Correctness of a Garbage Collector via Local Reasoning

Lars Birkedal
Noah Torp-Smith
John C. Reynolds

IT University Technical Report Series **TR-2003-30**

ISSN 1600–6100 **July 2003**

ISSN 1600–6100

ISBN Get ISBN from BO

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone:	**+45 38 16 88 88**
Telefax:	**+45 38 16 88 99**
Web	**`www.itu.dk`**

Correctness of a Garbage Collector via Local Reasoning

Lars Birkedal
IT University of Copenhagen

Noah Torp-Smith
IT University of Copenhagen

John C. Reynolds
Carnegie Mellon University

Abstract

We give a programming language, model, and logic appropriate for implementing and reasoning about a memory management system. We then outline what is meant by correctness of a copying garbage collector, and employ a variant of the novel Separation Logics given in [ORY01], [Rey02c], to formally specify correctness. We then prove that our implementation meets its specification, using the logic we have given, and auxiliary variables [OG76].

Contents

List of Figures

1 Introduction and Motivation

Proving correctness of programs that explicitly manipulate imperative data structures has been a major difficulty, 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 [OP99], [Rey00], [Pym02]. In his thesis, Yang showed this to be a promising direction by giving an elegant proof of the non-trivial Schorr-Waite graph marking algorithm [Yan01]. This algorithm's complex pointer manipulations makes it a popular "benchmark test" for program logics, when one wants to demonstrate novel program proving techniques.

The aim of this paper is to use Separation Logic to prove correctness of a simple copying garbage collector. The interest in this is (at least) twofold. First, it provides another example of the benefits of Separation Logic, inasmusch as it provides a new example of a proof of a non-trivial algorithm. Secondly, copying garbage collectors are used in practical settings, for example in implementations of different functional programming languages [JL97], since the time-complexity of these algorithms is linear in the number of "live cells", which on average constitute only a small proportion (approximately 1.5 percent for Standard ML of New Jersey [App92]) of the total heap for these languages.

Another motivation is that there are questions in the literature might help answering. In [COB02], a garbage collected language is analyzed, and various results are presented. In particular, it is shown that a special treatment of the existential quantifier is needed in this setting. An underlying garbage collector is implicitly presumed in the operational semantics of the language, inasmuch as a *partial pruning* and an α*-renaming* of the current state is allowed at any time during execution of a program. Since it is not in the scope of that paper, it is not mentioned *how* this pruning / renaming is done, let alone proven that it is done correctly. A brief remark on the desirability of such a proof is mentioned in the end of the paper. In fact, this was the original motivation for taking on the challenge of proving a garbage collector correct.

The Typed Assembly Language (TAL) [MWCG99] is important for Foundational Proof Carrying Code (FPCC) [App01], since safety is guaranteed by the type system. In TAL, a `malloc` (but no `dealloc`) construct is part of the instruction set. This is a difference between TAL and other, more machine-like, assembly languages, where only "raw" pointer manipulations are allowed, and there are no constructs for allocating and deallocating memory, so that the programmer does not have unlimited memory resources. It is our belief that a formal proof of a garbage collector can be a part of the work needed to mimic the work of [MWCG99] in a more "realistic" setting.

The rest of the paper is organized as follows. In Section 2, we give an informal introduction to Separation Logic, and motivate its usefulness. In Section 3, we give the basic definitions and formalizations for the version of Separation Logic used in this exposition. In Section 4, we present the programming language used for implementation of the garbage collector, and give a Hoare-style program logic to connect the logic and the programming language. Section 5 gives general notions about garbage collection, including a correctness criterion and in Section 6, we present our implementation of a stop-and-copy garbage collector, and give a specification for the implementation. In Section 7, we formally prove the specification, and in Section 8, we show that the specification suffices to show correctness of the garbage collector. In Sections 9 and 10, we give an account of related and future work, and conclude.

Basic knowledge about Hoare Logic and semantics is a prerequisite for this paper.

2 An Introduction to Separation Logic

In this section, we will give a brief introduction to Separation Logic. It is strongly based on the expositions in [Rey02c] and [Rey02b]. We will not give formal definitions of the concepts used here, since it will clutter the presentation, and since we will extend Separation Logic with new notions later. For formal definitions, we refer to Sections 3 and 4.

Separation Logic is an extension of traditional Hoare Logic [Hoa69]. 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". Accordingly, the assertion language is extended with basic predicates concerning the heap, and two new connectives: the *separating conjunction* ($*$) and 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 don't go wrong."

for Separation Logic.

2.1 An Example

As mentioned in the Introduction, Separation Logic provides reasoning principles for proving programs that manipulate shared mutable data structures. We will demonstrate an advantage Separation Logic has compared to more "traditional" program logics by an example.

Suppose we represent the sequence $\alpha_0 = n_1, \ldots, n_k$ with a singly-linked list in the heap like in Fig. 1.

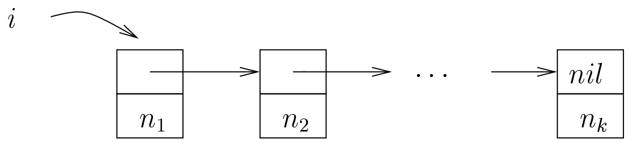

Figure 1: A list representation in the heap

If we want to reverse this list, we write the program

$$\mathsf{prev} := nil;\ \mathbf{while}\ i \neq nil\ \mathbf{do}\ \mathsf{next} := [i];\ [i] := \mathsf{prev};\ \mathsf{prev} := i;\ i := \mathsf{next}\ \mathbf{od},$$

where $[e]$ denotes the value stored at the address denoted by e in the heap.

At a point of execution, the heap looks like in Fig. 2.

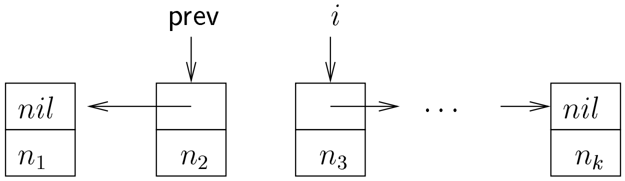

Figure 2: During list reversal

After execution of the program, we have the heap depicted in Fig. 3.

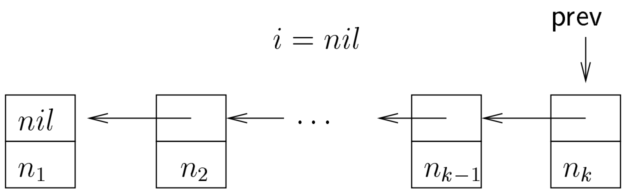

Figure 3: After list reversal

In order to prove the program correct, we must exhibit an invariant for the **while** loop. For this, it is illuminating to consider Fig. 2. We see that prev represents a list fragment β that has already been reversed, and that the list α represented by i has not been reversed. The reverse of α_0 is the reverse of α concatenated by β.

As a first try of an invariant, we might therefore define the predicate list by induction on the sequence α by

$$\begin{aligned}&\mathsf{list}(\varepsilon, i) \equiv i = nil\\ &\mathsf{list}((n \cdot \alpha), i) \equiv \exists j.\ i \hookrightarrow j, n \wedge \mathsf{list}(\alpha, j),\end{aligned}$$

and then claim that the following invariant is adequate:

$$\exists \alpha, \beta.\ \mathsf{list}(\alpha, i) \wedge \mathsf{list}(\beta, \mathsf{prev}) \wedge \alpha_0^{-1} = \alpha^{-1} \cdot \beta,$$

where α^{-1} is the reverse of the sequence α. The problem with this invariant is that it is not strong enough, since it does not prevent sharing between the lists represented by i and prev. For example, the assertion above might be satisfied by a heap that looks like that of Fig. 4.

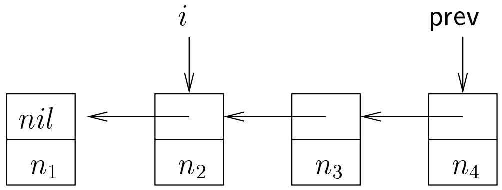

Figure 4: A counterexample

For this heap, however, the program will malfunction and create a list with a loop. To prevent sharing between the two lists, we might strengthen the invariant. First we can define the predicate $\mathsf{Reachable}_n$ by

$$\begin{aligned}
&\mathsf{Reachable}_0(i, j) \equiv i = j \\
&\mathsf{Reachable}_{n+1}(i, j) \equiv \exists a, k.\ i \hookrightarrow k, a \wedge \mathsf{Reachable}_n(k, j)
\end{aligned}$$

and then

$$\mathsf{Reachable}(i, j) \equiv \exists n \geq 0.\ \mathsf{Reachable}_n(i, j).$$

An adequate invariant is then

$$\begin{aligned}
&(\exists \alpha, \beta.\ \mathsf{list}(\alpha, i) \wedge \mathsf{list}(\beta, \mathsf{prev}) \wedge \alpha_0^{-1} = \alpha^{-1} \cdot \beta) \wedge \\
&(\forall k.\ \mathsf{Reachable}(i, k) \wedge \mathsf{Reachable}(\mathsf{prev}, k) \rightarrow k = nil).
\end{aligned}$$

As Reynolds points out, we might also want to make sure that another list γ, represented by x in the heap, is not modified by our program. Then we would have to make sure that all non-nil locations reachable from both i and prev are different from those reachable from x. The formulas involved quickly become large and hard to reason about. Separation Logic deals with this problem. Instead of something like

$$\begin{aligned}
&(\exists \alpha, \beta.\ \mathsf{list}(\alpha, i) \wedge \mathsf{list}(\beta, \mathsf{prev}) \wedge \alpha_0^{-1} = \alpha^{-1} \cdot \beta) \wedge \\
&(\forall k.\ \mathsf{Reachable}(i, k) \wedge \mathsf{Reachable}(\mathsf{prev}, k) \rightarrow k = nil) \wedge \\
&\mathsf{list}(\gamma, x) \wedge \\
&(\forall k.\ (\mathsf{Reachable}(x, k) \wedge (\mathsf{Reachable}(i, k) \vee \mathsf{Reachable}(\mathsf{prev}, k)) \rightarrow k = nil),
\end{aligned}$$

we can write the more succinct assertion

$$\exists \alpha, \beta.\ (\mathsf{list}(\alpha, i) * \mathsf{list}(\beta, \mathsf{prev}) * \mathsf{list}(\gamma, x)) \wedge \alpha_0^{-1} = \alpha^{-1} \cdot \beta$$

in Separation Logic. It precisely says that the lists α, β, γ are represented in *disjoint* parts of the heap.

