \mathsf{FORW} \cup \{\mathsf{root}\};\\ & \mathsf{UNFORW} := \mathsf{UNFORW} \setminus \{\mathsf{root}\};\\ & \varphi := \varphi \cup \{(\mathsf{root}, \mathsf{free})\};\\ & \mathsf{free} := \mathsf{free} + 8\,.\end{array}$$

As mentioned, we use local reasoning and thus we first infer a local specification for INIT, and then use this local specification and the frame rule to obtain a global specification for INIT^*.

The local specification for INIT below only mentions the locations that are read or manipulated by INIT. We use the notation $\{A\} \Rightarrow \{B\}$ for applications of the

rule of consequence, *i.e.*, to denote that $A \to B$ is valid.

$$
\begin{array}{l}
\left\{ \begin{array}{l} (\exists y.\ (\mathsf{root}, y) \in \mathsf{head} \wedge \mathsf{root} \mapsto y) * (\exists y'.\ (\mathsf{root}, y') \in \mathsf{tail} \wedge \mathsf{root} + 4 \mapsto y') * \\ (\mathsf{free} \mapsto -, -) \end{array} \right\} \\
\quad t_1 := [\mathsf{root}] \\
\left\{ \begin{array}{l} ((\mathsf{root}, t_1) \in \mathsf{head} \wedge \mathsf{root} \mapsto t_1) * (\exists y'.\ (\mathsf{root}, y') \in \mathsf{tail} \wedge \mathsf{root} + 4 \mapsto y') * \\ (\mathsf{free} \mapsto -, -) \end{array} \right\} \\
\quad t_2 := [\mathsf{root} + 4] \\
\left\{ \begin{array}{l} ((\mathsf{root}, t_1) \in \mathsf{head} \wedge \mathsf{root} \mapsto t_1) * ((\mathsf{root}, t_2) \in \mathsf{tail} \wedge \mathsf{root} + 4 \mapsto t_2) * \\ (\mathsf{free} \mapsto -, -) \end{array} \right\} \\
\quad [\mathsf{free}] := t_1 \\
\quad [\mathsf{free} + 4] := t_2 \\
\left\{ \begin{array}{l} ((\mathsf{root}, t_1) \in \mathsf{head} \wedge \mathsf{root} \mapsto t_1) * ((\mathsf{root}, t_2) \in \mathsf{tail} \wedge \mathsf{root} + 4 \mapsto t_2) * \\ (\mathsf{free} \mapsto t_1, t_2) \end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l} (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge ((\mathsf{root} \mapsto t_1) * (\mathsf{root} + 4 \mapsto t_2) * \\ (\mathsf{free} \mapsto t_1, t_2)) \end{array} \right\} \\
\quad [\mathsf{root}] := \mathsf{free} \\
\left\{ \begin{array}{l} (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge ((\mathsf{root} \mapsto \mathsf{free}, -) * \\ (\mathsf{free} \mapsto t_1, t_2)) \end{array} \right\}
\end{array}
$$

We explain the first of these specifications in detail. First, the specification

$$
\begin{array}{l}
\{(\exists y.\ (\mathsf{root}, y) \in \mathsf{head} \wedge \mathsf{root} \mapsto y)\} \\
\quad t_1 := [\mathsf{root}] \\
\{((\mathsf{root}, t_1) \in \mathsf{head} \wedge \mathsf{root} \mapsto t_1)\}
\end{array}
$$

is valid by the rule (21) for heap lookup. The first specification above is then valid by the frame rule. The second specification is valid by the same argument. For the third specification, we use the frame rule again, along with two applications of the rule (22) for heap update. The implication follows from Remark 3.7 (we use purity of the assertions $(\mathsf{root}, t_1) \in \mathsf{head}$ and $(\mathsf{root}, t_2) \in \mathsf{tail}$), and for the last specification, we use the rule (22) for update, the frame rule, and an obvious rule for existentials, to "forget" an occurence of t_2.

From the local specification for INIT above, we infer the following specification for INIT* using the frame rule. We have labeled the steps with numbers and emphasized changes from the preceding stage in the assertions by underlining, to ease the reading. For brevity, we use φ' instead of $\varphi \setminus \{(\mathsf{root}, \mathsf{free} - 8)\}$ in the last step below.

$\{\mathsf{InitAss}\}$

(1) $\mathsf{scan} := \mathsf{offset};\ \mathsf{free} := \mathsf{offset};\ \mathsf{FWD} := \emptyset;\ \mathsf{UFWD} := \mathsf{RCH};\ \varphi := \emptyset;$

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \underline{\mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \mathsf{UFWD} = \mathsf{RCH}} \wedge \\ ((\forall_* x \in \mathsf{RCH}.\ ((\exists y.\ (x,y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x,y') \in \mathsf{tail} \wedge x+4 \mapsto y'))) * \\ (\forall_* x \in \mathsf{NEW}.\ x \mapsto -,-)) \end{array} \right\}$$

(2)⇓

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \underline{\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \wedge \mathsf{root} \in \mathsf{UFWD}} \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \mathsf{UFWD} = \mathsf{RCH} \wedge \\ (\underline{\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{FREE}}}) \end{array} \right\}$$

(3)⇓

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \wedge \mathsf{root} \in \mathsf{UFWD} \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \mathsf{UFWD} = \mathsf{RCH} \wedge \\ ((\underline{\mathsf{A}_{\mathsf{UFWD}-\mathsf{root}}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FIN}} * \underline{\mathsf{A}_{\mathsf{FREE}-\mathsf{free}}}) * \\ (\underline{(\exists y.\ (\mathsf{root}, y) \in \mathsf{head} \wedge \mathsf{root} \mapsto y)} * \underline{(\exists y'.\ (\mathsf{root}, y') \in \mathsf{tail} \wedge \mathsf{root}+4 \mapsto y')} * \\ \underline{(\mathsf{free} \mapsto -,-)})) \end{array} \right\}$$

(4) INIT

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \wedge \mathsf{root} \in \mathsf{UFWD} \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \mathsf{UFWD} = \mathsf{RCH} \wedge \\ ((\mathsf{A}_{\mathsf{UFWD}-\mathsf{root}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\ (((\underline{\mathsf{root} \mapsto \mathsf{free}, -}) * (\mathsf{free} \mapsto t_1, t_2)) \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \underline{\mathsf{tail}})) \end{array} \right\}$$

(5)⇓

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \wedge \mathsf{root} \in \mathsf{UFWD} \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FWD} = \emptyset \wedge \varphi = \emptyset \wedge \mathsf{UFWD} = \mathsf{RCH} \wedge \\ \underline{(\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \neg(\mathsf{root} \in \mathsf{FWD}) \wedge \neg((\mathsf{root}, \mathsf{free}) \in \varphi)} \wedge \\ ((\mathsf{A}_{\mathsf{UFWD}-\mathsf{root}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\ ((\mathsf{root} \mapsto \mathsf{free}, -) * (\mathsf{free} \mapsto t_1, t_2))) \end{array} \right\}$$

(6) $\mathsf{FWD} := \mathsf{FWD} \cup \{\mathsf{root}\};\ \mathsf{UFWD} := \mathsf{UFWD} \setminus \{\mathsf{root}\};$
$\varphi := \varphi \cup \{(\mathsf{root}, \mathsf{free})\};\ \mathsf{free} := \mathsf{free} + 8$

$$\left\{ \begin{array}{l} \mathsf{I}_c \wedge \underline{\mathsf{iso}(\varphi', \mathsf{FWD} \setminus \{\mathsf{root}\}, \mathsf{BUSY} \setminus \{\mathsf{free}-8\})} \wedge \\ \underline{(\mathsf{FWD} \setminus \{\mathsf{root}\}) \cup (\mathsf{UFWD} \cup \{\mathsf{root}\}) = \mathsf{RCH})} \wedge \\ \underline{(\mathsf{root} \in \mathsf{FWD}) \wedge \neg(\mathsf{root} \in \mathsf{UFWD}) \wedge (\mathsf{root}, \mathsf{free}-8) \in \varphi} \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \underline{\mathsf{offset} = \mathsf{free}-8} \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\ ((\forall_* x \in ((\underline{\mathsf{UFWD} \cup \{\mathsf{root}\}}) \setminus \{\mathsf{root}\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \wedge x \mapsto y) * \\ \quad (\exists y'.\ (x,y') \in \mathsf{tail} \wedge x+4 \mapsto y'))) * \\ (\forall_* x \in (\underline{\mathsf{FWD} \setminus \{\mathsf{root}\}}).\ (\exists y.\ (x,y) \in \underline{\varphi'} \wedge x \mapsto y, -)) * \\ (\forall_* x \in \mathsf{FIN}.\quad ((\exists y.\ (x,y) \in \underline{\varphi'} \odot (\mathsf{head} \circ (\underline{\varphi'})^\dagger) \wedge x \mapsto y) * \\ \quad (\exists y'.\ (x,y') \in \underline{\varphi'} \odot (\mathsf{tail} \circ (\underline{\varphi'})^\dagger) \wedge x+4 \mapsto y'))) * \\ (\forall_* x \in (\underline{\mathsf{UFIN} \setminus \{(\mathsf{free}-8)\}}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ (\underline{\varphi'})^\dagger \wedge x \mapsto y) * \\ \quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ (\underline{\varphi'}) \wedge x+4 \mapsto y'))) * \\ (\forall_* x \in ((\underline{\mathsf{FREE} \cup \{(\mathsf{free}-8)\}}) \setminus \{(\mathsf{free}-8)\}).\ x \mapsto -,-) * \\ (\mathsf{root} \mapsto (\underline{\mathsf{free}-8}), -) * ((\underline{\mathsf{free}-8}) \mapsto t_1, t_2)) \end{array} \right\}$$

In this derivation, the first step uses Hoare's rule for assignment several times. The second step uses the rule (11) for $\forall_*$ (to conclude $\mathsf{A}_{\mathsf{FWD}}, \mathsf{A}_{\mathsf{FIN}}$, and $\mathsf{A}_{\mathsf{UFIN}}$) and (10) (for $\mathsf{A}_{\mathsf{UFWD}}$ and $\mathsf{A}_{\mathsf{FREE}}$). This step also uses the rules (34), (83), (57), and (44). The third step uses the rule (14) and the fact that $\mathsf{free} \in \mathsf{FREE}$ by definition, and we use the frame rule and purity along with our local specification from before to

take the fourth step. The fifth step is a consequence of purity and the rules (55), (56), and the last step follows from Hoare's rule for assignment.

We must show that the invariant I follows from the conclusion in this derivation. We make an informal argument of this here, but a completely formal argument may be found in Appendix C.

For the pure parts of I, the only thing to notice is that since $\varphi \setminus \{(\mathsf{root}, \mathsf{free} - 8)\}$ is a bijection, we can conclude that when we add the pair $(\mathsf{root}, \mathsf{free} - 8)$ to it and also add the two pointers to the relevant sets, it is still a bijection.

For the impure part of I, we argue that each of the iterated separating conjunctions can be inferred from the corresponding parts of the conclusion above.

The iterated separating conjunction $\mathsf{A}_{\mathsf{UFWD}}$ over UFWD can be inferred from the conclusion because $\neg(\mathsf{root} \in \mathsf{UFWD})$ implies that

$$(\mathsf{UFWD} \cup \{\mathsf{root}\}) \setminus \{\mathsf{root}\} = \mathsf{UFWD}.$$

The same argument can be used for $\mathsf{A}_{\mathsf{FREE}}$. The assertion $\mathsf{A}_{\mathsf{FWD}}$ can be inferred from the iterated separating conjunction over $\mathsf{FWD} \setminus \{\mathsf{root}\}$, and $\mathsf{root} \mapsto (\mathsf{free} - 8), -$ in the conclusion above, since $\mathsf{root} \in \mathsf{FWD}$ and we can infer the assertion

$$(\exists y.\ (\mathsf{root}, y) \in \varphi \wedge \mathsf{root} \mapsto y, -)$$

for root, and therefore add root to the set $\mathsf{FWD} \setminus \{\mathsf{root}\}$ over which we quantify in the iterated separating conjunction in the conclusion above. For $\mathsf{A}_{\mathsf{FIN}}$ we can use the conjunct $\mathsf{scan} = \mathsf{free}$ to infer that both of the assertions about FIN are equivalent to emp.