2.2 The Frame Rule

In traditional Hoare logic (without shared data structures), one has the *rule of constancy*

$$\frac{\{A\}\ C\ \{A'\}}{\{A \wedge B\}\ C\ \{A' \wedge B\}}\ \text{Modifies}(C) \cap FV(B) = \varnothing$$

This rule has been useful, since it has allowed reasoning about only the "parts" (in a syntactic sense) of an assertion whose truth might be modified by the program fragment. In the presence of sharing, however, the rule of constancy is *unsound*. This can be seen by the following counterexample.

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

Here, x and y are both references to the same heap cell, and the problem is, of course, that if we change the value in this heap cell, the assertion about what y points to does not remain true. Separation Logic also has an answer to this problem.

Perhaps the most important rule in Separation Logic is the *Frame Rule*.

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

This rule allows us to do *local reasoning*. Suppose we have a program that mainly consists of a **while**-loop that does 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 only deals with a small portion of this structure. The Frame Rule allows us to prove the invariant by proving a specification which mentions only the parts of the heap that is actually manipulated in one loop iteration, and then conclude the specification regarding the full structure from this. We will see several applications of the Frame Rule rule in later sections.

2.3 A Brief History

An intuitionistic version of a logic with the notion of separating conjunction was discovered independently by Reynolds [Rey00] and O'Hearn and Ishtiaq [IO01]. Since the logic was an instance of the logic of Bunched Implications (**BI**), [OP99], [Pym02], O'Hearn and Ishtiaq could introduce the separating implication. O'Hearn and Ishtiaq also presented a classical version of the logic in their paper. A version of the logic which allowed for address arithmetic was later introduced by Reynolds [ORY01], and the Frame Rule described above was discovered by O'Hearn and was first presented in [IO01].

3 Syntax and Semantics

In this section, we present our basic storage model and the syntax and semantics of expressions and assertions. The basis of the system is the standard Separation Logic with pointer arithmetic [ORY01], but we take locations to be integers that are multiples of four, and we extend the term and assertion languages with finite sets and relations, new basic assertions about these, and an important new connective $\forall_*$ due to Reynolds.

3.1 Storage Model

We assume five countably infinite sets *Var*int, *Var*fs, *Var*frp, *Var*fri, *Var*path of variables, and we let *Var* be the disjoint union of these sets. We let metavariables

$$\begin{aligned}&x, y, p, q, \ldots \in \mathit{Var}\mathsf{int},\\ &m, m', \ldots \in \mathit{Var}\mathsf{fs},\\ &f, f' \ldots \in \mathit{Var}\mathsf{frp},\\ &g, g', \ldots \in \mathit{Var}\mathsf{fri},\ \text{and}\\ &P, P', \ldots \in \mathit{Var}\mathsf{path}\end{aligned}$$

range over each of these sets, and assume a type-function

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

indicating which type a given variable has. The set of *locations* is the set of natural numbers that are divisible by 4, and the set of *pointers* is the set of natural numbers that are divisible by 8. A *heap* is a finite, partial map from locations to integers, and finally, *stacks* are finite, partial maps from *Var* to the disjoint union of integers, finite sets of pointers, paths, finite relations on pointers, and finite relations between pointers and integers, where *paths* are finite words over the alphabet $\{\mathsf{head}, \mathsf{tail}\}$. More formally, we define:

Variables	$x, y, \ldots$	$\in$	*Var*
Pointers	Ptr	$\stackrel{\text{def}}{=}$	$\{8n \mid n \in \mathbb{N}\}$
Locations	Loc	$\stackrel{\text{def}}{=}$	$\{4n \mid n \in \mathbb{N}\}$
Finite sets of pointers	FS	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(\mathsf{Ptr})$
Finite rel'ns on pointers	FRP	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(\mathsf{Ptr} \times \mathsf{Ptr})$
Finite rel'ns with integers	FRi	$\stackrel{\text{def}}{=}$	$\mathcal{P}_{fin}(\mathsf{Ptr} \times \mathbb{Z})$
Paths	Path	$\stackrel{\text{def}}{=}$	$\{\mathsf{head}, \mathsf{tail}\}^*$
Values	Val	$\stackrel{\text{def}}{=}$	$\mathbb{Z} \cup \mathsf{FS} \cup \mathsf{FRP} \cup \mathsf{FRi} \cup \mathsf{Path}$
	Heaps	$\stackrel{\text{def}}{=}$	$\mathsf{Loc} \rightharpoonup_{fin} \mathbb{Z}$
	Stacks	$\stackrel{\text{def}}{=}$	$\{s : \mathit{Var} \rightharpoonup_{fin} \mathsf{Val} \mid \forall x \in \mathit{Var}.\ s(x) \in [\![\tau(x)]\!]\}$
	States	$\stackrel{\text{def}}{=}$	$\mathsf{Stacks} \times \mathsf{Heaps}$,

where $[\![\mathsf{int}]\!] = \mathbb{Z}$, $[\![\mathsf{fs}]\!] = \mathsf{FS}$, $[\![\mathsf{fri}]\!] = \mathsf{FRi}$ $[\![\mathsf{path}]\!] = \mathsf{Path}$, and $[\![\mathsf{frp}]\!] = \mathsf{FRP}$.

This constitutes our storage model.

3.2 Expressions

We define a syntax for expressions of each of the types $\mathsf{int}, \mathsf{fs}, \mathsf{frp}, \mathsf{fri}, \mathsf{path}$. We will not have expressions that involve the heap, but have manipulations involving the heap as parts of command forms in our programming language, which will be presented later. Expressions of type int are defined by the following grammar:

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

where $n \in \mathbb{Z}$ and $j \in \mathbb{N} \setminus \{0\}$. Although the superscript that indicates the type is only meant to indicate the type of variables, we will sometimes use a superscript to indicate the type of composite expressions. At other times, we will omit the superscripts, even on variables, if it causes no confusion.

Expressions of type fs are defined as follows

$$m^{\mathsf{fs}} \quad ::= \quad \varnothing^{\mathsf{fs}} \mid x^{\mathsf{fs}} \mid \{e^{\mathsf{int}}\} \mid m^{\mathsf{fs}} \oplus e^{\mathsf{int}} \mid m^{\mathsf{fs}} \ominus e^{\mathsf{int}} \mid \mathit{Itv}(e^{\mathsf{int}}, e^{\mathsf{int}}) \mid m^{\mathsf{fs}} \cup m^{\mathsf{fs}}$$

Expressions of type frp are defined by the grammar

$$f^{\mathsf{frp}} \quad ::= \quad \varnothing^{\mathsf{frp}} \mid x^{\mathsf{frp}} \mid f^{\mathsf{frp}} \oplus (e^{\mathsf{int}}, e^{\mathsf{int}}) \mid f_1^{\mathsf{frp}} \circ f_2^{\mathsf{frp}} \mid f^{\dagger}$$

The syntax for expressions of type fri is:

$$g^{\mathsf{fri}} \quad ::= \quad x^{\mathsf{fri}} \mid f^{\mathsf{frp}} \circ g^{\mathsf{fri}} \mid f^{\mathsf{frp}} \circledcirc g^{\mathsf{fri}}$$

Finally, expressions of type path are given by

$$P^{\mathsf{path}} \quad ::= \quad \varepsilon \mid P^{\mathsf{path}} \cdot \mathsf{head} \mid P^{\mathsf{path}} \cdot \mathsf{tail}$$

We now turn to the semantics of terms. The semantics of terms of type int is standard (except perhaps for terms of form $\# m^{\mathsf{fs}}$, whose semantics will be defined shortly), and we omit it. 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".