For $\mathsf{A}_{\mathsf{UFIN}}$, we can use the parts of the conclusion above that involve φ, head, and tail along with what we know about $\mathsf{free} - 8$, to infer

$$\begin{array}{l}(\exists y.\ (\mathsf{free} - 8) \mapsto y \wedge (\mathsf{free} - 8, y) \in \mathsf{head} \circ \varphi^{\dagger}) * \\ (\exists y'.\ ((\mathsf{free} - 8) + 4) \mapsto y' \wedge (\mathsf{free} - 8, y') \in \mathsf{tail} \circ \varphi^{\dagger}),\end{array}$$

and then add $\mathsf{free} - 8$ to $\mathsf{UFIN} \setminus \{\mathsf{free} - 8\}$, to obtain the desired iterated separating conjunction.

This establishes that running INIT^* starting from a state satisfying the assertion InitAss from Section 5.2 terminates in a state that satisfies I, as desired.

6.2 Maintaining the Invariant

We have shown that I is established by the initializing code. The next step is to show that I is indeed an invariant, *i.e.*, that the specification

$$\begin{array}{l}\{\mathsf{I} \ \wedge \ \mathsf{scan} \neq \mathsf{free}\} \\ \quad \mathsf{BODY} \\ \{\mathsf{I}\}\end{array}$$

holds, where BODY is the body of the **while** loop. Note that BODY consists of two similar parts ScanCar and ScanCdr, one for each field of the cell pointed to by scan, they are marked in the code with comments. Between these halves, that cell is in a "mixed state": the first field of it is finished, whereas the other is about to

be scanned. The aim is thus to show that the following specifications hold.

$$\begin{array}{l}\{\mathsf{I} \ \wedge \ \mathsf{scan} \neq \mathsf{free}\}\\ \quad \mathsf{ScanCar};\\ \{\mathsf{I}'\}\\ \quad \mathsf{ScanCdr};\\ \quad \mathsf{scan} := \mathsf{scan} + 8\\ \{\mathsf{I}\},\end{array} \tag{26}$$

where I' is an assertion which holds in the intermediate state where scan is "halfway between UFIN and FIN":

$$\begin{array}{l}\mathsf{I}' \equiv\\ \mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge\\ (\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}} *\\ (\exists y.\ (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^{\dagger}) \wedge \mathsf{scan} \mapsto y) *\\ (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto y'))\end{array}$$

We focus on showing the first of the involved specifications, $\{\mathsf{I}\}\ \mathsf{ScanCar}\ \{\mathsf{I}'\}$. The proof of the other is analogous and not described in all details. First, we describe ScanCar informally. It "scans" the first field in the cell pointed to by scan, and there are three branches according to the value a in it (and maybe the place it points to):

(1) If a is a non-pointer, nothing happens.
(2) If a is a pointer, we branch according to the value $b = [a]$ of the first field of the cell pointed to by a.
   (a) If b is a forwarding pointer, i.e., a pointer in NEW, we just update [scan] to b.
   (b) If b is not a forwarding pointer, we copy the cell and update $[a]$ to a forwarding pointer, and we also update [scan] to point to the new copy.

The effect of the command $a := [\mathsf{scan}]$ is formalized in this specification.

$$\begin{array}{l}\{\mathsf{I} \wedge \mathsf{scan} \neq \mathsf{free}\}\\ \Downarrow\\ \left\{\begin{array}{l}\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \underline{\mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}}} * \mathsf{A}_{\mathsf{FREE}}) *\\ \underline{(\exists y.\ (\mathsf{scan}, y) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{scan} \mapsto y)} * \underline{(\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto y'))}\end{array}\right\}\\ \quad a := [\mathsf{scan}]\\ \left\{\begin{array}{l}\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) *\\ \underline{((\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{scan} \mapsto a)} * (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto y'))\end{array}\right\}\\ \Downarrow\\ \left\{\begin{array}{l}\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \underline{(\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^{\dagger}} \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) *\\ \underline{(\mathsf{scan} \mapsto a)} * (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto y'))\end{array}\right\}.\end{array}$$

The first step here is due to (14), the specification step uses the rule (21) for lookup, the frame rule and purity; the last rewriting uses purity.

Henceforth, we let I_a be the conclusion in the derivation above.

According to the rule of conditionals, there are two specifications to be shown, according to the outer **if**-branch in ScanCar. The first of these is

$$\begin{array}{l}\{\mathsf{I}_a \wedge \neg(a \bmod 8 = 0)\}\\ \Downarrow\\ \{\mathsf{I}_a \wedge \neg\mathsf{Ptr}(a)\}\\ \quad \mathbf{skip}\\ \{\mathsf{I}'\}\end{array} \tag{27}$$

This specification is shown in Sec. 6.2.1. The second specification we have to show for the outer **if**-branch contains an inner **if**-branch, so it too splits into two specifications. Before writing these down, we formalize the effect of the command $b := [a]$. We have

$$\begin{array}{ll} & \{\mathsf{I}_a \wedge a \bmod 8 = 0\}\\ (1)\ \Downarrow & \\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a)\}\\ (2)\ \Downarrow & \\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge a \in \mathsf{RCH}\}\\ (3)\ \Downarrow & \\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \in \mathsf{FWD} \vee a \in \mathsf{UFWD})\}\\ (4)\ \Downarrow & \\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow - \vee a \hookrightarrow -)\}\\ (5)\ \Downarrow & \\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow -)\}\\ & \quad b := [a]\\ & \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b)\}\end{array}$$

The first of the implications uses (49), and the second follows from $(\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH})$, (73), and (74). The third follows from (77), and the fourth from (12). Finally the specification is an instance of the rule for lookup and the frame rule, since b does not occur free in $\mathsf{I}_a \wedge \mathsf{Ptr}(a)$.

According to the rule of conditionals, there are two more specifications to show to conclude the desired specification $\{\mathsf{I} \wedge \neg(\mathsf{scan} = \mathsf{free})\}\ \mathsf{ScanCar}\ \{\mathsf{I}'\}$. They are

$$\begin{array}{l}\{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge b \bmod 8 = 0 \wedge \mathsf{offset} \le b \le \mathsf{maxFree}\}\\ \Downarrow\\ \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge b \in \mathsf{NEW}\}\\ \quad [\mathsf{scan}] := b\\ \{\mathsf{I}'\}\end{array} \tag{28}$$

and

$$\begin{array}{l}\{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge \neg(b \bmod 8 = 0 \wedge \mathsf{offset} \le b \le \mathsf{maxFree})\}\\ \Downarrow\\ \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge \neg(b \in \mathsf{NEW})\}\\ \quad \mathsf{CopyCell}^*\\ \{\mathsf{I}'\},\end{array} \tag{29}$$

where

$$\begin{array}{rcl}\mathsf{CopyCell}^* & \equiv & t_1 := [a];\\ & & t_2 := [a+4];\\ & & [\mathsf{free}] := t_1;\\ & & [\mathsf{free}+4] := t_2;\\ & & [a] := \mathsf{free};\\ & & [\mathsf{scan}] := \mathsf{free};\\ & & \mathsf{FWD} := \mathsf{FWD} \cup \{a\};\\ & & \mathsf{UFWD} := \mathsf{UFWD} \setminus \{a\};\\ & & \varphi := \varphi \cup \{(a, \mathsf{free})\};\\ & & \mathsf{free} := \mathsf{free} + 8\end{array}$$

These specifications are shown in Sec. 6.2.2 and Sec. 6.2.3, respectively. But first, we note a lemma for later use.

LEMMA 6.1. I_{pure} *implies* $\mathsf{free} \leq \mathsf{maxFree}$.

PROOF. By the rule (46) for intervals with a common start-point, it suffices to show that $\#\mathsf{BUSY} \leq \#\mathsf{NEW}$. But this follows from

$$\#\mathsf{BUSY} = \#\mathsf{FWD} \leq \#\mathsf{RCH} \leq \#\mathsf{NEW}$$

The first equality follows from $\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY})$ and (65), whereas the first inequality follows from (44), (79), and $\mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH}$. The last inequality is part of I_{pure}. □

6.2.1 *If Nothing Happens.* We show that the specification (27) holds. According to the rule for **skip** and the rule of consequence, that amounts to

LEMMA 6.2. *The assertion*

$$A \equiv \mathsf{I}_a \wedge \neg \mathsf{Ptr}(a)$$

implies I'.

PROOF. We have

$$\begin{array}{l}\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \neg \mathsf{Ptr}(a) \wedge (\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) *\\ (\mathsf{scan} \mapsto a) * (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y'))\\ \Downarrow\\ \mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \neg \mathsf{Ptr}(a) \wedge \underline{(\mathsf{scan}, a) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)} \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) *\\ (\mathsf{scan} \mapsto a) * (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y'))\\ \Downarrow\\ \mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \neg \mathsf{Ptr}(a) \wedge\\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) *\\ (\underline{(\mathsf{scan}, a) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)} \wedge \mathsf{scan} \mapsto a) *\\ (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y'))\\ \Downarrow\\ \mathsf{I}'\end{array}$$

The first implication follows from the fact that A implies $(\mathsf{scan}, a) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)$ by (67). The second implication follows from purity. □

This implies that the specification (27) holds.

6.2.2 *If We do not Copy.* We show the specification (28) in this section. The proof goes as follows: first, we show that the precondition implies $a \in \mathsf{FWD}$, and use this to infer $(\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)$. Then we use a local specification to infer the desired global specification.

LEMMA 6.3. *The assertion*

$$A \equiv \mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge b \in \mathsf{NEW}$$

implies $a \in \mathsf{FWD}$.

PROOF. We give an informal argument here, and we refer to Appendix D for a more formal proof. We have $a \in \mathsf{RCH}$ which is equal to the union of FWD and UFWD, so it suffices to assume $a \in \mathsf{UFWD}$ and derive a contradiction. If $a \in \mathsf{UFWD}$, we know that the assertion in the body of $\mathsf{A}_{\mathsf{UFWD}}$ holds for a. This implies that whatever a points to is related to a by head, and since any pointer in the range of head is is RCH, which is disjoint from NEW, we get the desired contradiction from $a \hookrightarrow b$ and $b \in \mathsf{NEW}$. □

LEMMA 6.4. *The assertion A from Lemma 6.3 implies* $(\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)$.

PROOF. We use Lemma 6.3 and show $A \wedge a \in \mathsf{FWD} \rightarrow (\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)$. By (68),

$$(\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge A \wedge (a, b) \in \varphi \rightarrow (\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger),$$

so it suffices to show

$$A \wedge a \in \mathsf{FWD} \rightarrow (a, b) \in \varphi.$$

Like before, we use (16) and show

$$\mathsf{A}_{\mathsf{FWD}} \wedge a \hookrightarrow b \wedge a \in \mathsf{FWD} \rightarrow (a, b) \in \varphi.$$

We have

$$\begin{array}{l} \mathsf{A}_{\mathsf{FWD}} \wedge a \hookrightarrow b \wedge a \in \mathsf{FWD} \\ \Downarrow \\ (\underline{\mathsf{A}_{\mathsf{FWD}-a}} * \underline{(\exists y.\ (a, y) \in \varphi \wedge a \mapsto y, -)}) \wedge a \hookrightarrow b \\ \Downarrow \\ \mathsf{A}_{\mathsf{FWD}-a} * (\underline{(a, b) \in \varphi \wedge a \mapsto b, -}) \\ \Downarrow \\ (a, b) \in \varphi \end{array}$$

In this derivation, we have first used (14), then (18), and finally purity. □

We turn to the local specification for this branch of the program. Again, it only mentions the footprint of the branch:

$$\begin{array}{l} \{\mathsf{scan} \mapsto - \wedge (\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)\} \\ \quad [\mathsf{scan}] := b \\ \{\mathsf{scan} \mapsto b \wedge (\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)\} \\ \Downarrow \\ \{\exists y.\ (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto y\} \end{array} \tag{30}$$

The first step follows from the rule (22) for heap update and the rule of conjunction.

We can now show a global specification for $[\mathsf{scan}] := b$.

$$\begin{array}{l} \{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge b \in \mathsf{NEW}\} \\ \Downarrow \\ \{\mathsf{I}_a \wedge \underline{(\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)}\} \\ \Downarrow \\ \left\{ \begin{array}{l} \mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\ (\mathsf{scan} \mapsto a \wedge \underline{(\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)}) * \\ (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y')) \end{array} \right\} \\ \quad [\mathsf{scan}] := b \\ \left\{ \begin{array}{l} \mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\ (\underline{\mathsf{scan} \mapsto b} \wedge (\mathsf{scan}, b) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)) * \\ (\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y')) \end{array} \right\} \\ \Downarrow \\ \mathsf{I}' \end{array} \tag{31}$$

For the first implication, we use Lemma 6.4, and for the second, we use purity. The specification step follows from our local specification (30), the frame rule, and purity. This proves the desired global specification (28).

REMARK 6.5. *If we did not have the separating conjunction $*$, the specification step in (31) would require us to ensure that the assignment to the heap cell* [scan] *does not affect any of the assertions* $\mathsf{A}_\_$*, via non-interference predicates stating that, e.g.,* scan *is not in any of the sets involved in the specification.*

We are now ready to address the specification (29) for the most complicated branch of ScanCar. The code resembles INIT^*, hence the proof of its specification will be similar to the proof in Section 6.1.

6.2.3 *If We Copy.* We show that the specification (29) is derivable. To this end, $\mathsf{CopyCell}^*$ is split into two parts:

$$\begin{array}{lll} \mathsf{CopyCell} \equiv & t_1 := [a]; & \mathsf{Increment} \equiv \ \mathsf{FWD} := \mathsf{FWD} \cup \{a\}; \\ & t_2 := [a+4]; & \quad\ \mathsf{UFWD} := \mathsf{UFWD} \setminus \{a\}; \\ & [\mathsf{free}] := t_1; & \quad\ \varphi := \varphi \cup \{(a, \mathsf{free})\}; \\ & [\mathsf{free}+4] := t_2; & \quad\ \mathsf{free} := \mathsf{free} + 8; \\ & [a] := \mathsf{free}; & \\ & [\mathsf{scan}] := \mathsf{free}; & \end{array}$$

We first show that in this case, $a \in \mathsf{UFWD}$. Then we derive a local specification for CopyCell, which leads to the desired global specification for CopyCell*.

LEMMA 6.6. *The assertion*

$$A \equiv \mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge \neg(b \in \mathsf{NEW})$$

implies $a \in \mathsf{UFWD} \wedge \neg(a \in \mathsf{FWD})$.

PROOF. We give an informal proof here; a completely formal proof is given in Appendix E. The proof goes by contradiction (as the proof of Lemma 6.3); this time we assume $a \in \mathsf{FWD}$ and derive a contradiction. As in the proof just mentioned, if a is in FWD, then whatever a points to is related to a in φ, which is a bijection which has BUSY as its codomain. This contradicts that a points to b which is not in NEW. □

We turn to the local specification for CopyCell. As usual, it only involves the footprint of the program fragment.

$$
\begin{array}{l}
\left\{ \begin{array}{l} (\exists y.\ (a,y) \in \mathsf{head} \wedge a \mapsto y) * (\exists y'.\ (a,y') \in \mathsf{tail} \wedge a+4 \mapsto y') * \\ (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-) \end{array} \right\} \\
\quad t_1 := [a] \\
\left\{ \begin{array}{l} (\underline{(a,t_1) \in \mathsf{head} \wedge a \mapsto t_1}) * (\exists y.\ (a,y') \in \mathsf{tail} \wedge a+4 \mapsto y') * \\ (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-) \end{array} \right\} \\
\quad t_2 := [a+4] \\
\left\{ \begin{array}{l} ((a,t_1) \in \mathsf{head} \wedge a \mapsto t_1) * (\underline{(a,t_2) \in \mathsf{tail} \wedge a+4 \mapsto t_2}) * \\ (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-) \end{array} \right\} \\
\quad [\mathsf{free}] := t_1 \\
\left\{ \begin{array}{l} ((a,t_1) \in \mathsf{head} \wedge a \mapsto t_1) * ((a,t_2) \in \mathsf{tail} \wedge a+4 \mapsto t_2) * \\ (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto \underline{t_1},-) \end{array} \right\} \\
\quad [\mathsf{free}+4] := t_2 \\
\left\{ \begin{array}{l} ((a,t_1) \in \mathsf{head} \wedge a \mapsto t_1) * ((a,t_2) \in \mathsf{tail} \wedge a+4 \mapsto t_2) * \\ (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,\underline{t_2}) \end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l} ((a \mapsto t_1) * (a+4 \mapsto t_2) * (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,t_2)) \wedge \\ \underline{(a,t_1) \in \mathsf{head} \wedge (a,t_2) \in \mathsf{tail}} \end{array} \right\} \\
\quad [a] := \mathsf{free} \\
\left\{ \begin{array}{l} ((\underline{a \mapsto \mathsf{free}}) * (a+4 \mapsto t_2) * (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,t_2)) \wedge \\ (a,t_1) \in \mathsf{head} \wedge (a,t_2) \in \mathsf{tail} \end{array} \right\} \\
\quad [\mathsf{scan}] := \mathsf{free} \\
\left\{ \begin{array}{l} ((a \mapsto \mathsf{free}) * (a+4 \mapsto t_2) * (\underline{\mathsf{scan} \mapsto \mathsf{free}}) * (\mathsf{free} \mapsto t_1,t_2)) \wedge \\ (a,t_1) \in \mathsf{head} \wedge (a,t_2) \in \mathsf{tail} \end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l} ((\underline{a \mapsto \mathsf{free},-}) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1,t_2)) \wedge \\ (a,t_1) \in \mathsf{head} \wedge (a,t_2) \in \mathsf{tail} \end{array} \right\}
\end{array}
\tag{32}
$$

The implication in the middle of this derivation is due to pureness, and the rest of the steps use the rules for lookup (21) and update (22), along with the frame rule.

We now infer the specification for $\mathsf{CopyCell}^*$ via the local specification (32). In this derivation, we write φ' instead of $\varphi \setminus \{(a, \mathsf{free} - 8)\}$ for brevity.

$$
\begin{array}{l}
\{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge \neg(b \in \mathsf{NEW})\} \\
\Downarrow \\
\{\mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge \underline{a \in \mathsf{UFWD} \wedge \neg(a \in \mathsf{FWD})}\} \\
\Downarrow \\
\left\{ \begin{array}{l}
\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge (\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(a) \wedge a \in \mathsf{UFWD} \wedge \\
\neg(a \in \mathsf{FWD}) \wedge \forall x.\ \neg((a, x) \in \varphi) \wedge \\
((\underline{\mathsf{A}_{\mathsf{UFWD}-a}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \underline{\mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}}} * \underline{\mathsf{A}_{\mathsf{FREE}-\mathsf{free}}}) * \\
\underline{(\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y')} * \\
\underline{(\exists y.\ (a, y) \in \mathsf{head} \wedge a \mapsto y)} * \underline{(\exists y'.\ (a, y') \in \mathsf{tail} \wedge a + 4 \mapsto y')} * \\
\underline{(\mathsf{scan} \mapsto -)} * \underline{(\mathsf{free} \mapsto -, -)})
\end{array} \right\} \\
\quad \mathsf{CopyCell} \\
\left\{ \begin{array}{l}
\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge (\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(a) \wedge a \in \mathsf{UFWD} \wedge \\
\neg(a \in \mathsf{FWD}) \wedge \forall x.\ \neg((a, x) \in \varphi) \wedge \\
((\mathsf{A}_{\mathsf{UFWD}-a} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\
(\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y') * \\
(\underline{(a \mapsto \mathsf{free}, -)} * \underline{(\mathsf{scan} \mapsto \mathsf{free})} * \underline{(\mathsf{free} \mapsto t_1, t_2)}) \wedge \underline{(a, t_1) \in \mathsf{head}} \wedge \underline{(a, t_2) \in \mathsf{tail}})
\end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l}
\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge (\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(a) \wedge a \in \mathsf{UFWD} \wedge \\
\neg(a \in \mathsf{FWD}) \wedge \neg(\underline{(a, \mathsf{free}) \in \varphi}) \wedge \underline{(a, t_1) \in \mathsf{head}} \wedge \underline{(a, t_2) \in \mathsf{tail}} \wedge \\
((\mathsf{A}_{\mathsf{UFWD}-a} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\
(\exists y'.\ (\mathsf{scan}, y') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto y') * \\
((a \mapsto \mathsf{free}, -) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1, t_2)))
\end{array} \right\} \\
\quad \mathsf{FWD} := \mathsf{FWD} \cup \{a\};\ \mathsf{UFWD} := \mathsf{UFWD} \setminus \{a\}; \\
\quad \varphi := \varphi \cup \{(a, \mathsf{free})\};\ \mathsf{free} := \mathsf{free} + 8 \\
\left\{ \begin{array}{l}
\mathsf{I}_c \wedge \mathsf{root} \in \mathsf{FWD} \setminus \{a\} \wedge \\
\underline{\mathsf{iso}(\varphi', \mathsf{FWD} \setminus \{a\}, \mathsf{BUSY} \setminus \{\mathsf{free} - 8\})} \wedge \\
\underline{(\mathsf{RCH} = (\mathsf{FWD} \setminus \{a\}) \cup (\mathsf{UFWD} \cup \{a\}))} \wedge \\
\underline{\mathsf{scan} \leq \mathsf{free} - 8} \wedge \mathsf{offset} \leq \mathsf{scan} \wedge \underline{\mathsf{Ptr}(\mathsf{free} - 8)} \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\underline{\mathsf{scan} \neq \mathsf{free} - 8} \wedge \underline{(\mathsf{scan}, a) \in \mathsf{head} \circ (\varphi')^\dagger} \wedge \mathsf{Ptr}(a) \wedge \underline{\neg(a \in \mathsf{UFWD})} \wedge \\
\underline{a \in \mathsf{FWD}} \wedge (a, t_1) \in \mathsf{head} \wedge (a, t_2) \in \mathsf{tail} \wedge \\
((\forall_* x \in ((\underline{\mathsf{UFWD} \cup \{a\}}) \setminus \{a\}).\ ((\exists y.\ (x, y) \in \mathsf{head} \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x, y') \in \mathsf{tail} \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in (\underline{\mathsf{FWD} \setminus \{a\}}).\ (\exists y.\ (x, y) \in \underline{\varphi'} \wedge x \mapsto y, -)) * (a \mapsto (\underline{\mathsf{free} - 8}), -) * \\
(\forall_* x \in \mathsf{FIN}.\quad ((\exists y.\ (x, y) \in \underline{\varphi'} \odot (\mathsf{head} \circ (\underline{\varphi'})^\dagger) \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x, y') \in \underline{\varphi'} \odot (\mathsf{tail} \circ (\underline{\varphi'})^\dagger) \wedge x + 4 \mapsto y'))) * \\
(\forall_* x \in (\underline{\mathsf{UFIN} \setminus \{(\mathsf{free} - 8)\}}).\ ((\exists y.\ (x, y) \in \mathsf{head} \circ (\underline{\varphi'})^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x, y') \in \mathsf{tail} \circ (\underline{\varphi'}) \wedge x + 4 \mapsto y')) * ((\underline{\mathsf{free} - 8}) \mapsto t_1, t_2)) * \\
(\forall_* x \in ((\underline{\mathsf{FREE} \cup \{(\mathsf{free} - 8)\}}) \setminus \{(\mathsf{free} - 8)\}).\ x \mapsto -, -))
\end{array} \right\}
\end{array}
$$

The first implication follows from Lemma 6.6. The second follows from (14) and (66). The global specification for CopyCell follows from our local specification (32), purity, and the frame rule. The implication immediately thereafter is a consequence

of purity. The specification for the three auxiliary variables and the update of free follows from Hoare's rule for assignment and obvious rules for intervals.