The semantics of terms of type fs is a finite set of pointers. $\oplus$ adds an element to a set, if it is a pointer, whereas $\ominus$ removes a pointer from a set. $Itv(e_1, e_2)$ is the set of pointers in the half-open interval from e_1 to e_2.

$$\begin{array}{rcl}
[\![\varnothing^{\mathsf{fs}}]\!]s &=& \varnothing \\
[\![x^{\mathsf{fs}}]\!]s &=& s(x) \\
[\![\{e\}]\!]s &=& \{[\![e]\!]s\} \cap Ptr \\
[\![m^{\mathsf{fs}} \oplus e^{\mathsf{int}}]\!]s &=& \begin{cases} [\![m]\!]s \cup \{[\![e]\!]s\} & \text{if } [\![e]\!]s \in Ptr \\ [\![m]\!]s & \text{otherwise} \end{cases} \\
[\![m^{\mathsf{fs}} \ominus e^{\mathsf{int}}]\!]s &=& [\![m]\!]s \setminus \{[\![e]\!]s\} \\
[\![Itv(e_1^{\mathsf{int}}, e_2^{\mathsf{int}})]\!]s &=& \{p \in Ptr \mid [\![e_1]\!]s \leq p \ \wedge \ p < [\![e_2]\!]s\} \\
[\![m_1^{\mathsf{fs}} \cup m_2^{\mathsf{fs}}]\!]s &=& [\![m_1]\!]s \cup [\![m_2]\!]s
\end{array}$$

With this, we can define the semantics of terms of the form $\#m^{\mathsf{fs}}$ (it is just the number of elements in the finite set denoted by m):

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

The semantics of terms of type frp is a finite set of pairs of pointers (or, equivalently, a finite relation on pointers).

$$\begin{array}{rcl}
[\![\varnothing^{\mathsf{frp}}]\!]s &=& \varnothing \\
[\![x^{\mathsf{frp}}]\!]s &=& s(x) \\
[\![f^{\mathsf{frp}} \oplus (e_1^{\mathsf{int}}, e_2^{\mathsf{int}})]\!]s &=& \begin{cases} [\![f]\!]s \cup \{([\![e_1]\!]s, [\![e_2]\!]s)\} & \text{if } ([\![e_1]\!]s, [\![e_2]\!]s) \in Ptr \times Ptr \\ [\![f]\!]s & \text{otherwise} \end{cases} \\
[\![f^{\dagger}]\!]s &=& \{(p', p) \mid (p, p') \in [\![f]\!]s\} \\
[\![f_1^{\mathsf{frp}} \circ f_2^{\mathsf{frp}}]\!]s &=& \{(p, p'') \mid \exists p'.\ (p, p') \in [\![f_2]\!]s \ \wedge \ (p', p'') \in [\![f_1]\!]s\}
\end{array}$$

The semantics for terms of type path is straightforward, in that the denotation of a term is equal to itself:

$$\begin{array}{rcl}
[\![\varepsilon]\!]s &=& \varepsilon, \text{ the empty word} \\
[\![P \cdot \mathsf{head}]\!]s &=& [\![P]\!]s \cdot \mathsf{head} \\
[\![P \cdot \mathsf{tail}]\!]s &=& [\![P]\!]s \cdot \mathsf{tail}
\end{array}$$

To conclude our semantics for terms, we give the semantics for terms of type fri. The $\circledcirc$ construct will be used to model the structure-preserving property of a garbage collector, (cf. [COB02], and Definition 5.2 later in this paper).

$$\begin{array}{rcl}
[\![x^{\mathsf{fri}}]\!]s &=& s(x) \\
[\![g^{\mathsf{fri}} \circ f^{\mathsf{frp}}]\!]s &=& \{(p, n) \mid \exists p' \in Ptr.\ (p, p') \in [\![f]\!]s \ \wedge (p', n) \in [\![g]\!]s\} \\
[\![f^{\mathsf{frp}} \circledcirc g^{\mathsf{fri}}]\!] &=& \{(p, n) \mid ((p, n) \in [\![g]\!]s \wedge n \notin Ptr) \vee \\
&& \quad (\exists p' \in Ptr.\ (p, p') \in [\![g]\!]s \wedge (p', n) \in [\![f]\!]s)\}
\end{array}$$

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

Lemma 3.1. *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', we have*

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

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

We will sometimes need to use equality between expressions. We will use $\equiv$ to mean syntactic equality between expressions, and we will sometimes write $e_1 = e_2$ to denote that $[\![e_1]\!]s = [\![e_2]\!]s$ for all stacks s.

3.3 Assertions

We define an assertion language and give a semantics for it. The core part of the language is the standard Separation Logic [Rey02c], but we have extended it with a number of basic assertions concerning finite sets and relations.

Let δ range over Types. The set of *assertions* is generated by the following grammar:

$$
\begin{array}{lcll}
A & ::= & e_1 \leq e_2 & \mid e_1 = e_2 \\
& & \mid \neg e & \mid \mathsf{T} \\
& & \mid \mathsf{F} & \mid \neg A \\
& & \mid A \to B & \mid A_1 \wedge A_2 \\
& & \mid A_1 \vee A_2 & \mid \forall x^{\delta}.\, A \\
& & \mid \exists x^{\delta}.\, A & \mid \mathsf{emp} \\
& & \mid e_1 \mapsto e_2 & \mid A_1 * A_2 \\
& & \mid A_1 -\!\!* A_2 & \mid \forall_* p^{\mathsf{int}} \in m^{\mathsf{fs}}.\, A \\
& & \mid e^{\mathsf{int}} \in m^{\mathsf{fs}} & \mid \mathsf{Disjoint}(m_1, m_2) \\
& & \mid \mathsf{Ptr}(e^{\mathsf{int}}) & \mid m_1^{\mathsf{fs}} = m_2^{\mathsf{fs}} \\
& & \mid \mathsf{PtrRg}(g^{\mathsf{fri}}, m^{\mathsf{fs}}) & \mid \mathsf{iso}(f^{\mathsf{frp}}, m_1^{\mathsf{fs}}, m_2^{\mathsf{fs}}) \\
& & \mid \mathsf{isUnion}(m_1^{\mathsf{fs}}, m_2^{\mathsf{fs}}, m^{\mathsf{fs}}) & \mid \mathsf{SbSet}(m_1, m_2) \\
& & \mid \mathsf{Tfun}(f^{\mathsf{frp}}, m^{\mathsf{fs}}) & \mid \mathsf{Tfun}(g^{\mathsf{fri}}, m^{\mathsf{fs}}) \\
& & \mid \mathsf{eval}(f_1^{\mathsf{frp}}, f_2^{\mathsf{frp}}, P^{\mathsf{path}}, e_1^{\mathsf{int}}, e_2^{\mathsf{int}}) & \mid \mathsf{Reachable}(f_1^{\mathsf{fri}}, f_2^{\mathsf{fri}}, m^{\mathsf{fs}}, e^{\mathsf{int}}) \\
& & \mid (e_1, e_2) \in g^{\mathsf{fri}} & \mid (e_1, e_2) \in f^{\mathsf{frp}}
\end{array}
$$

We will use the following standard shorthand notations

$$
\begin{array}{rcl}
p \mapsto e_1, e_2 & \stackrel{\text{def}}{=} & (p \mapsto e_1) * (p + 4 \mapsto e_2) \\
p \hookrightarrow e & \stackrel{\text{def}}{=} & p \mapsto e * \mathsf{T} \\
p \hookrightarrow e_1, e_2 & \stackrel{\text{def}}{=} & p \mapsto e_1, e_2 * \mathsf{T} \\
p \mapsto - & \stackrel{\text{def}}{=} & \exists x^{\mathsf{int}}.\, p \mapsto x \\
p \mapsto -, - & \stackrel{\text{def}}{=} & \exists x^{\mathsf{int}}, y^{\mathsf{int}}.\, p \mapsto x, y
\end{array}
$$

The notations $p \mapsto e_1, e_2, p \hookrightarrow e_1, e_2, p \mapsto -, -, p \hookrightarrow -, -$ make sense for all locations, but we shall only use them when p denotes a pointer.

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

The formal semantics for propositions is given by a judgement of the form

$$s, h \Vdash A,$$

the intended meaning of which is that the proposition A holds in the state s, h. We require $FV(A) \subseteq dom(s)$. We let b range over the boolean expressions $e_1 \leq e_2, e_1 = e_2, \neg e$, and δ ranges over Types. The semantics is given in Fig. 5. We have only given one of the clauses for each of the connectives Tfun and $\in$, the missing clauses are obvious.

A brief explanation is appropriate here. The assertion forms emp, $e_1 \mapsto e_2$, $A * B$, and $A -\!\!* B$ are taken from standard Separation Logic. 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 $A -\!\!* B$ 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. eval and $\mathsf{Reachable}$ concern evaluation of paths, and $\mathsf{Disjoint}$, iso, $\mathsf{isUnion}$, SbSet, and Tfun are "set theoretic" assertions. PtrRg says that any pointer which is a second component in any pair in the relation denoted g is in the set denoted by m. Finally, $\forall_*$ is an *iterated separating conjunction*. Informally, if $s, h \Vdash \forall_* p \in m.\, A$, and if $[\![m]\!]s = \{p_1, \ldots, p_k\}$, then h can be split into disjoint heaps $h = h_1 * \cdots * h_k$ with $s, h_1 \Vdash A[p_1/p], \ldots, s, h_k \Vdash A[p_k/p]$.