REMARK 6.7. *Notice the crucial use of local reasoning in this derivation. If we did not have the separating conjunction, then for each of the updates of the heap in* CopyCell *in this global specification, we would have to make sure that the location we update does not interfere with each of the assertions* $\mathsf{A}_{\mathsf{UFWD}-a}, \mathsf{A}_{\mathsf{FWD}}$*, etc. Essentially the same remark can be made about the proof in Section 6.1 for* INIT.

Like in Section 6.1, we must now show that I' follows from the conclusion in the derivation above. The proof of this, however, is for the most part completely analogous to the proof there (if one replaces root by a). Therefore, we omit it here; the diligent reader may find a proof in Appendix F.

We therefore conclude that the first part of the specification (26) for the **while**-loop holds. The treatment of the other half will not be as detailed as this one, since the proofs are completely analogous for the most part. However, the specification for $\mathsf{scan} := \mathsf{scan} + 8$ needs an argument.

6.2.4 *After* ScanCdr. We show that the invariant I is established after running $\mathsf{ScanCdr}; \mathsf{scan} := \mathsf{scan}+8$ in a state in which I' holds. We omit the detailed proof for ScanCdr, since it is analogous to that of ScanCar. One can obtain the specification

$$\{\mathsf{I}'\}\ \ \mathsf{ScanCdr}\ \left\{\begin{array}{l}\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \\ ((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\ (\exists y.\ (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^{\dagger}) \wedge \mathsf{scan} \mapsto y) * \\ (\exists y'.\ (\mathsf{scan}, y') \in \varphi \odot (\mathsf{tail} \circ \varphi^{\dagger}) \wedge \mathsf{scan} + 4 \mapsto y'))\end{array}\right\}$$

Letting A be the conclusion in the above specification, we must show that

$$\{A\}\ \ \mathsf{scan} := \mathsf{scan} + 8\ \ \{\mathsf{I}\} \tag{33}$$

holds. Intuitively, this specification holds because by increasing scan, we move the border between the intervals FIN and UFIN, and so the cell that has been scanned by the current iteration of the **while** loop moves from UFIN to FIN. There is a formal derivation of (33) in Appendix G.

This means that the specification (26) holds, as desired, and hence we can conclude

THEOREM 6.8. *The implementation GC* of Cheney's algorithm in Appendix A is a correct copying garbage collector in the sense of Definition 5.1.*

7. CONCEPTUAL REMARKS

In this section, we discuss our proof at the meta-level. In particular, we illustrate what we gained from using separation logic, and in particular local reasoning. We also discuss our extensions of standard separation logic.

7.1 Benefits of Local Reasoning

Given that one of the aims of this paper is to demonstrate the power of separation logic, it is natural to ask to what extent local reasoning helped in the proof, and what we could have done without it.

A "brute force" way of answering this question would be to present a proof of our implementation that does not use local reasoning, and then compare the two proofs. That task, however, would be too tedious. Instead, we argue that proofs of "nontrivial" pointer-manipulating programs in general tend to be more complicated than proofs of the same programs in separation logic. We have also made remarks in the proof where we outline how local reasoning helped us (cf. Remarks 6.5 and 6.7).

A semantic analysis of pointer manipulations that does not use separation logic may be found in the papers [Calcagno etal. 2000] and [Bornat 2000]. In the first-mentioned paper, it is shown how to reason about programs that manipulate a list stored in the heap. The idea is to treat the list as a sequence of locations, and when a location is updated, the precondition is that the updated location is disjoint from the list. An important difference from our work is that lists are inductively defined; the structure we garbage collect can be cyclic, and thus cannot be defined inductively. In [Bornat 2000], proofs of several pointer manipulating programs are outlined. Some of these programs manipulate structures in the heap that are not inductively defined. In these cases, the approach is to determine the set of locations that is involved in the representation of a structure, and for each heap update, one has to ensure that the updated location is disjoint from this set, using non-interference predicates.

In contrast, local reasoning takes advantage of the $*$ connective and the frame rule to state the required non-interference *implicitly*. The proof of the critical operations, namely the heap updates, are simple and do not require any non-interference predicates.

7.2 Remarks on Our Extension of Separation Logic

As mentioned, our proof uses an extension of standard separation logic with finite sets, relations, paths, and the iterated separating conjunction $\forall_*$. It is therefore appropriate to discuss the applicability of these extensions of standard separation logic in other settings.

As mentioned, we believe that the approach with sets and relations is applicable in other settings where the goal is to establish the existence of a relation between the heap before and after execution of a program, in particular if the structure represented in the heap is not definable by induction. The approach would be similar to that used here: given a snapshot of execution like that of Fig. 3, it might be possible to divide the heap into disjoint portions where the locations in each portion have a certain property. Given such a partition, one can then use the $\forall_*$ connective to give an unambiguous description of the heap and use this in a specification, as we have done in our specification and proof.

On the other hand, it is quite possible that some of our extensions are of limited use in other settings. We use paths and the reachability predicates to express what is reachable before execution of our garbage collector. In a list reversal program, for example, this would be explicitly assumed in the specification. Also the PtrRg is

used to take a crucial step in the proof of Lemma 6.3, but it is not straightforward to think of other proofs of programs where such a predicate would be crucial.

8. RELATED WORK

There have been several proposals for ways of using types to manage the problem of reasoning about programs that manipulate imperative data structures [Crary etal. 1999; Smith etal. 2000; Ahmed etal. 2003; Petersen etal. 2003]. They are based on the idea that well-typed programs do not go wrong, but they are not aimed at giving proofs of *correctness*. In the work [Crary etal. 1999] on capabilities, traditional region calculus [Tofte and Talpin 1994] is extended with an annotation of a capability to each region, and this gives criteria to decide when it is safe to deallocate a region. In the setting of alias types [Smith etal. 2000], a static notion of constraint is used describe the shape of the heap, and this is used to decide when it is safe to execute a program. In the work [Ahmed etal. 2003] on hierarchical storage, ideas from BI [Pym 2002] and region calculi are used to give a type system with structure on locations. In [Petersen etal. 2003] Petersen et. al. propose to use a type theory based on ordered linear logic as a foundation for defining how data is laid out in memory. The type theory in [Petersen etal. 2003] builds upon a concrete allocation model such as the one provided by Cheney's copying garbage collector. In the paper [Hawblitzel etal. 2004] by Hawblitzel and others, low-level pointer operations are added to standard λ-calculi in order to use ideas from well-studied type systems (including the work mentioned above) when reasoning about pointer programs. The work in the just cited paper is still in progress and will be interesting to follow, also considering two of the authors' work on separation logic for higher-order programming languages [Birkedal etal. 2005].

The first attempt of a formal correctness proof of a garbage collector was published in [Dijkstra etal.], where the problem "was selected as one of the most challenging – and hopefully, most instructive! – problems". The proof given there is informal and merely gives an idea of how to obtain a formal proof. Other informal proofs were published in [Ben-Ari 1984] and [Pixley 1988]. The fact that a "mechanically verifiable proof would need all kinds of trivial invariants" was used to justify the informal approach. Russinoff [Russinoff 1994] explored how great a detail that was needed for a formal proof and demonstrated that the proofs in [Ben-Ari 1984] and [Pixley 1988] are fallacious. Wadler [Wadler 1976] gave an analysis and gave (semantic) proofs of complexity properties of a realtime garbage collection system, and moreover, there have been several formal verifications of correctness proofs of *abstract* versions of mark-and-sweep garbage collectors, using several different techniques [Russinoff 1994; Jackson 1998; Havelund 1999; Coupet-Grimal 2003]. Note that many of the garbage collectors mentioned in this paragraph are *concurrent* and as such, more complicated to prove. Recent progress [O'Hearn 2004] paves the way for verifying concurrent garbage collectors in separation logic. A comparison between logical frameworks for verifying proofs has been performed by Burdy [Burdy 2001].

In their work on a type preserving garbage collector, Wang and Appel [Wang and Appel 2001] transform well-typed programs into a form where they call a function, which acts as a garbage collector for the program. This function is designed such that it is well-typed in the target language, and thus is safe to execute. The

approach of Wang and Appel guarantees safety, but not correctness of the garbage collector, and there is no treatment of cyclic data structures, since the user language does not create cyclic data structures. Monnier and Shao [Monnier and Shao 2002] combine ideas from region calculi and alias types in their work on typed regions and propose a programming language with a type system expressive enough to type a garbage collector, which is type preserving, generational, and handles cyclic data structures. As mentioned in Section 9, it is on the schedule for future work to extend our reasoning principles to a complete runtime system, and not only a garbage collector. Fluet and Wang [Fluet and Wang 2004] have implemented a safe runtime system for Scheme in Cyclone [Jim etal. 2002], including a copying garbage collector which is also based on Cheney's algorithm. Since Cyclone does not allow address arithmetic, they use a linked list to keep track of their queue in the NEW-space, whereas our implementation language uses address arithmetic and exploits the contiguous space NEW to implement the "implicit queue" that is used in the breadth-first search of Cheney's algorithm. Further, the type system of Cyclone guarantees memory safety, whereas our proof implies the existence of an isomorphism between the heaps before and after execution of the garbage collector. Another implementation of a complete runtime system for Scheme is the VLISP project, an overview of which can be found in [Guttmann etal. 1992] and [Guttmann etal. 1994]. As part of that project, a proof that a garbage collector implemented on a Garbage Collected Stored Bytecode Machine, establishes a "state correspondence" between the states before and after execution is included in a technical report [Swarup etal. 1992]. However, this proof is carried out at a completely semantical level, using the operational semantics and the above-mentioned state correspondences, which are similar to simulation relations. As mentioned in Section 3.2, we simply assume that there is only one cell in the root set before we garbage collect, and we have thus glossed over the issues of correctly constructing and keeping track of the root set. In both the implementation from [Fluet and Wang 2004] and the VLISP project, it is shown that the root sets are correctly maintained.

Recently, there has been a lot of work on Proof Carrying Code [Necula and Lee 1996], [Necula 1997]. The basic idea of a code producer submitting a proof of safety along with a program could, of course, be transferred to low-level programming languages, like the one used with separation logic. Nipkow's research group in Munich has developed a framework for formally verifying programs in traditional Hoare logic with arrays [Mehta and Nipkow 2003], and an extension to separation logic is at its early stages [Weber 2004]. Also, Berdine *et al.* are working on Smallfoot [Berdine etal. 2005], a model checker for low-level programs that uses ideas from separation logic. Once more developed, these would allow one to verify correctness proof mechanically and perhaps to ship the proof of a garbage collector along with proofs of programs using the garbage collector.

9. CONCLUSION AND FUTURE WORK

We have specified and proved correct Cheney's copying garbage collector using local reasoning in an extension of standard separation logic. The specification and the proof are manageable because of local reasoning and we conclude that the idea of local reasoning scales well to such challenging algorithms.

We have extended separation logic with sets and relations, generalized the iterated separating conjunction and shown how these features can be used to specify naturally and prove correct an algorithm involving movement of cyclic data structures. We believe the methods used herein are of wider use and future work should include further experimentation with other subtle algorithms, such as those analyzed in [Bornat etal. 2004] (and also, Bornat's methods might be applicable to Cheney's algorithm).

One the goals of this paper was to prove the simple variant of Cheney's collector, but it is natural to ask whether the approach of this work scales to more complex systems where the collected data have more complex types or where the collector is of a different type than stop-and-copy. We do not have a proof of such a collector, but we believe that an extension of the methodology presented here will serve as a basis for proofs of such algorithms. For example, in a more complex type system, the definition of a heap morphism needs to be refined, and presumably this will induce new notions in the logic.

One could argue that it is a weakness of separation logic that we had to extend it with the above mentioned new constructs, since it would be worrying if one would have to extend the logic for every new major proof. However, as explained in the recent article [Biering etal. 2005], one can see these extensions as simple *definitional* extensions of *higher-order* separation logic. That is, there is a single logic in which one can define, e.g., the finite sets and relations and then prove in the logic (rather than semantically) that properties such as those in Appendix B hold.