$$
\begin{array}{lcl}
s, h & \Vdash & b \quad \text{iff} \quad [\![b]\!]s \neq 0 \\
s, h & \Vdash & \mathsf{T} \quad \text{always} \\
s, h & \Vdash & \mathsf{F} \quad \text{never} \\
s, h & \Vdash & \neg A \quad \text{iff} \quad s, h \nVdash A \\
s, h & \Vdash & A \to B \quad \text{iff} \quad s, h \Vdash A \text{ implies } s, h \Vdash B \\
s, h & \Vdash & A \wedge B \quad \text{iff} \quad s, h \Vdash A \text{ and } s, h \Vdash B \\
s, h & \Vdash & A \vee B \quad \text{iff} \quad s, h \Vdash A \text{ or } s, h \Vdash B \\
s, h & \Vdash & \forall x^{\delta}.\, A \quad \text{iff} \quad \text{for all } v \in [\![\delta]\!], s[x \mapsto v], h \Vdash A \\
s, h & \Vdash & \exists x^{\delta}.\, A \quad \text{iff} \quad \text{there is } v \in [\![\delta]\!] \text{ such that } s[x \mapsto v], h \Vdash A \\
s, h & \Vdash & \mathsf{emp} \quad \text{iff} \quad \mathit{dom}(h) = \varnothing \\
s, h & \Vdash & e_1 \mapsto e_2 \quad \text{iff} \quad \mathit{dom}(h) = \{[\![e_1]\!]s\} \text{ and } h([\![e_1]\!]s) = [\![e_2]\!]s \\
s, h & \Vdash & A * B \text{ iff there exist heaps } h_1, h_2 \text{ such that} \\
 & & h_1 \# h_2,\ h_1 * h_2 = h, s, h_1 \Vdash A, \text{ and } s, h_2 \Vdash B \\
s, h & \Vdash & A \mathrel{-\!\!*} B \text{ iff} \\
 & & s, h * h' \Vdash B \text{ for all } h'. \text{ such that } h \# h' \text{ and } s, h' \Vdash A \\
s, h & \Vdash & \forall_* p \in m.\, A \text{ iff} \\
 & & \quad \begin{cases} s, h \Vdash A[p_1/p] * \cdots * A[p_k/p], & \text{if } [\![m]\!]s = \{p_1, \ldots, p_k\} \\ s, h \Vdash \mathsf{emp} & \text{if } [\![m]\!]s = \varnothing \end{cases} \\
s, h & \Vdash & e \in m \text{ iff} \\
 & & \quad [\![e]\!]s \in [\![m]\!]s \\
s, h & \Vdash & \mathsf{Disjoint}(m_1, m_2) \text{ iff } [\![m_1]\!]s \cap [\![m_2]\!]s = \varnothing \\
s, h & \Vdash & \mathsf{Ptr}(e) \text{ iff } [\![e]\!]s \bmod 8 = 0 \\
s, h & \Vdash & m_1 = m_2 \text{ iff } [\![m_1]\!]s = [\![m_2]\!]s \\
s, h & \Vdash & \mathsf{PtrRg}(g, m) \text{ iff} \\
 & & \quad \forall (p, q) \in [\![g]\!]s.\, q \in \mathit{Ptr} \Rightarrow q \in [\![m]\!]s \\
s, h & \Vdash & \mathsf{iso}(f, m_1, m_2) \text{ iff} \\
 & & \quad \forall p_1 \in M_1.\, \exists! p_2 \in M_2.\, (p_1, p_2) \in \varphi \wedge \\
 & & \quad \forall p_2 \in M_2.\, \exists! p_1 \in M_1.\, (p_1, p_2) \in \varphi \wedge \\
 & & \quad \forall (p_1, p_2) \in \varphi.\, p_1 \in M_1 \wedge p_2 \in M_2, \\
 & & \quad \text{where } M_1 = [\![m_1]\!]s, M_2 = [\![m_2]\!]s, \varphi = [\![f]\!]s \\
s, h & \Vdash & \mathsf{isUnion}(m_1, m_2, m) \text{ iff} \\
 & & \quad [\![m]\!]s = [\![m_1]\!]s \cup [\![m_2]\!]s \\
s, h & \Vdash & \mathsf{SbSet}(m_1, m_2) \text{ iff } [\![m_1]\!]s \subseteq [\![m_2]\!]s \\
s, h & \Vdash & \mathsf{Tfun}(f, m) \text{ iff} \\
 & & \quad \forall p \in [\![m]\!]s.\, \exists! n \in \mathbb{Z}.\, (p, n) \in [\![f]\!]s \\
s, h & \Vdash & \mathsf{eval}(f^{\mathsf{frp}}, g^{\mathsf{frp}}, P^{\mathsf{path}}, p^{\mathsf{int}}, e^{\mathsf{int}}) \text{ iff} \\
 & & \quad (P = \varepsilon \text{ and } s, h \Vdash p = e), \text{ or} \\
 & & \quad (P = P' \cdot \mathsf{head} \text{ and } \exists p' \in \mathit{Ptr}.\, s, h \Vdash \mathsf{eval}(f, g, P', p, p') \\
 & & \quad\quad \text{and } s, h \Vdash (p', e) \in f), \text{ or} \\
 & & \quad (P = P' \cdot \mathsf{tail} \text{ and } \exists p' \in \mathit{Ptr}.\, s, h \Vdash \mathsf{eval}(f, g, P', p, p') \\
 & & \quad\quad \text{and } s, h \Vdash (p', e) \in g). \\
s, h & \Vdash & \mathsf{Reachable}(f_1, f_2, m^{\mathsf{fs}}, e^{\mathsf{int}}) \text{ iff} \\
 & & [\![m]\!]s = \{p \in \mathit{Ptr} \mid \exists P \in \mathsf{Path}.\, s, h \Vdash \mathsf{eval}(f_1, f_2, P, e, p)\} \\
s, h & \Vdash & (e_1, e_2) \in f \text{ iff } ([\![e_1]\!]s, [\![e_2]\!]s) \in [\![f]\!]s
\end{array}
$$

Figure 5: Semantics of Assertions

In Fig. 5, we have used the notation $h_1 \# h_2$ to indicate $dom(h_1) \cap dom(h_2) = \varnothing$ (we call such heaps *disjoint*), and if $h_1 \# h_2$, we can define the combined heap $h_1 * h_2$ by

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

Note that the semantics is classical for the standard first-order logic fragment. As for the expressions, we have a standard substitution lemma for assertions.

Lemma 3.2. *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,*

$$s, h \Vdash A[e/x] \quad \textit{iff} \quad s[x \mapsto [\![e]\!]s], h \Vdash A$$

PROOF: The proof is a standard induction on the structure of assertions. Readers who wish to verify this will find it useful to show that if e' is an expression of type δ and if s, h is a state such that $[\![p]\!]s \in \mathsf{Ptr}, [\![e]\!]s \in \mathbb{Z}, [\![f_1]\!]s \in \mathsf{FRi}$, $[\![f_2]\!]s \in \mathsf{FRi}$, then for all $P \in \mathsf{Path}$,

$$s, h \Vdash \mathsf{eval}(f_1, f_2, P, p, e)[e'/x] \quad \text{iff} \quad s[x \mapsto [\![e']\!]s], h \Vdash \mathsf{eval}(f_1, f_2, P, p, e).$$

This is verified by induction on paths. □

Definition 3.3. For later use, we introduce some special classes of assertions. The definitions are taken from [Yan01] and [Rey02a].

- We call an assertion A *pure* if its validity does not depend on the heap, i.e., if $s, h \Vdash A$ if and only if $s, h' \Vdash A$, for all stacks s and heaps h, h'.
- We call an assertion A *monotone* if, for all stacks s and heaps h, h',

  $$s, h \Vdash A \text{ and } h \subseteq h' \text{ imply } s, h' \Vdash A.$$

  Here, $\subseteq$ is just set-theoretic inclusion of graphs.
- Assertion A is *domain-exact*, if $s, h \Vdash A$ and $s, h' \Vdash A$ imply $dom(h) = dom(h')$.
- Assertion A is *strictly exact*, if $s, h \Vdash A$ and $s, h' \Vdash A$ imply $h = h'$.

Remark 3.4.

- For a pure assertion A, $\wedge$ distributes over $*$:

  $$s, h \Vdash A \wedge (B * C) \quad \text{iff} \quad s, h \Vdash (A \wedge B) * (A \wedge C)$$

  for any assertions B, C.
- Pure assertions are monotone, and strictly exact assertions are domain-exact.
- Syntactically, an assertion is pure, if it does not contain any occurrences of emp, $\forall_*$, and $\mapsto$. Assertions made up by emp, $e_1 \mapsto e_2$, and $*$ are strictly exact.
- If $A(t)$ is pure, then $\exists t.\ (p \mapsto t\ \wedge A)$ is domain-exact.

3.4 Some Useful Rules

Definition 3.5. We call an assertion A *valid* if, for all states s, h with $FV(A) \subseteq dom(s)$, $s, h \Vdash A$.

We present a set of rules that are valid with respect to Definition 3.5. They will be needed in proofs later, and we believe that these rules 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.

Here are the rules:

Rules for $\forall_*$:

$$(\forall_* p \in m.\ A(p)) \ \wedge\ m = m' \rightarrow \forall_* p \in m'.A(p) \tag{1}$$

$$m = \varnothing \rightarrow ((\forall_* p \in m.A) \leftrightarrow \mathsf{emp}) \tag{2}$$

$$\begin{aligned}&(\forall_* p \in m.\ p \mapsto -\ \wedge\ A(p)) \wedge x \in m \rightarrow \\ &(\forall_* p \in m.\ p \mapsto -\ \wedge\ A(p)) \wedge (x \hookrightarrow -)\end{aligned} \tag{3}$$

When and A is precise,

$$(\forall_* p \in m.\ A(p)) * (\forall_* p \in m'.\ A(p)) \leftrightarrow (\forall_* p \in m \cup m'.\ A(p)) \tag{4}$$

Note that if m'' is the disjoint union of m, m', we do not need A to be precise in (4). As a special case, we get

$$\begin{aligned}&(\forall_* p \in m.\ A(p)) \wedge x \in m \rightarrow \\ &(\forall_* p \in (m \ominus x).\ A(p)) * A[x/p].\end{aligned} \tag{5}$$

Another consequence of (4) is

$$(x \in m) \wedge ((\forall_* p \in (m \ominus x).A) * A[x/p]) \rightarrow \forall_* p \in m.\ A \tag{6}$$

Rules for iso:

$$\mathsf{iso}(\varnothing, \varnothing, \varnothing) \tag{7}$$

$$\begin{aligned}&\mathsf{Ptr}(x_1) \wedge \mathsf{Ptr}(x_2) \wedge \neg(x_1 \in m_1)\ \wedge\ \neg(x_2 \in m_2)\ \wedge\ \mathsf{iso}(f, m_1, m_2) \rightarrow \\ &\mathsf{iso}(f \oplus (x_1, x_2), m_1 \oplus x_1, m_2 \oplus x_2)\end{aligned} \tag{8}$$

$$\begin{aligned}&(x_1 \in m_1) \wedge (x_2 \in m_2) \wedge (x_1, x_2) \in f \wedge \\ &\mathsf{iso}(f \ominus (x_1, x_2), m_1 \ominus x_1, m_2 \ominus x_2) \rightarrow \mathsf{iso}(f, m_1, m_2)\end{aligned} \tag{9}$$

$$\mathsf{iso}(f, m_1, m_2)\ \wedge\ (x_1, x_2) \in f \rightarrow x_1 \in m_1\ \wedge\ x_2 \in m_2 \tag{10}$$

$$\mathsf{iso}(f, m_1, m_2) \rightarrow \mathsf{iso}(f^\dagger, m_2, m_1) \tag{11}$$

$$\mathsf{iso}(f, m_1, m_2)\ \wedge\ p \in m_1 \rightarrow \exists q.\ (p, q) \in f\ \wedge\ q \in m_2 \tag{12}$$

$$\mathsf{Tfun}(g, m_2)\ \wedge\ \mathsf{iso}(f, m_1, m_2) \rightarrow \mathsf{Tfun}(g \circ f, m_1) \tag{13}$$

$$\mathsf{iso}(f, m_1, m_2) \rightarrow \mathsf{Tfun}(f, m_1) \tag{14}$$

$$\mathsf{iso}(f, m_1, m_2) \rightarrow \#m_1 = \#m_2 \tag{15}$$

$$\mathsf{iso}(f, m_1, m_2) \wedge \neg(x \in m_1) \rightarrow \forall e^{\mathsf{int}}.\ \neg((x, e) \in f) \tag{16}$$

Rules for $\circledcirc$

$$(x, y) \in g\ \wedge\ \neg\mathsf{Ptr}(y) \rightarrow \forall f^{\mathsf{frp}}.\ (x, y) \in f \circledcirc g \tag{17}$$

$$(x, y) \in g\ \wedge\ \mathsf{Ptr}(y)\ \wedge\ (y, z) \in f \rightarrow (x, z) \in f \circledcirc g \tag{18}$$

$$\mathsf{Tfun}(g, m) \wedge (q, p) \in g \wedge \mathsf{Ptr}(p) \wedge (q, r) \in f \circledcirc g \rightarrow (p, r) \in f \tag{19}$$

$$\begin{aligned}&(x, y) \in f \circledcirc g \rightarrow \\ &((x, y) \in g \wedge \neg\mathsf{Ptr}(y)) \vee (\exists p.\ \mathsf{Ptr}(p) \wedge (x, p) \in g \wedge (p, y) \in f)\end{aligned} \tag{20}$$

$$(x, y) \in f \circledcirc g \wedge (x, y') \in f \circledcirc g \wedge \mathsf{Tfun}(f, m) \wedge \mathsf{Tfun}(g, m') \rightarrow y = y' \tag{21}$$

Rules for isUnion, Disjoint, and SbSet

$$\mathsf{isUnion}(m_1, m_2, m) \wedge x \in m \to (x \in m_1 \vee x \in m_2) \tag{22}$$

$$\mathsf{isUnion}(m_1, m_2, m) \ \wedge \ x \in m \ \wedge \neg(x \in m_1) \to x \in m_2 \tag{23}$$

$$\mathsf{isUnion}(\varnothing, \varnothing, \varnothing) \tag{24}$$

$$\mathsf{isUnion}(m, \varnothing, m) \tag{25}$$

$$\mathsf{isUnion}(m_1, m_2, m) \to \mathsf{isUnion}(m_2, m_1, m) \tag{26}$$

$$\mathsf{isUnion}(m_1, m_2, m) \to \mathsf{SbSet}(m_1, m) \ \wedge \ \mathsf{SbSet}(m_2, m) \tag{27}$$

$$\mathsf{Disjoint}(m_1, m_2) \to \forall x.\ \neg(x \in m_1 \ \wedge \ x \in m_2) \tag{28}$$

$$\begin{array}{l} \mathsf{isUnion}(m_1, m_2, m) \ \wedge \ x \in m_1 \to \mathsf{isUnion}(m_1 \ominus x, m_2 \oplus x, m) \\ \mathsf{isUnion}(m_1 \ominus x, m_2 \oplus x, m) \ \wedge \ x \in m_1 \to \mathsf{isUnion}(m_1, m_2, m) \end{array} \tag{29}$$

$$(\forall p.\ (p \in m_1) \to (p \in m_2)) \leftrightarrow \mathsf{SbSet}(m_1, m_2) \tag{30}$$

$$\mathsf{Disjoint}(m_1, m_2) \wedge \mathsf{SbSet}(m_1', m_1) \to \mathsf{Disjoint}(m_1', m_2) \tag{31}$$

Rules for eval and Reachable

$$\mathsf{eval}(f, g, \varepsilon, p, q) \to p = q \tag{32}$$

$$\mathsf{eval}(f, g, P \cdot \mathsf{head}, p, q) \to \exists r.\ \mathsf{Ptr}(r) \wedge \ (r, q) \in f \ \wedge \ \mathsf{eval}(f, g, P, p, r) \tag{33}$$

$$\mathsf{eval}(f, g, P \cdot \mathsf{tail}, p, q) \to \exists r.\ \mathsf{Ptr}(r) \wedge \ (r, q) \in g \ \wedge \ \mathsf{eval}(f, g, P, p, r) \tag{34}$$

$$\mathsf{Reachable}(p, m, f, g) \wedge p' \in m \to \exists P^{\mathsf{path}}.\ \mathsf{eval}(f, g, P, p, p') \tag{35}$$

An induction principle in harmony with the inductive definition of paths:

$$\begin{array}{l} \forall f^{\mathsf{fri}}, g^{\mathsf{fri}}, p^{\mathsf{int}}. \\ \quad ((\forall p'^{\mathsf{int}}.\ \mathsf{eval}(f, g, \varepsilon, p, p') \to A(p')) \wedge \\ \quad (((\forall P^{\mathsf{path}}.\ \forall p'^{\mathsf{int}}.\ \mathsf{eval}(f, g, P, p, p')) \to A(p')) \to \\ \quad\ (((\forall p''^{\mathsf{int}}.\ \mathsf{eval}(f, g, P \cdot \mathsf{head}, p, p'')) \to A(p'')) \wedge \\ \quad\ ((\forall p'''^{\mathsf{int}}.\ \mathsf{eval}(f, g, P \cdot \mathsf{tail}, p, p''')) \to A(p''')))))) \to \\ \quad ((\forall P^{\mathsf{path}}, q^{\mathsf{int}}.\ \mathsf{eval}(f, g, P, p, q)) \to A(q)) \end{array} \tag{36}$$

General / Structural rules

$$A * (\exists t.\ B) \to \exists t.\ (A * B) \quad \text{when } t \notin FV(A) \tag{37}$$

When A is pure,

$$\begin{array}{l} (p \hookrightarrow x) \ \wedge \ ((\exists t.\ p \mapsto t \ \wedge A(t)) * B) \to \\ (p \mapsto x \ \wedge A[x/t])) * B \end{array} \tag{38}$$

If P is pure and P' is monotone,

$$A \wedge P \to P' \Rightarrow (A * B) \wedge P \to P' \tag{39}$$

Rules for PtrRg

$$\mathsf{PtrRg}(f, m) \ \wedge \ \mathsf{Ptr}(x) \ \wedge \ (p, x) \in f \to x \in m \tag{40}$$

$$\mathsf{PtrRg}(f, m) \to \mathsf{PtrRg}(f \circ g, m) \tag{41}$$

Rules for elementary set-manipulation

$$x \geq y \to Itv(x, y) = \varnothing \tag{42}$$
$$p \in m \ominus q \to \neg(p = q) \tag{43}$$
$$x \in m \to (m \ominus x) \oplus x = m \tag{44}$$
$$\neg(x \in m) \to (m \oplus x) \ominus x = m \tag{45}$$
$$x \in (m \ominus y) \to x \in m \tag{46}$$
$$(p, q) \in f \ominus (x, y) \to (p, q) \in f \tag{47}$$
$$\neg(p - 8 \in Itv(p, q)) \tag{48}$$
$$p \leq q - 8 \ \wedge \mathsf{Ptr}(q) \to q - 8 \in Itv(p, q) \tag{49}$$
$$p \leq q \ \wedge \ q \leq p' - 8 \ \wedge \ \mathsf{Ptr}(q) \wedge \mathsf{Ptr}(p) \wedge \mathsf{Ptr}(p') \to q \in Itv(p, p') \tag{50}$$
$$\mathsf{SbSet}(m_1, m_2) \to \#m_1 \leq \#m_2 \tag{51}$$
$$x \in m \to \mathsf{Ptr}(x) \tag{52}$$
$$m_1 = Itv(x, x_1) \wedge m_2 = Itv(x, x_2) \wedge \#m_1 \leq \#m_2 \wedge \mathsf{Ptr}(x_1) \wedge \mathsf{Ptr}(x_2) \to x_1 \leq x_2 \tag{53}$$
$$\mathsf{SbSet}(m_1, m_2) \wedge \mathsf{SbSet}(m_2, m_1) \to m_1 = m_2 \tag{54}$$
$$\mathsf{iso}(f, m_1, m_2) \wedge (x, x') \in g \circ f \circ f^{\dagger} \to (x, x') \in g \tag{55}$$
$$x \bmod 8 = 0 \to \mathsf{Ptr}(x) \tag{56}$$
$$\mathsf{Ptr}(x) \to \mathsf{Ptr}(x - 8) \wedge \mathsf{Ptr}(x + 8) \tag{57}$$
$$(x, y) \in f^{\mathsf{fri}} \to (y, x) \in f^{\dagger} \tag{58}$$
$$(x, y) \in f_1 \wedge (y, z) \in f_2 \to (x, z) \in g \circ f \tag{59}$$
$$y \in Itv(x_1, x_2) \wedge x_2 \leq x_3 \to y \in Itv(x_1, x_3) \tag{60}$$
$$m = \varnothing \to \forall x.\, \neg(x \in m) \tag{61}$$
$$f = \varnothing \to \forall x, y.\, \neg((x, y) \in f) \tag{62}$$
$$\tag{63}$$

Rule for Single-Valuedness

$$x \hookrightarrow y \ \wedge \ x \hookrightarrow y' \to y = y' \tag{64}$$

We have the expected result:

Theorem 3.6. *The rules (1) - (64) are all valid.*

The following lemma will be useful when we reason about assertions involving $\forall_*$.