Future work also includes studying how to specify and prove correct combinations of user level programs and runtime systems, as mentioned in the introduction. In his work on Foundational Proof Carrying Code [Appel 2001], Appel suggests compiling high-level languages into the Typed Assembly Language [Morrisett etal. 1999]. Our work offers an alternative to this. We suggest compiling types from high-level languages into *garbage insensitive predicates*, in the sense of [Calcagno etal. 2003], and using our memory allocator and garbage collector as an implementation of the `malloc` operation of TAL. By the nature of garbage insensitive predicates, we would have $\{P\}\ GC\ \{P\}$ for these predicates, for any correct garbage collector GC, and thus predicates resulting from type-safety guarantees would be preserved by the garbage collector, as desired.

Acknowledgments

The authors wish to thank Peter O'Hearn, Hongseok Yang, Richard Bornat, Cristiano Calcagno, Henning Niss, Martin Elsman, and Mads Tofte for insightful discussions. We also thank the anonymous referees for their useful and detailed comments.

REFERENCES

Aditya, S. and Caro, A. 1993. Compiler-directed type reconstruction for polymorphic languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, Copenhagen, Denmark, pp. 74–82. ACM Press.

Aditya, S., Flood, C. H., and Hicks, J. E. 1994. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP '94)*, Orlando, Florida, United States, pp. 12–23. ACM Press.

Ahmed, A., Jia, L., and Walker, D. 2003. Reasoning about hierarchical storage. In *Proc. of the Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS'03)*, Ottawa, Canada. IEEE Press.

Appel, A. W. 2001. Foundational proof carrying code. In *Proc. of the Sixteenth IEEE Symposium on Logic in Computer Science (LICS'01)*, Boston, MA, USA. IEEE Press.

Appel, A. W. and Gonçalves, M. J. R. 1993. Hash-consing garbage collection. Technical Report CS-TR-412-93 (February), Princeton University.

Ben-Ari, M. 1984. Algorithms for on-the-fly garbage collection. *ACM Transactions of Principles on Programming Languages and Systems (TOPLAS) 6,* 3, 333–344.

Berdine, J., Calcagno, C., and O'Hearn, P. W. 2005. Symbolic execution with separation logic. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05)*, Tsukuba, Japan, pp. 52–68. Springer Verlag.

Biering, B., Birkedal, L., and Torp-Smith, N. 2005. BI-hyperdoctrines and higher order separation logic. In *Proc. of ESOP 2005: The European Symposium on Programming*, Edinburgh, Scotland, pp. 233–247.

Birkedal, L., Torp-Smith, N., and Reynolds, J. 2004. Local reasoning about a copying garbage collector. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, Venice, Italy, pp. 220–231.

Birkedal, L., Torp-Smith, N., and Yang, H. 2005. Semantics of separation-logic typing and higher-order frame rules. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, Chicago, IL, USA, pp. 260–269. IEEE Press.

Bornat, R. 2000. Proving pointer programs in Hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction*, Volume 1837 of *Lecture Notes In Computer Science*, Ponte De Lima, Portugal, pp. 102–126. Springer Verlag.

Bornat, R. 2003. Correctness of copydag via local reasoning. Private Communication.

Bornat, R., Calcagno, C., and O'Hearn, P. 2004. Local reasoning, separation and aliasing. In *Proceedings of SPACE 2004*, Venice, Italy.

Burdy, L. 2001. B vs Coq to prove a garbage collector. In *Proc. of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, Volume 2152 of *LNCS*, Edinburgh, Scotland. Springer Verlag.

Calcagno, C., Ishtiaq, S., and O'Hearn, P. W. 2000. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *Proc. of the Second International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, Montreal, Canada.

Calcagno, C., O'Hearn, P. W., and Bornat, R. 2003. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science 298,* 3, 557–587.

Cheney, C. J. 1970. A nonrecursive list compacting algorithm. *Communications of the ACM 13,* 11 (November), 677–678.

Coupet-Grimal, S. 2003. C. nouvet. *Journal of Logic and Computation 13,* 6 (December), 815–833.

Crary, K., Walker, D., and Morrisett, G. 1999. Typed memory management in a calculus of capabilities. In *Proc. of the 26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL'99)*, San Antonio, TX, USA, pp. 262–275.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, G. S., and Steffens, E. M. F. On-the-fly garbage collection: an exercise in cooperation.

Fluet, M. and Wang, D. 2004. Implementation and performance evaluation of a safe runtime system in Cyclone. In *Informal Proceedings of the Second Workshop on Semantics, Program*

Analysis and Computing Environments for Memory Management (SPACE'04), Venice, Italy. ACM/SIGPLAN.

Goto, E. 1974. Monocopy and associative algorithms in extended lisp. Technical Report TR 74-03, University of Tokyo.

Gries, D. and Levin, G. 1980. Assignment and procedure call proof rules. *ACM Transactions on Programming Languages and Systems (TOPLAS) 2,* 4 (October), 564–579.

Guttmann, J., Ramsdell, J., and Wand, M. 1994. VLISP: A verified implementation of scheme. *Lisp and Symbolic Computation 8,* 1–2, 5–32.

Guttmann, J. D., Monk, L. G., Ramsdell, J. D., Farmer, W. M., and Swarup, V. 1992. A guide to VLISP, a verified programming language implementation. Technical Report M92B091, The MITRE Corporation.

Hallenberg, N., Elsman, M., and Tofte, M. 2002. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, Berlin. ACM Press.

Havelund, K. 1999. Mechanical verification of a garbage collector. In *Proc. of the Fourth International Workshop Formal Methods for Parallel Programming : Theory and Applications (FMPPTA'99)*, San Juan, Puerto Rico.

Hawblitzel, C., Wei, E., Huang, H., Krupski, E., and Wittie, L. 2004. Low-level linear memory management. In *Proceedings of SPACE 2004*, Venice, Italy.

Hoare, C. A. R. 1969. An axiomatic approach to computer programming. *Communications of the ACM 12,* 583, 576–580.

Jackson, P. B. 1998. Verifying a garbage collection algorithm. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, Volume 1479 of *LNCS*, Canberra, Australia, pp. 225–244. Springer Verlag.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. 2002. Cyclone: A safe dialect of C. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, USA, pp. 275–288.

Mehta, F. and Nipkow, T. 2003. Proving pointer programs in higher-order logic. In *Automated Deduction – CADE-19*.

Monnier, S. and Shao, Z. 2002. Typed regions. Technical Report YALEU/DCS/TR-1242, Dept. of Computer Science, Yale University, New Haven, CT.

Morrisett, G., Felleisen, M., and Harper, R. 1995. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, La Jolla, California, United States, pp. 66–77. ACM Press.

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21,* 3, 527–568.

Necula, G. C. 1997. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, pp. 106–119.

Necula, G. C. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, Berkeley, CA, USA, pp. 229–243.

O'Hearn, P. W. 2004. Resources, concurrency and local reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04)*, Volume 3170 of *LNCS*, London, England, pp. 49–67.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL 2001)*, Berlin, Germany.

Owicki, S. and Gries, D. 1976. An axiomatic proof technique for parallel programs. *Acta Informatica 6,* 4, 319–340.

Petersen, L., Harper, R., Crary, K., and Pfenning, F. 2003. A type theory for memory allocation and data layout. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, New Orleans, LA, USA, pp. 172–184.

Pixley, C. 1988. An incremental garbage collection algorithm for multimutator systems. *Distributed Computing 3,* 1, 41–50.

Pym, D. 2002. *The Semantics and Proof Theory of the Logic of Bunched Implications*, Volume 26 of *Applied Logics Series*. Kluwer.

Reynolds, J. C. 1981. *The Craft of Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall, London.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, pp. 55–74. IEEE Press.

Russinoff, D. M. 1994. A mechanically verified incremental garbage collector. *Formal Aspects of Computing 6*, 359–390.

Smith, F., Walker, D., and Morrisett, G. 2000. Alias types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP'00)*, Berlin, Germany, pp. 366–381.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. 2005. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, Long Beach, CA, USA, pp. 183–194. ACM Press.

Swarup, V., Farmer, W. M., Guttmann, J. D., Monk, L. G., and Ramsdell, J. D. 1992. The VLISP byte-code interpreter. Technical Report M 92B096, The MITRE Corporation.

Tofte, M. and Birkedal, L. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS) 20,* 4 (July), 734–767.

Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. 2004. A retrospective on region-based memory management. *Higher-Order Symbolic Computation 17,* 3 (September), 245–265.

Tofte, M. and Talpin, J.-P. 1994. Implementation of the call-by-value lambda-calculus using a stack of regions. In *Proc. of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, OR, USA, pp. 188–201.

Wadler, P. L. 1976. Analysis of an algorithm for real time garbage collection. *Communications of the ACM 19,* 9 (September), 491–500.

Wang, D. and Appel, A. W. 2001. Type preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London, United Kingdom, pp. 166–178.

Weber, T. 2004. Towards mechanized program verification with separation logic. In *Annual Conference of the European Association for Computer Science Logic (CSL'04)*, Karpacz, Poland.

Yang, H. 2001. Local reasoning for stateful programs. Ph. D. thesis, University of Illinois, Urbana-Champaign.

A. IMPLEMENTATION OF CHENEY'S ALGORITHM

```
alloc(l,n1,n2) {
  if (free < maxFree)
    [free] := n1;
    [free + 4] := n2;
    free := free + 8;
    l := free-8
  else
    if (offset = startOld) then
      offset := startNew;
      maxFree := endNew
    else
      offset := startOld;
      maxFree := endOld
    fi;
// Garbage Collection starts
    scan := offset;
    free := offset;
|   FWD := ∅;
|   UFWD := RCH;
|   φ := ∅;
    t1 := [root];
    t2 := [root + 4];
    [free] := t1;
    [free + 4] := t2;
    [root] := free;
|   FWD := FWD ∪ {root};
|   UFWD := UFWD \ {root};
|   φ := φ ∪ {(root, free)};
    free := free + 8;

    while ¬(scan = free)
    // ScanCar begins
      a := [scan];
      if (a mod 8 = 0)
        b := [a];
        if (b mod 8 = 0 ∧
            offset ≤ b ∧
            b ≤ maxFree)
          [scan] := b
        else
        // CopyCell* begins
          t1 := [a];
          t2 := [a + 4];
          [free] := t1;
          [free + 4] := t2;
          [a] := free;
          [scan] := free;
|         FWD := FWD ∪ {a};
|         UFWD := UFWD \ {a};
|         φ := φ ∪ {(a, free)};
          free := free + 8
        // CopyCell* ends
        fi;
      else skip
      fi;
    // ScanCar ends
    // ScanCdr begins
      a := [scan + 4];

      if (a mod 8 = 0)
        b := [a];
        if (b mod 8 = 0 ∧
            offset ≤ b ∧
            b ≤ maxFree)
          [scan + 4] := b
        else
          t1 := [a];
          t2 := [a + 4];
          [free] := t1;
          [free + 4] := t2;
          [a] := free;
          [scan + 4] := free;
|         FWD := FWD ∪ {a};
|         UFWD := UFWD \ {a};
|         φ := φ ∪ {(a, free)};
          free := free + 8
        fi;
      else skip
      fi;
    // ScanCdr ends
      scan := scan + 8
    od;
// Garbage Collection ends
    root := offset;
    alloc(l,n1,n2)
  fi
}
```