Lemma 3.7. *Suppose* $s, h \Vdash \forall_* p \in m.\, A$ *and that* $\forall p'.\, p' \in m \wedge A[p'/p] \to B[p'/p]$ *is valid. Then* $s, h \Vdash \forall_* p \in m.\, B$.

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

$$s, h \Vdash \forall_* p \in m.\, A \iff s, h \Vdash \mathsf{emp} \iff s, h \Vdash \forall_* p \in m.\, B$$

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

$$s, h \Vdash A[p_1/p] * \cdots * A[p_k/p],$$

so there are pairwise disjoint heaps $h_1, \ldots, h_k$ with $\bigcup_{i=1\ldots k} h_i = h$ such that for $i = 1 \ldots k$, $s, h_i \Vdash A[p_i/p]$. Since we have $s, h_i \Vdash p_i \in m$, we have $s, h_i \Vdash A[p_i/p] \wedge p_i \in m$, so by assumption, we get $s, h_i \Vdash B[p_i/p]$, and this means

$$s, h \Vdash \forall_* p \in m.\, B,$$

as desired. □

This means that to infer $\forall_* p \in m.\ B$ from $\forall_* p \in m.\ A$, it suffices to show that $\forall p'.\ p' \in m \wedge A[p'/p] \to B[p'/p]$ is valid. In this way, we can do "implication under $\forall_*$".

Another useful rule comes from the following lemma for which we omit the proof.

Lemma 3.8. *If P is a pure assertion, and if $P \wedge A \to A'$ and $P \wedge B \to B'$ are valid, then $P \wedge (A * B) \to P \wedge (A' * B')$ is valid.*

By induction, this means that in order to infer $P \wedge (A_1 * \cdots * A_k)$ from $P' \wedge (A'_1 * \cdots * A'_k)$, it suffices to show $P' \to P$ and

$$P \wedge A'_1 \to A_1, \text{ and } \cdots \text{ ,and } P \wedge A'_k \to A_k.$$

As an example of a rule that can be derived from the rules above, we get the following from (38) and purity. Here, A must be a pure assertion.

$$\begin{array}{l} (p \hookrightarrow x) \;\wedge\; ((\exists t.\ (p \mapsto t \;\wedge A(t))) * B) \\ \Downarrow \\ (p \mapsto x \;\wedge A[x/t]) * B \\ \Downarrow \\ ((p \mapsto x) * B) \;\wedge A[x/t] \\ \Downarrow \\ A[x/t] \end{array} \tag{65}$$

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

4 The Implementation Language

In this section we first define the syntax and semantics of the programming language used for the implementation of the garbage collector. Next, we use the assertion language we have just defined to give Hoare-like program logic for the language.

4.1 Syntax and Semantics

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

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

Note that there are no constructs for allocating new locations on the heap. It would be straightforward to add a **cons**-like construct to the language, but since it would make the language nondeterministic, and we will not need it, we have omitted it. Note also that the forms $x := [e]$ and $[e] := e$ allow pointer arithmetic.

The operational semantics is given by a relation $\rightsquigarrow$ on *configurations*. 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:

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

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

$$\frac{[\![e]\!]s = n \quad n \in dom(h) \quad h(n) = n'}{x^{\mathsf{int}} ::= [e], s, h \rightsquigarrow s[x \mapsto n'], h} \qquad \frac{[\![e_1]\!]s = n_1 \quad [\![e_2]\!]s = n_2 \quad n_1 \in dom(h)}{[e_1] := e_2, s, h \rightsquigarrow s, h[n_1 \mapsto n_2]}$$

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

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

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

We remark that the semantics is easily seen to be deterministic. Let us introduce some terminology regarding the semantics.

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

As is standard, we define Modifies(C) for a command C to be the set of variables that are modified by the command, 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 $FV(C)$ for a command is just the set of variables that occur in C.

We now turn to program logic.

4.2 Partial Correctness Specifications and Program Logic

To reason about programs in the implementation language, we introduce partial correctness specifications.

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 $FV(A, C, B) \subseteq dom(s)$, $s, h \Vdash A$ implies

- C, s, h does not go wrong, and
- if $C, s, h \rightsquigarrow^* s', h'$, then $s', h' \Vdash B$.

We will give a program logic that is sound with respect to our partial correctness interpretation of specifications. In these rules, e, m, and f range over terms of type int, fs, and frp, respectively.

Rule for **skip**

$$\{A\}\ \mathbf{skip}\ \{A\}$$

Rules for assignment

$$\{B[e/x]\}\ x^{\mathsf{int}} := e\ \{B\}$$
$$\{B[m/x]\}\ x^{\mathsf{fs}} := m\ \{B\}$$
$$\{B[f/x]\}\ x^{\mathsf{frp}} := f\ \{B\}$$

Rule for sequencing

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

Rule for conditionals

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

Rule for **while** loops

$$\frac{\{A\ \wedge\ b\}\; C\; \{A\}}{\{A\}\; \textbf{while}\; b\; \textbf{do}\; C\; \{A\ \wedge\ \neg b\}}$$

Rule of consequence

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

Rule for heap lookup: When n is a number

$$\{e \mapsto -\ \wedge\ x = n\}\; x^{\mathsf{int}} := [e]\; \{e[n/x] \mapsto x\}$$

Second rule for heap lookup: When $x \notin FV(A, p)$,

$$\begin{array}{l} \{\exists t.\, p \mapsto t\ \wedge\ A\} \\ \quad x := [p] \\ \{p \mapsto x \wedge A[x/t]\} \end{array} \tag{66}$$

Third rule for lookup. If $y \notin FV(A)$, and x and y are distinct variables,

$$\begin{array}{l} \{A \wedge (x \hookrightarrow -)\} \\ \quad y := [x] \\ \{A \wedge (x \hookrightarrow y)\} \end{array} \tag{67}$$

Rule for heap update

$$\{e_1 \mapsto -\}\; [e_1] := e_2\; \{e_1 \mapsto e_2\} \tag{68}$$

Rule for pointers:

$$\{\mathsf{Ptr}(x)\}\; x := x + 8\; \{\mathsf{Ptr}(x)\}$$

Rules for $\oplus, \ominus$:

$$\begin{array}{l} \{\mathsf{Ptr}(x)\}\; m := m \oplus x\; \{x \in m \wedge \mathsf{Ptr}(x)\} \\ \{\mathsf{T}\}\; m := m \ominus x\; \{\neg(x \in m)\} \\ \{\mathsf{Ptr}(x) \wedge \mathsf{Ptr}(y)\}\; f := f \oplus (x, y)\; \{(x, y) \in f \wedge \mathsf{Ptr}(x) \wedge \mathsf{Ptr}(y)\} \\ \{\mathsf{T}\}\; f := f \ominus (x, y)\; \{\neg((x, y) \in f)\} \end{array}$$

$\forall$ introduction rule

$$\frac{\{A\}\; C\; \{B\}}{\{\forall x.\, A\}\; C\; \{\forall x.\, B\}}$$

The Frame Rule

$$\frac{\{A\}\; C\; \{B\}}{\{A * A'\}\; C\; \{B * A'\}}\ \text{Modifies}(C) \cap FV(A') = \varnothing$$

Derived Rule for Pure Assertions

$$\frac{\{B\}\; C\; \{B'\}}{\{B\ \wedge\ A\}\; C\; \{B' \wedge\ A\}}\ A \text{ pure, and Modifies}(C) \cap FV(A) = \varnothing$$

Rule of Conjunction

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

As an example of a useful derived rule, we note how pure assertions can move in and out of the Frame Rule:

$$\frac{\{P \wedge A\}\;C\;\{B\}}{\{P \wedge (A * B')\}\;C\;\{B * B'\}}\;P \text{ pure, and Modifies}(C) \cap FV(B') = \varnothing \tag{69}$$

This follows from Remark 3.4 and the rule of consequence. Another simple derived rule is the following. If x does not occur free in A, we have

$$\begin{array}{l} \{A\} \\ \Downarrow \\ \{A \wedge y = y\} \\ \Downarrow \\ \{(A \wedge y = x)[y/x]\} \\ \quad x := y \\ \{A \wedge x = y\} \end{array} \tag{70}$$

We then have the expected soundness result.