B. SET-THEORETIC RULES

Elementary rules

$$e \geq e' \rightarrow Itv(e, e') = \emptyset \tag{34}$$
$$e \in m \rightarrow (m \setminus \{e\}) \cup \{e\} = m \tag{35}$$
$$\neg(e \in m) \rightarrow (m \cup \{e\}) \setminus \{e\} = m \tag{36}$$
$$e \in (m \setminus \{e'\}) \rightarrow e \in m \tag{37}$$
$$\mathsf{Ptr}(e) \rightarrow e \in m \cup \{e\} \tag{38}$$
$$(e, e') \in r \setminus \{(e_1, e_2)\} \rightarrow (e, e') \in r \tag{39}$$
$$\mathsf{Ptr}(e) \wedge \mathsf{Ptr}(e') \rightarrow (e, e') \in r \cup \{(e, e')\} \tag{40}$$
$$\neg(e - 8 \in Itv(e, e')) \tag{41}$$
$$e \leq e' - 8 \ \wedge \mathsf{Ptr}(e') \rightarrow e' - 8 \in Itv(e, e') \tag{42}$$
$$e \leq e' \ \wedge \ e' \leq e'' - 8 \ \wedge \ \mathsf{Ptr}(e) \wedge \mathsf{Ptr}(e') \wedge \mathsf{Ptr}(e'') \rightarrow e' \in Itv(e, e'') \tag{43}$$
$$m_1 \subseteq m_2 \rightarrow \#m_1 \leq \#m_2 \tag{44}$$
$$e \in m \rightarrow \mathsf{Ptr}(e) \tag{45}$$
$$m_1 = Itv(e, e_1) \wedge m_2 = Itv(e, e_2) \wedge \#m_1 \leq \#m_2 \wedge \mathsf{Ptr}(e_1) \wedge \mathsf{Ptr}(e_2) \rightarrow e_1 \leq e_2 \tag{46}$$
$$m_1 \subseteq m_2 \wedge m_2 \subseteq m_1 \rightarrow m_1 = m_2 \tag{47}$$
$$\mathsf{iso}(r, m_1, m_2) \wedge (e, e') \in \rho \circ r \circ r^\dagger \rightarrow (e, e') \in \rho \tag{48}$$
$$e \bmod 8 = 0 \rightarrow \mathsf{Ptr}(e) \tag{49}$$
$$\mathsf{Ptr}(e) \rightarrow \mathsf{Ptr}(e - 8) \wedge \mathsf{Ptr}(e + 8) \tag{50}$$
$$(e, e') \in r \rightarrow (e', e) \in r^\dagger \tag{51}$$
$$(e, e') \in r_1 \wedge (e', e'') \in r_2 \rightarrow (e, e'') \in r_2 \circ r_1 \tag{52}$$
$$(e, e') \in (r_1 \circ r_2) \circ r_3 \longleftrightarrow (e, e') \in r_1 \circ (r_2 \circ r_3) \tag{53}$$
$$e \in Itv(e_1, e_2) \wedge e_2 \leq e_3 \rightarrow e \in Itv(e_1, e_3) \tag{54}$$
$$m = \emptyset \rightarrow \forall x.\ \neg(x \in m) \tag{55}$$
$$r = \emptyset \rightarrow \forall x, y.\ \neg((x, y) \in r) \tag{56}$$

Rules for iso:

$$\mathsf{iso}(\emptyset, \emptyset, \emptyset) \tag{57}$$

$$\mathsf{Ptr}(e_1) \wedge \mathsf{Ptr}(e_2) \wedge \neg(e_1 \in m_1) \ \wedge \ \neg(e_2 \in m_2) \ \wedge \ \mathsf{iso}(r, m_1, m_2) \rightarrow \mathsf{iso}(r \cup \{(e_1, e_2)\}, m_1 \cup \{e_1\}, m_2 \cup \{e_2\}) \tag{58}$$

$$(e_1 \in m_1) \wedge (e_2 \in m_2) \wedge (e_1, e_2) \in r \wedge \mathsf{iso}(r \setminus \{(e_1, e_2)\}, m_1 \setminus \{e_1\}, m_2 \setminus \{e_2\}) \rightarrow \mathsf{iso}(r, m_1, m_2) \tag{59}$$

$$\mathsf{iso}(r, m_1, m_2) \ \wedge \ (e_1, e_2) \in r \rightarrow e_1 \in m_1 \ \wedge \ e_2 \in m_2 \tag{60}$$

$$\mathsf{iso}(r, m_1, m_2) \rightarrow \mathsf{iso}(r^\dagger, m_2, m_1) \tag{61}$$

$$\mathsf{iso}(r, m_1, m_2) \ \wedge \ e \in m_1 \rightarrow \exists x.\ (e, x) \in r \ \wedge \ x \in m_2 \tag{62}$$

$$\mathsf{Tfun}(\rho, m_2) \ \wedge \ \mathsf{iso}(r, m_1, m_2) \rightarrow \mathsf{Tfun}(\rho \circ r, m_1) \tag{63}$$

$$\mathsf{iso}(r, m_1, m_2) \rightarrow \mathsf{Tfun}(r, m_1) \tag{64}$$

$$\mathsf{iso}(r, m_1, m_2) \rightarrow \#m_1 = \#m_2 \tag{65}$$

$$\mathsf{iso}(r, m_1, m_2) \wedge \neg(e \in m_1) \rightarrow \forall x^{\mathsf{int}}.\ \neg((e, x) \in r) \tag{66}$$

Rules for $\odot$

$$(e, e') \in \rho \ \wedge \ \neg\mathsf{Ptr}(e') \rightarrow \forall r^{\mathsf{frp}}.\ (e, e') \in r \odot \rho \tag{67}$$

$$(e, e') \in \rho \ \wedge \ \mathsf{Ptr}(e') \ \wedge \ (e', e'') \in r \rightarrow (e, e'') \in r \odot \rho \tag{68}$$

$$\mathsf{Tfun}(\rho, m) \wedge (e', e) \in \rho \wedge \mathsf{Ptr}(e) \wedge e' \in m \wedge (e', e'') \in r \odot \rho \rightarrow (e, e'') \in r \tag{69}$$

$$\begin{array}{l}(e, e') \in r \odot \rho \rightarrow \\ \ ((e, e') \in \rho \wedge \neg\mathsf{Ptr}(e')) \vee (\exists x.\ \mathsf{Ptr}(x) \wedge (e, x) \in \rho \wedge (x, e') \in r)\end{array} \tag{70}$$

$$\begin{array}{l}(e, e') \in r \odot \rho \wedge (e, e'') \in r \odot \rho \wedge \mathsf{Tfun}(r, m) \wedge \mathsf{Tfun}(\rho, m') \ \wedge \\ e \in m' \wedge ((\exists z.\ (e, z) \in \rho \wedge z \in m) \vee \neg\mathsf{Ptr}(e')) \rightarrow e' = e''\end{array} \tag{71}$$

$$(e, e') \in r \odot (\rho \circ \rho') \longleftrightarrow (e, e') \in (r \odot \rho) \circ \rho' \tag{72}$$

Rules for PtrRg

$$\mathsf{PtrRg}(r, m) \ \wedge \ \mathsf{Ptr}(e) \ \wedge \ (e', e) \in r \rightarrow e \in m \tag{73}$$

$$\mathsf{PtrRg}(r, m) \rightarrow \mathsf{PtrRg}(r \circ \rho, m) \tag{74}$$

Rules for $\subseteq$ and $\perp$

$$m \cup \emptyset = m \tag{75}$$

$$m_1 \cup m_2 = m_2 \cup m_1 \tag{76}$$

$$e \in m_1 \cup m_2 \rightarrow (e \in m_1 \vee e \in m_2) \tag{77}$$

$$e \in m_1 \cup m_2 \wedge \neg(e \in m_1) \rightarrow e \in m_2 \tag{78}$$

$$m_1 \cup m_2 = m \rightarrow m_1 \subseteq m \ \wedge \ m_2 \subseteq m \tag{79}$$

$$m_1 \perp m_2 \rightarrow \forall x.\ \neg(x \in m_1 \ \wedge \ x \in m_2) \tag{80}$$

$$\begin{array}{l}m_1 \cup m_2 = m \ \wedge \ e \in m_1 \rightarrow (m_1 \setminus \{e\}) \cup (m_2 \cup \{e\}) = m \\ (m_1 \setminus \{e\}) \cup (m_2 \cup \{e\}) = m \ \wedge \ e \in m_1 \rightarrow m_1 \cup m_2 = m\end{array} \tag{81}$$

$$(\forall x.\ (x \in m_1) \rightarrow (x \in m_2)) \longleftrightarrow m_1 \subseteq m_2 \tag{82}$$

$$m_1 \perp m_2 \wedge m_1' \subseteq m_1 \rightarrow m_1' \perp m_2 \tag{83}$$

C. FORMAL PROOF OF INIT*

We need to show that I follows from the conclusion of the derivation in Section 6.1.

I can be viewed as a conjunction $\mathsf{I} \equiv \mathsf{I}_1 \wedge \cdots \wedge \mathsf{I}_k$ of assertions, where some of the I_is are pure and one is not pure; let us say that I_k is the impure part of I, and write

I_k on the form $A_1 * \cdots * A_m$. Similarly, the conclusion in the derivation has the form $\mathsf{I}'_1 \wedge \cdots \wedge \mathsf{I}_{k'}$ where I'_i are pure for $i \in \{1, \cdots, k'-1\}$ and where $\mathsf{I}_{k'} \equiv A'_1 * \cdots * A'_m$.

We show that each of $\mathsf{I}_1, \ldots, \mathsf{I}_{k-1}$ follow from $\mathsf{I}'_1 \wedge \cdots \wedge \mathsf{I}'_{k'-1}$ and each of the A_i from $\mathsf{I}'_1 \wedge \cdots \wedge \mathsf{I}'_{k'-1} \wedge A'_i$; this is sufficient by Lemma 3.10. We start by proving the pure conjuncts of I from the conclusion in the derivation in Section 6.1.

—$\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY})$. This follows from

$$\mathsf{iso}(\varphi \setminus \{(\mathsf{root}, \mathsf{free}-8)\}, \mathsf{FWD} \setminus \{\mathsf{root}\}, \mathsf{BUSY} \setminus \{(\mathsf{free}-8)\}) \wedge \mathsf{root} \in \mathsf{FWD} \wedge (\mathsf{free}-8) \in \mathsf{BUSY} \wedge (\mathsf{root}, \mathsf{free}-8) \in \varphi \wedge \mathsf{Ptr}(\mathsf{free}-8),$$

(59), and (42).

—$\mathsf{RCH} = \mathsf{FWD} \cup \mathsf{UFWD}$ follows from

$$(\mathsf{FWD} \setminus \{\mathsf{root}\}) \cup (\mathsf{UFWD} \cup \{\mathsf{root}\}) = \mathsf{RCH}) \wedge \mathsf{root} \in \mathsf{FWD}$$

and (81).

—$\mathsf{I}_c \wedge (\mathsf{RCH} \perp \mathsf{NEW}) \wedge \#\mathsf{RCH} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FWD} \wedge \mathsf{offset} \leq \mathsf{scan}$ is part of the conclusion in Section 6.1.

—$\mathsf{scan} \leq \mathsf{free}$. Follows from $\mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} - 8 = \mathsf{offset}$.

—$\mathsf{Ptr}(\mathsf{scan}) \wedge \mathsf{Ptr}(\mathsf{free})$. Follows from (50) and $\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{Ptr}(\mathsf{free}-8)$.

—$(\mathsf{root}, \mathsf{offset}) \in \varphi$ follows from $(\mathsf{root}, \mathsf{free}-8) \in \varphi$ and $\mathsf{offset} = \mathsf{free} - 8$.

For the impure parts, the argument is a bit more complicated. We deal with each of the parts in the iterated separating conjunction $(\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UFIN}} * \mathsf{A}_{\mathsf{FREE}})$ separately.
For UFWD, we have

$$\begin{array}{l}
\neg(\mathsf{root} \in \mathsf{UFWD}) \wedge \\
(\forall_* x \in ((\mathsf{UFWD} \cup \{\mathsf{root}\}) \setminus \{\mathsf{root}\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x,y') \in \mathsf{tail} \wedge x+4 \mapsto y'))) \\
\Downarrow \\
\underline{((\mathsf{UFWD} \cup \{\mathsf{root}\}) \setminus \{\mathsf{root}\}) = \mathsf{UFWD}} \wedge \\
(\forall_* x \in ((\mathsf{UFWD} \cup \{\mathsf{root}\}) \setminus \{\mathsf{root}\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x,y') \in \mathsf{tail} \wedge x+4 \mapsto y'))) \\
\Downarrow \\
(\forall_* x \in \underline{\mathsf{UFWD}}.\ ((\exists y.\ (x,y) \in \mathsf{head} \wedge x \mapsto y) * (\exists y'.\ (x,y') \in \mathsf{tail} \wedge x+4 \mapsto y'))) \\
\| \\
\mathsf{A}_{\mathsf{UFWD}},
\end{array}$$

where the two implications follow from (36) and (10), respectively.

For FWD,

$$
\begin{array}{l}
(\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge \mathsf{root} \in \mathsf{FWD} \wedge \\
((\forall_* x \in (\mathsf{FWD} \setminus \{\mathsf{root}\}).\ (\exists y.\ (x, y) \in \varphi \setminus \{(\mathsf{root}, \mathsf{free} - 8)\} \wedge p \mapsto y, -))* \\
(\mathsf{root} \mapsto \mathsf{free} - 8, -)) \\
\Downarrow \\
\mathsf{root} \in \mathsf{FWD} \wedge \\
((\forall_* x \in (\mathsf{FWD} \setminus \{\mathsf{root}\}).\ (\exists y.\ (x, y) \in \underline{\varphi} \wedge x \mapsto y, -))* \\
\underline{(\mathsf{root} \mapsto \mathsf{free} - 8, - \wedge (\mathsf{root}, \mathsf{free} - 8) \in \varphi)}) \\
\Downarrow \\
\mathsf{root} \in \mathsf{FWD} \wedge \\
((\forall_* x \in (\mathsf{FWD} \setminus \{\mathsf{root}\}).\ (\exists y.\ (x, y) \in \varphi \wedge x \mapsto y, -))* \\
\underline{(\exists y.\ \mathsf{root} \mapsto y, - \wedge (\mathsf{root}, y) \in \varphi)}) \\
\Downarrow \\
\forall_* x \in \underline{\mathsf{FWD}}.\ (\exists y.\ (x, y) \in \varphi \wedge x \mapsto y, -) \\
\| \\
\mathsf{A}_{\mathsf{FWD}}
\end{array}
$$

The first implication follows from the rule for pure assertions in Remark 3.7, from (39), and Lemma 3.9. The third implication is an instance of (14).

For FIN, it is easiest to note that the condition $\mathsf{scan} = \mathsf{offset}$ in the conclusion of the derivation above makes both of the assertions about FIN equivalent to emp.

The following derivation takes care of FREE (we implicitly use (41)):

$$
\begin{array}{l}
\neg(\mathsf{free} - 8 \in \mathsf{FREE}) \wedge \\
\forall_* x \in ((\mathsf{FREE} \cup \{(\mathsf{free} - 8)\}) \setminus \{(\mathsf{free} - 8)\}.\ x.\ \rightarrow -, -) \\
\Downarrow \\
\underline{\mathsf{FREE} = (\mathsf{FREE} \cup \{(\mathsf{free} - 8)\}) \setminus \{(\mathsf{free} - 8)\}} \wedge \\
\forall_* x \in ((\mathsf{FREE} \cup \{(\mathsf{free} - 8)\}) \setminus \{(\mathsf{free} - 8)\}.\ x.\ \rightarrow -, -) \\
\Downarrow \\
\forall_* x \in \underline{\mathsf{FREE}}.\ x \mapsto -, - \\
\| \\
\mathsf{A}_{\mathsf{FREE}}
\end{array}
$$

The first implication here is an instance of (36), and the second implication follows from (10).

Finally, for UFIN, we use (42) and get

$$
\begin{array}{l}
(\mathsf{root}, \mathsf{free}-8) \in \varphi \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
\mathsf{Ptr}(\mathsf{free}-8) \wedge (\mathsf{free}-8) \in \mathsf{UFIN} \wedge \\
((\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{free}-8\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ (\varphi \setminus \{(\mathsf{root}, \mathsf{free}-8)\})^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ (\varphi \setminus \{(\mathsf{root}, \mathsf{free}-8)\})^\dagger) \wedge x+4 \mapsto y'))) * \\
(\mathsf{free}-8 \mapsto t_1, t_2)) \\
(1)\Downarrow \\
\underline{(\mathsf{free}-8, \mathsf{root}) \in \varphi^\dagger} \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
\mathsf{Ptr}(\mathsf{free}-8) \wedge (\mathsf{free}-8) \in \mathsf{UFIN} \wedge \\
((\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{free}-8\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \underline{\varphi^\dagger} \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \underline{\varphi^\dagger} \wedge x+4 \mapsto y'))) * \\
(\mathsf{free}-8 \mapsto t_1) * ((\mathsf{free}-8)+4 \mapsto t_2)) \\
(2)\Downarrow \\
\underline{(\mathsf{free}-8, t_1) \in \mathsf{head} \circ \varphi^\dagger} \wedge \underline{(\mathsf{free}-8, t_2) \in \mathsf{tail} \circ \varphi^\dagger} \wedge (\mathsf{free}-8) \in \mathsf{UFIN} \wedge \\
((\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{free}-8\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\mathsf{free}-8 \mapsto t_1) * ((\mathsf{free}-8)+4 \mapsto t_2)) \\
(3)\Downarrow \\
(\mathsf{free}-8) \in \mathsf{UFIN} \wedge \\
((\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{free}-8\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\mathsf{free}-8 \mapsto t_1 \wedge \underline{(\mathsf{free}-8, t_1) \in \mathsf{head} \circ \varphi^\dagger}) * \\
((\mathsf{free}-8)+4 \mapsto t_2 \wedge \underline{(\mathsf{free}-8, t_2) \in \mathsf{tail} \circ \varphi^\dagger})) \\
(4)\Downarrow \\
(\mathsf{free}-8) \in \mathsf{UFIN} \wedge \\
((\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{free}-8\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\underline{\exists y.\ \mathsf{free}-8 \mapsto y \wedge (\mathsf{free}-8, y) \in \mathsf{head} \circ \varphi^\dagger}) * \\
(\underline{\exists y'.\ (\mathsf{free}-8)+4 \mapsto y' \wedge (\mathsf{free}-8, y') \in \mathsf{tail} \circ \varphi^\dagger})) \\
(5)\Downarrow \\
\forall_* x \in \underline{\mathsf{UFIN}}.\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')) \\
\| \\
\mathsf{A}_{\mathsf{UFIN}}
\end{array}
$$

Here, the first implication uses (39), (51), and Lemma 3.9. The second follows from (52), and the third implication follows from purity. Finally, the last implication is an instance of the rule (14).

D. FORMAL PROOF OF LEMMA 6.3

The assertion $(\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(a) \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH})$ implies $a \in \mathsf{RCH}$ by (74) and (73), so A implies $a \in \mathsf{RCH}$. By (78),

$$a \in \mathsf{RCH} \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \wedge \neg(a \in \mathsf{UFWD}) \rightarrow a \in \mathsf{FWD},$$

so we assume $a \in \mathsf{UFWD}$ and derive a contradiction, i.e., we show $A \wedge (a \in \mathsf{UFWD}) \rightarrow \mathsf{F}$. By (16), it suffices to show (since F is a intuitionistic assertion)

$$\begin{array}{l}\mathsf{A}_{\mathsf{UFWD}} \wedge (a \hookrightarrow b) \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \\ \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \wedge a \in \mathsf{UFWD} \rightarrow \mathsf{F}\end{array}$$

We have

$$\begin{array}{ll} & \mathsf{A}_{\mathsf{UFWD}} \wedge a \hookrightarrow b \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \\ & \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \wedge a \in \mathsf{UFWD} \\ (1)\Downarrow & \\ & (\underline{\mathsf{A}_{\mathsf{UFWD}-a} * (\exists y.\ (a,y) \in \mathsf{head} \wedge a \mapsto y) * (\exists y'.\ (a,y') \in \mathsf{tail} \wedge a+4 \mapsto y')}) \wedge \\ & a \hookrightarrow b \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \\ (2)\Downarrow & \\ & (\mathsf{A}_{\mathsf{UFWD}-a} * (\underline{a+4 \mapsto -}) * (\exists y.\ (a,y) \in \mathsf{head} \wedge a \mapsto y)) \wedge \\ & a \hookrightarrow b \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \\ (3)\Downarrow & \\ & (\mathsf{A}_{\mathsf{UFWD}-a} * (a+4 \mapsto -) * (\underline{(a,b) \in \mathsf{head} \wedge a \mapsto b})) \wedge \\ & \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \\ (4)\Downarrow & \\ & (\mathsf{A}_{\mathsf{UFWD}-a} * (a+4 \mapsto -) * (a \mapsto b)) \wedge \\ & \underline{(a,b) \in \mathsf{head}} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \mathsf{Ptr}(b) \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \\ (5)\Downarrow & \\ & \underline{b \in \mathsf{RCH}} \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge b \in \mathsf{NEW} \\ (6)\Downarrow & \\ & \mathsf{F}\end{array}$$

The first of these implications follows from (14), the third follows from (15), the fourth from purity, the fifth from (73), and the last follows from (80). □

E. FORMAL PROOF OF LEMMA 6.6

First,

$$\begin{array}{l} \mathsf{I}_a \wedge \mathsf{Ptr}(a) \wedge (a \hookrightarrow b) \wedge \neg(b \in \mathsf{NEW}) \\ \Downarrow \\ (\mathsf{scan}, a) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{RCH}) \wedge \\ \mathsf{Ptr}(a) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \\ \Downarrow \\ \underline{a \in \mathsf{RCH}} \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \\ \Downarrow \\ a \in \mathsf{FWD} \vee a \in \mathsf{UFWD}\end{array}$$

The second implication follows from (73) and (74), and the third is by (77). So, as in the proof of Lemma 6.3, we assume $a \in \mathsf{FWD}$ and derive a contradiction, i.e., we show $(a \in \mathsf{FWD}) \wedge A \rightarrow \mathsf{F}$. By (16) and Lemma 6.1, the following derivation

establishes this.

$$
\begin{array}{ll}
 & \mathsf{A_{FWD}} \wedge a \hookrightarrow b \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \wedge a \in \mathsf{FWD} \\
(1) \Downarrow & \\
 & (\underline{\mathsf{A}_{\mathsf{FWD}-a}} * \underline{(\exists y.\ (a,y) \in \varphi \wedge a \mapsto y, -)}) \wedge \\
 & a \hookrightarrow b \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(2) \Downarrow & \\
 & (\mathsf{A}_{\mathsf{FWD}-a} * (\underline{a+4 \mapsto -}) * (\exists y.\ (a,y) \in \varphi \wedge a \mapsto y)) \wedge \\
 & a \hookrightarrow b \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(3) \Downarrow & \\
 & (\mathsf{A}_{\mathsf{FWD}-a} * (a+4 \mapsto -) * (\underline{(a,b) \in \varphi \wedge a \mapsto b})) \wedge \\
 & \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(4) \Downarrow & \\
 & (\mathsf{A}_{\mathsf{FWD}-a} * (a+4 \mapsto -) * (a \mapsto b)) \wedge \\
 & \underline{(a,b) \in \varphi} \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(5) \Downarrow & \\
 & (a,b) \in \varphi \wedge \mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(6) \Downarrow & \\
 & \underline{b \in \mathsf{BUSY}} \wedge \neg(b \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(7) \Downarrow & \\
 & b \in Itv(\mathsf{offset}, \mathsf{free}) \wedge \neg(b \in Itv(\mathsf{offset}, \mathsf{maxFree})) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
(8) \Downarrow & \\
 & \mathsf{F}
\end{array}
$$

The first of these implications follows from (14), the second is a matter of notation. The third implication is an instance of (15), and the next comes from purity. The sixth implication in the derivation follows from (60), and the last is by (54). This shows the lemma. □