Theorem 4.5. *If a specification $\{A\}\;C\;\{B\}$ is derivable by the rules above, then $\{A\}\;C\;\{B\}$ holds.*

To conclude, a little word of warning about sets and the $\oplus/\ominus$ constructs: Traditionally, one has the following rule in Hoare Logic:

$$\{B(x)\}\;x := x + 1\;\{B[x - 1/x]\}$$

But note that we can *not* in general use the following rule

$$\{B(m)\}\;m := m \oplus x\;\{B[m \ominus x/m]\}$$

The problem is, of course, that we can use the rule for assignment in the first case, since in ordinary arithmetic, we always have $x = (x + 1) - 1$, and therefore

$$\{B(x)\}\; \Rightarrow \{B[((x + 1) - 1)/x]\}\;x := x + 1\;\{B[x - 1/x]\}$$

is an instance of the "substitution rule" that the rule for assignment yields. In the second case, this fails, because we do *not* have $m = (m \ominus x) \oplus x$ in general. However, if we add assumptions that ensure this, we get sound rules:

$$\begin{array}{ll} \{B(m)\;\wedge x \in m\} & \{B(m)\;\wedge\;\neg(x \in m) \wedge \mathsf{Ptr}(x)\} \\ \quad m := m \ominus x & \quad m := m \oplus x \\ \{B[m \oplus x/m] \wedge \neg(x \in m)\} & \{B[m \ominus x/m] \wedge x \in m\} \end{array} \tag{71}$$

When we increase endpoints of an interval, we must also take precautions about assertions that involve this interval. Specifically, if $m \equiv Itv(x_1, x_2)$, we have the rules

$$\begin{array}{ll} \{B(m)\;\wedge \mathsf{Ptr}(x_1)\} & \{B(m)\;\wedge \mathsf{Ptr}(x_2)\} \\ \quad x_1 := x_1 + 8 & \quad x_2 := x_2 + 8 \\ \{B[m \oplus (x_1 - 8)/m]\} & \{B[m \ominus (x_2 - 8)/m]\} \end{array} \tag{72}$$

We also have the obvious rules resembling (71) for relations.

5 Garbage Collection

In this section, we give an explanation of a correctness criterion of a garbage collector. It is based on the analysis in [COB02]. We will first outline assumptions about the overall system, and then we will give relevant definitions for specifying correctness of garbage collectors.

5.1 Assumptions

We assume that the garbage collector is written in our implementation language as part of a runtime system which a *user language* can depend on. We will not formalize the user language here, but it can, for example, be like the programming language in [COB02]. The important issue is that it always allocates locations in pairs and that its control over the heap is restricted to variable references (so there is no possibility to access the heap with non-variable references, like in $[x+84]$). In the user language, there are at least two "kinds" of values: pointers and numbers. There might be an explicit type system to distinguish them, or the distinction might be implicit in the operational semantics (like in the language of [COB02]). In the implementation language, we do not have these types; both numbers and pointers are simply integers.

To make the distinction between numbers and pointers from the user language in the implementation, we make the following translation of values from the user language into the implementation language:

- When we store an integer from the user language in the heap, we multiply it by 4 and add 1 to it.
- The translation of nil is set to -1
- The translation of a heap allocation, $x := \mathbf{cons}(E_1, E_2)$, is a call to the memory allocator that we have implemented. A program sketch of it looks like this:

```
alloc(l,n1,n2) {
  if (any_space_left) {
    allocate 2 heap cells;
    store(n1,n2);
    return first address to user lang;
  }
  else {
    Garbage collect;
    alloc(l,n1,n2);
  }
}
```

  The allocator returns the address of the first of the two cells that are allocated. As a consequence, all pointers in the user language correspond to natural numbers divisible by 8.

For simplicity, we assume that there is only one live cell, called root, in the domain of the stack from the user language. This simplifies the implementation of the garbage collector, and the proof technique used will still be clear.

5.2 Expressing Garbage Collection

We give the relevant definitions for specifying correctness of a copying garbage collector. The definitions are based on those in [COB02].

Definition 5.1. Let (s, h) be a state with $\mathsf{root} \in dom(s)$.

- Pointer p is *reachable from pointer* q in the state (s, h) if $p = q$ or if $s, h \Vdash p \hookrightarrow p_1, p_2$, and q is reachable from p_1 or p_2 in (s, h). p is called *reachable* in (s, h) if p is reachable from $s(\mathsf{root})$ in (s, h).
- (s, h) is *garbage free* if every pointer $p \in dom(h)$ is reachable in (s, h).
- $prune(s, h) = (s, g)$, where $g \subseteq h$ is the subheap of h restricted to those pointers reachable in (s, h).

Definition 5.2. Let s, h and s', h' be states. Call (s, h) and (s', h') *weakly heap-isomorphic* if there is a bijection $\beta : dom(h) \to dom(h')$ such that for all pointers $p \in dom(h)$, $h'(\beta(p)) = \beta^*(h(p))$ and $h'(\beta(p)+4) = \beta^*(h(p+4))$. Here, β^* is the extension of β to $\mathbb{Z}$ that is the identity on numbers that are not pointers. If β is a weak heap isomorphism, and $\beta(s(\mathsf{root})) = s'(\mathsf{root})$, we call β a *heap isomorphism*.

The formal notion of garbage collection is then as follows:

Definition 5.3. Let $(s, h), (s', h')$ be states. We say that h' is a *garbage collected version* of h if there exists a heap isomorphism $\beta : prune(s, h) \cong prune(s', h')$.

Note that the identity does the job, i.e., (s, h) is a garbage collected version of itself for any state (s, h). A stronger notion of garbage collection would, of course, be to require that β is a heap isomorphism $\beta : prune(s, h) \cong s', h'$. But the garbage collector, we analyze in this paper does not remove anything from the domain of the heap, so we will stick to the weaker requirement.

Therefore, we say that a garbage collector GC is *correct*, if $GC, s, h \rightsquigarrow^* s', h'$ implies that s', h' is a garbage collected version of s, h.

5.3 Auxiliary Variables

An implementation of a garbage collector does not need to use finite sets or maps like those in our term and programming languages. Indeed, these have merely been included to ease our task of reasoning about the garbage collector, and we shall use them as *auxiliary* variables, in the sense of [OG76]. Let us recall the definitions.

Definition 5.4 ([OG76, (3.6)]). Let C be a command, and let AV be a set of variables that appear in C only in assignments $x := E$, where $x \in AV$. Then AV is an *auxiliary variable set* for C.

It should be noted that x may appear in E in the assignment form above, and that in our setting, the x can have the types $\mathsf{int}, \mathsf{fs}$, and frp.

Proposition 5.5 ([OG76, (3.7)]). *Let AV be an auxiliary variable set for C', and P and Q assertions not containing free variables from AV. Let C be the command obtained from C' by deleting all assignments to the variables in AV. Then*

$$\frac{\{P\}\ C'\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

This proposition allows us to insert "extra" assignments to variables that are not directly needed for garbage collection, but rather for reasoning about it. The proposition ensures that we can obtain the same conclusions about the original program that we can for the "new" program (provided the conclusions do not contain the auxiliary variables). This technique will be used in our proof of correctness.

6 The Garbage Collection Algorithm

We have chosen to implement and reason about *Cheney's algorithm*, which was originally presented in [Che70], and is described in more detail in [JL97]. We assume that we are given two contiguous "semi-heaps" of equal size, i.e., that two intervals, $\mathsf{OLD} \equiv Itv(\mathsf{startOld}, \mathsf{endOld})$ and $\mathsf{NEW} \equiv Itv(\mathsf{startNew}, \mathsf{endNew})$ (with $\mathsf{endOld} - \mathsf{startOld} = \mathsf{endNew} - \mathsf{startNew} > 0$) are given (and in the domain of the heap). These will play the roles of "FromSpace" and "ToSpace" from [JL97].

6.1 Implementation

The implementation of the algorithm is given in Figure 6. We have not equipped variables with types, but they should be evident. As described in Section 5, the $\texttt{alloc}(l, n_1, n_2)$ procedure is called when a **cons** operation in the user language is invoked. l is the location that is returned to the user language, and n_1, n_2 are the (encodings of) values that are to be stored in the heap by the memory management system. As mentioned, we assume that we are given four constants $\mathsf{startOld}, \mathsf{endOld}, \mathsf{startNew}, \mathsf{endNew}$ which delimit the two parts of the heap that we will deal with. It allocates in OLD, if there is enough space for it. If not, it swaps the rôles of the two parts of the heap and performs a garbage collection by copying all live cells from OLD to NEW, so space is (hopefully) freed, and then calls itself

```
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;
      scan := startNew;
      free := startNew;
      maxFree := endNew
    else
      offset := startOld;
      scan := startOld;
      free := startOld;
      maxFree := endOld
    fi;
// Garbage Collection starts
    φ := ∅;
    FORW := ∅;
    UNFORW := ALIVE;
    t1 := [root];
    t2 := [root + 4];
    [free] := t1;
    [free + 4] := t2;
    [root] := free;
    FORW := FORW ⊕ root;
    UNFORW := UNFORW ⊖ root;
    φ := φ ⊕ (root, free);
    free := free + 8;

    while ¬(scan = free)
    // ScanCar begins
      x := [scan];
      if (x mod 8 = 0)
        y := [x];
        if (y mod 8 = 0 ∧
            offset ≤ y ∧
            y ≤ maxFree)
          [scan] := y
        else
          t1 := [x];
          t2 := [x + 4];
          [free] := t1;
          [free + 4] := t2;
          [x] := free;
          [scan] := free;
          FORW := FORW ⊕ x;
          UNFORW := UNFORW ⊖ x;
          φ := φ ⊕ (x, free);
          free := free + 8
        fi;
      else skip
      fi;
    // ScanCar ends
    // ScanCdr begins
      x := [scan + 4];

      if (x mod 8 = 0)
        y := [x];
        if (y mod 8 = 0 ∧
            offset ≤ y ∧
            y ≤ maxFree)
          [scan + 4] := y
        else
          t1 := [x];
          t2 := [x + 4];
          [free] := t1;
          [free + 4] := t2;
          [x] := free;
          [scan + 4] := free;
          FORW := FORW ⊕ x;
          UNFORW := UNFORW ⊖ x;
          φ := φ ⊕ (x, free);
          free := free + 8
        fi;
      else skip
      fi;
    // ScanCdr ends
      scan := scan + 8
    od;
// Garbage Collection ends
    root := offset;
    malloc(l,n1,n2)
  fi
}
```

Figure 6: The Garbage Collection Algorithm

recursively. We will assume here that we are dealing with the case where we have to copy cells from the lower half of the address space to the upper half, so offset is set to startNew.