F. LAST STEP OF SPECIFICATION FOR COPYCELL*

We need to show that I' follows from the conclusion of the specification for $\mathsf{CopyCell}^*$ from Section 6.2.3. As mentioned, this proof is similar to the proof in Section 6.1. In particular, the pure part of I' follows from the pure part of the conclusion above by the same argument as in Section 6.1, and the same argument goes for the separating conjunction over the sets FWD, UFWD, and FREE, and for the location $\mathsf{scan}+4$. Thus, if we let B be the pure part of the conclusion of the global specification in Section 6.2.3, what is left to show is that

$$
\begin{array}{l}
B \wedge (((\mathsf{free}-8) \mapsto t_1, t_2) * (\mathsf{scan} \mapsto \mathsf{free}-8) * \\
(\forall_* x \in ((\mathsf{UFIN} \setminus \{\mathsf{scan}\}) \setminus \{(\mathsf{free}-8)\}). \\
((\exists y.\ (x,y) \in \mathsf{head} \circ (\varphi \setminus \{(a, \mathsf{free}-8)\})^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ (\varphi \setminus \{(a, \mathsf{free}-8)\})^\dagger) \wedge x+4 \mapsto y')))
\end{array}
\tag{84}
$$

implies

$$
\begin{array}{l}
(\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{scan}\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\exists y.\ (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto y),
\end{array}
\tag{85}
$$

and that

$$
\begin{array}{l}
B \wedge \\
(\forall_* x \in \mathsf{FIN}. \\
((\exists y.\ (x,y) \in \varphi \setminus \{(a, \mathsf{free}-8)\} \odot (\mathsf{head} \circ (\varphi \setminus \{(a,\mathsf{free}-8)\})^\dagger) \wedge x \mapsto y) * \\
(\exists y'.\ (x,y') \in (\varphi \setminus \{(a,\mathsf{free}-8)\}) \odot (\mathsf{tail} \circ (\varphi \setminus \{(a,\mathsf{free}-8)\})^\dagger) \wedge x+4 \mapsto y')))
\end{array}
$$

implies

$$
\begin{array}{l}
B \wedge \\
(\forall_* x \in \mathsf{FIN}.\ ((\exists y.\ (x,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x+4 \mapsto y'))).
\end{array}
$$

The last of these follows from (39) and Lemma 3.9. For the first, we have

$$
\begin{array}{l}
(a,t_1) \in \mathsf{head} \wedge (a,t_2) \in \mathsf{tail} \wedge (a, \mathsf{free}-8) \in \varphi \wedge \\
(\mathsf{scan}, a) \in \mathsf{head} \circ (\varphi \setminus \{(a,\mathsf{free}-8)\})^\dagger \wedge \mathsf{Ptr}(a) \wedge \\
((\forall_* x \in ((\mathsf{UFIN} \setminus \{\mathsf{scan}\}) \setminus \{(\mathsf{free}-8)\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ (\varphi \setminus \{(a,\mathsf{free}-8)\})^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ (\varphi \setminus \{(a,\mathsf{free}-8)\})^\dagger) \wedge x+4 \mapsto y')) * \\
(\mathsf{scan} \mapsto \mathsf{free}-8) * (\mathsf{free}-8 \mapsto t_1, t_2)) \\
(1)\Downarrow \\
\underline{(\mathsf{free}-8, t_1) \in \mathsf{head} \circ \varphi^\dagger} \wedge \underline{(\mathsf{free}-8, t_2) \in \mathsf{tail} \circ \varphi^\dagger} \wedge \\
\underline{(\mathsf{scan}, \mathsf{free}-8) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)} \wedge \\
(\forall_* x \in ((\mathsf{UFIN} \setminus \{\mathsf{scan}\}) \setminus \{(\mathsf{free}-8)\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')) * \\
(\mathsf{scan} \mapsto \mathsf{free}-8) * ((\mathsf{free}-8) \mapsto t_1) * ((\mathsf{free}-8)+4 \mapsto t_2)) \\
(2)\Downarrow \\
\forall_* x \in ((\mathsf{UFIN} \setminus \{\mathsf{scan}\}) \setminus \{(\mathsf{free}-8)\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')) * \\
(\mathsf{scan} \mapsto \mathsf{free}-8 \wedge \underline{(\mathsf{scan}, \mathsf{free}-8) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger)}) * \\
((\mathsf{free}-8) \mapsto t_1 \ \wedge \underline{(\mathsf{free}-8, t_1) \in \mathsf{head} \circ \varphi^\dagger}) * \\
((\mathsf{free}-8)+4 \mapsto t_2 \wedge \underline{(\mathsf{free}-8, t_2) \in \mathsf{tail} \circ \varphi^\dagger}) \\
(3)\Downarrow \\
\forall_* x \in ((\mathsf{UFIN} \setminus \{\mathsf{scan}\}) \setminus \{\mathsf{free}-8\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')) * \\
\underline{(\exists y.\ \mathsf{scan} \mapsto y \wedge (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger))} * \\
\underline{(\exists y.\ (\mathsf{free}-8) \mapsto y \wedge (\mathsf{free}-8, y) \in \mathsf{head} \circ \varphi^\dagger)} * \\
\underline{(\exists y'.\ (\mathsf{free}-8)+4 \mapsto y' \wedge (\mathsf{free}-8, y') \in \mathsf{tail} \circ \varphi^\dagger)} \\
(4)\Downarrow \\
\forall_* x \in (\underline{\mathsf{UFIN} \setminus \{\mathsf{scan}\}}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')) * \\
(\exists y.\ \mathsf{scan} \mapsto y \wedge (\mathsf{scan}, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger))
\end{array}
$$

The first implication follows from (39), ordinary composition of relations (52), the rule (68) for the special relation composition $\odot$, and from Lemma 3.9. The second

implication follows from purity, and the last implication from (14). We have thus obtained (84) ⇒ (85).

G. FORMAL DERIVATION OF (33)

By the rule for assignment and obvious rules for intervals, we get

$$
\begin{array}{l}
\{A\} \\
\Downarrow \\
\left\{ \begin{array}{l}
\mathsf{I}_{pure} \wedge \mathsf{scan} \neq \mathsf{free} \wedge \\
((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\forall_* x \in \mathsf{FIN}.\ ((\exists y.\ (x,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x+4 \mapsto y'))) * \\
(\forall_* x \in (\mathsf{UFIN} \setminus \{\mathsf{scan}\}).\ ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\exists y.\ (\mathsf{scan},y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto y) * \\
(\exists y'.\ (\mathsf{scan},y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge (\mathsf{scan}+4) \mapsto y'))
\end{array} \right\} \\
\quad \mathsf{scan} := \mathsf{scan} + 8 \\
\left\{ \begin{array}{l}
\mathsf{I}_c \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge \#\mathsf{RCH} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FWD} \wedge \\
\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \\
\underline{\mathsf{scan}-8 \leq \mathsf{free}} \wedge \mathsf{offset} \leq \mathsf{scan}-8 \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \underline{\mathsf{Ptr}(\mathsf{scan}-8)} \wedge \\
((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\forall_* x \in (\underline{\mathsf{FIN} \setminus \{\mathsf{scan}-8\}}).\ ((\exists y.\ (x,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x+4 \mapsto y'))) * \\
(\forall_* x \in ((\underline{\mathsf{UFIN} \cup \{\mathsf{scan}-8\}}) \setminus \{\mathsf{scan}-8\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y'))) * \\
(\exists y.\ (\underline{\mathsf{scan}-8},y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge (\mathsf{scan}-8) \mapsto y) * \\
(\exists y'.\ (\underline{\mathsf{scan}-8},y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge ((\mathsf{scan}-8)+4) \mapsto y'))
\end{array} \right\} \\
\Downarrow \\
\left\{ \begin{array}{l}
\mathsf{I}_c \wedge \mathsf{RCH} \perp \mathsf{NEW} \wedge \#\mathsf{RCH} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FWD} \wedge \\
\mathsf{iso}(\varphi, \mathsf{FWD}, \mathsf{BUSY}) \wedge \mathsf{FWD} \cup \mathsf{UFWD} = \mathsf{RCH} \\
\mathsf{scan}-8 \leq \mathsf{free} \wedge \mathsf{offset} \leq \mathsf{scan}-8 \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}-8) \wedge \\
((\mathsf{A}_{\mathsf{UFWD}} * \mathsf{A}_{\mathsf{FWD}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\forall_* x \in (\mathsf{FIN} \setminus \{\mathsf{scan}-8\}).\ ((\exists y.\ (x,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x+4 \mapsto y'))) * \\
(\underline{\exists y.\ (\mathsf{scan}-8,y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge (\mathsf{scan}-8) \mapsto y}) * \\
(\underline{\exists y'.\ (\mathsf{scan}-8,y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge ((\mathsf{scan}-8)+4) \mapsto y'}) * \\
(\forall_* x \in ((\mathsf{UFIN} \cup \{\mathsf{scan}-8\}) \setminus \{\mathsf{scan}-8\}). \\
\quad ((\exists y.\ (x,y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\
\quad (\exists y'.\ (x,y') \in \mathsf{tail} \circ \varphi^\dagger \wedge x+4 \mapsto y')))
\end{array} \right\}
\end{array}
$$

The specification step uses the rule for assignment.

Like in Section 6.1 and 6.2.3, we now have to show that the pure part of I follows from the pure part I''_p of the conclusion I'' in the derivation above, and that the separating conjunction in I follows from that of I'' and I''_p. The only problem in the

pure part of I is to conclude

$$\mathsf{Ptr}(\mathsf{scan}) \wedge \mathsf{scan} \leq \mathsf{free} \wedge \mathsf{offset} \leq \mathsf{scan}$$

But this follows from

$$\mathsf{Ptr}(\mathsf{scan} - 8) \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{scan} - 8 \neq \mathsf{free} \wedge \mathsf{scan} - 8 \leq \mathsf{free} \wedge \mathsf{offset} \leq \mathsf{scan} - 8.$$

For the heap-dependent part of I, we see that $\mathsf{A}_{\mathsf{UFWD}}$, $\mathsf{A}_{\mathsf{FWD}}$, and $\mathsf{A}_{\mathsf{FREE}}$ follow directly from the corresponding parts of I''. So what is left to show is that

$$\begin{array}{l}\mathsf{I}''_p \wedge \\ (\forall_* x \in ((\mathsf{UFIN} \cup \{(\mathsf{scan} - 8)\}) \setminus \{\mathsf{scan} - 8\}).\ ((\exists y.\ (x, y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\ \quad (\exists y'.\ (x, y') \in \mathsf{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'))\end{array}$$

implies

$$\begin{array}{c}\forall_* x \in \mathsf{UFIN}.\ ((\exists y.\ (x, y) \in \mathsf{head} \circ \varphi^\dagger \wedge x \mapsto y) * \\ (\exists y'.\ (x, y') \in \mathsf{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'),\end{array}$$

and that

$$\begin{array}{l}\mathsf{I}''_p \wedge \\ ((\forall_* x \in (\mathsf{FIN} \setminus \{\mathsf{scan} - 8\}).\ ((\exists y.\ (x, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\ \quad (\exists y'.\ (x, y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y'))) * \\ (\exists y.\ (\mathsf{scan} - 8, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} - 8 \mapsto y) * \\ (\exists y'.\ (\mathsf{scan} - 8, y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge \mathsf{scan} - 8 \mapsto y'))\end{array}$$

implies

$$\begin{array}{c}\forall_* x \in \mathsf{FIN}.\ ((\exists y.\ (x, y) \in \varphi \odot (\mathsf{head} \circ \varphi^\dagger) \wedge x \mapsto y) * \\ (\exists y'.\ (x, y') \in \varphi \odot (\mathsf{tail} \circ \varphi^\dagger) \wedge x + 4 \mapsto y')).\end{array}$$

For the first of these, the implication follows from (10), Lemma 3.9, and (39), since $\neg((\mathsf{scan} - 8) \in \mathsf{UFIN})$ implies $(\mathsf{UFIN} \cup \{\mathsf{scan} - 8\}) \setminus \{\mathsf{scan} - 8\} = \mathsf{UFIN}$ (recall $\mathsf{UFIN} \equiv Itv(\mathsf{scan}, \mathsf{free})$). The second implication follows from Lemma 3.9, (39), and (14), since $\mathsf{scan} - 8 \in \mathsf{FIN} \equiv Itv(\mathsf{offset}, \mathsf{scan})$.