We have used the simplifying assumption about root from Sec. 5.1 in the implementation, inasmuch as we only deal with that pointer before we start executing the **while** loop. If there were more locations in the domain of the stack of the user language, the code before the **while** loop would need to be modified to deal with all these pointers. We will concentrate on the garbage collection part GC^* of the memory allocator; it is delimited in the code by comments.

6.2 Specification

We present a specification for the garbage collector. We treat the assumption and the conclusion in separate subsections.

6.2.1 Preconditions

Informally, the conditions that must be met for a state s, h for the algorithm to work correctly are:

- There is a variable root $\in dom(s)$, and $s(\mathsf{root}) \in \mathsf{OLD}$.
- For all locations $l \in \mathsf{OLD}\& = \mathsf{OLD} \cup \{p + 4 \mid p \in \mathsf{OLD}\}$, if $h(l) = q$, and q is a pointer, then $q \in \mathsf{OLD}$. In particular, all reachable cells are in OLD.
- The reachable part of OLD comes in "pairs": if a pointer $p \in dom(h) \cap \mathsf{OLD}$, then $p + 4 \in dom(h)$, and if a location $l \notin Ptr$ is in $dom(h)$, then $l - 4 \in Ptr$ must also be in $dom(h)$.
- The locations in NEW& must be in the domain of the heap.

The last requirement means enables us to copy the live cells into NEW *in a deterministic way*. If we did not assume NEW to be in the domain of the initial heap, but rather used a non-deterministic **cons** operation, we would not be able to tell in advance where the copied cells were located, and that would make the breadth-first scanning, which the algorithm performs, more difficult.

If we define the finite set ALIVE to be the set of pointers that are reachable from root, and if we let head and tail be two single-valued relations (from pointers to integers) that record the initial state of the part of the heap whose domain is ALIVE&, then the following assertion will hold initially.

$$(\forall_* p \in \mathsf{ALIVE}.\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q')))*$$
$$(\forall_* p \in \mathsf{NEW}. p \mapsto -, -)$$

Note also that the assumptions mean that the contents of any cell in ALIVE is either a pointer in ALIVE or a nonpointer. Further assumptions needed are that various variables / constants denote pointers, and that ALIVE is the set of pointers that are reachable from root. Summing up, the following assertion is assumed to hold when we initiate the garbage collector:

$$
\begin{aligned}
&\mathsf{InitAss} \equiv \\
&\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{maxFree}) \wedge \mathsf{Disjoint}(\mathsf{OLD}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{ALIVE} \wedge \\
&\mathsf{SbSet}(\mathsf{ALIVE}, \mathsf{OLD}) \wedge \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge \#\mathsf{NEW} = \#\mathsf{OLD} \wedge \\
&\mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \\
&((\forall_* p \in \mathsf{ALIVE}.\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q')))* \\
&(\forall_* p \in \mathsf{NEW}. p \mapsto -, -) * \mathsf{T})
\end{aligned}
$$

The T in the assertion above is due to the fact that we have not said anything about unreachable cells. Our algorithm does not affect these "dead" cells, so the specification need not mention them (we can include them in the specification afterwards with the Frame Rule). Therefore, the T in InitAss will henceforth be omitted.

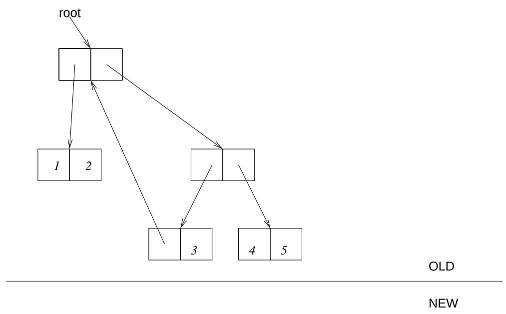

Figure 7: A sample heap

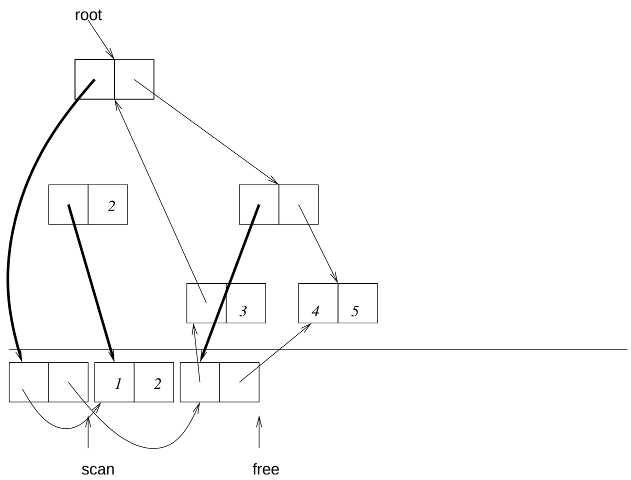

Figure 8: A stage in execution

We immediately note that some of the conjuncts in this assertion contain no free variables that are modified by the algorithm. Therefore, we define a (pure) assertion which is therefore trivially an invariant of the algorithm. This will save space later.

$$\begin{array}{l}\mathsf{I}_{pure} \equiv \\ \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{maxFree}) \wedge \\ \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE})\end{array}$$

6.2.2 The Invariant and the Conclusion

We have to exhibit an invariant for the **while**-loop that is weak enough for us to show and strong enough to conclude that there exists a weak heap-isomorphism between the initial heap and the heap after execution. We start with an informal analysis. Suppose we run the algorithm on the heap depicted in Fig. 7, where dead cells are not shown.

At some point during execution, the heap will look like that of Fig. 8 (forwarding pointers are marked with a bolder arrow).

We see that there are two “kinds” of cells in the part of the heap whose domain is ALIVE:

- Some have not yet been copied into NEW. These have have a pointer to a cell in ALIVE or a non-pointer in their first component. We call the set of these cells UNFORW, and it can be seen on Fig. 9.

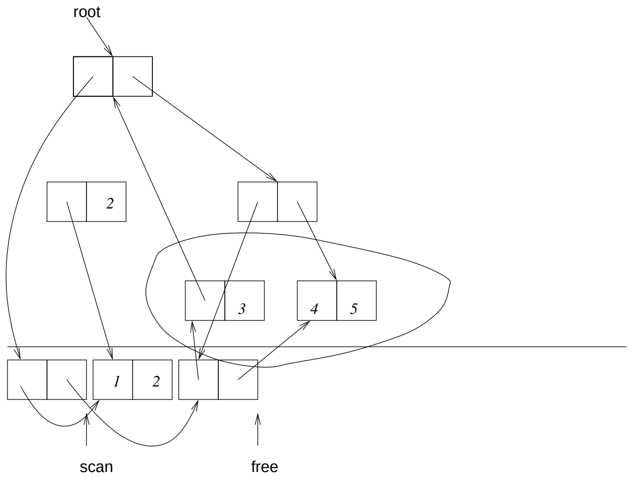

Figure 9: Illustration of $\mathsf{A}_{\mathsf{UNFORW}}$

- Some have been copied into NEW. These have their first component replaced with a pointer to the copy. The set of these cells is called FORW. See Fig. 10 for an illustration.

The pointer free in the program is used to indicate where to "allocate" the next cell we copy. So we may partition NEW into two disjoint sets BUSY and FREE, where

$$\mathsf{BUSY} \equiv Itv(\mathsf{offset}, \mathsf{free}), \quad \text{and}$$
$$\mathsf{FREE} \equiv Itv(\mathsf{free}, \mathsf{maxFree}).$$

We may partition NEW further. The pointer scan indicates the border between the cells that have already been "scanned" (and will not be manipulated further by the algorithm), and those that have not yet been scanned. So BUSY is equal to the disjoint union of the following intervals.

$$\mathsf{FIN} \equiv Itv(\mathsf{offset}, \mathsf{scan}), \quad \text{and}$$
$$\mathsf{UNFIN} \equiv Itv(\mathsf{scan}, \mathsf{free}).$$

Next, we note that each cell in FORW corresponds to exactly one cell in BUSY, namely the cell that it points to in its first component. Thus, there is a (single-valued) relation (on pointers) that can be viewed as a bijection,

$$\varphi \subseteq \mathsf{FORW} \times \mathsf{BUSY},$$

where $(p, q) \in \varphi$ iff q is the value of the heap at p after it has been updated to point to a cell in NEW.

Given these partitions and relations, we can formalize what we know about the pointers in each of the sets.

- The pointers p in UNFORW have not yet been altered by the algorithm, so for these, there are q, q' such that $(p, q) \in \mathsf{head}$, $(p, q') \in \mathsf{tail}$, and $p \hookrightarrow q, q'$. Therefore, we define the assertion

  $$\mathsf{A}_{\mathsf{UNFORW}} \equiv$$
  $$\forall_* p \in \mathsf{UNFORW}.\ ((\exists q.\ (p, q) \in \mathsf{head} \land p \mapsto q) * (\exists q.\ (p, q') \in \mathsf{tail} \land p + 4 \mapsto q'))$$

  This is illustrated in Fig. 9.

- The pointers p in FORW have their first component overwritten by the values they correspond to by the relation φ, so for these, there is a q such that p points to q and $(p, q) \in \varphi$. Consequently, we define (see also Fig. 10):

  $$\mathsf{A}_{\mathsf{FORW}} \equiv \forall_* p \in \mathsf{FORW}.(\exists q^{\mathsf{int}}.\ (p, q) \in \varphi \land p \mapsto q, -)$$

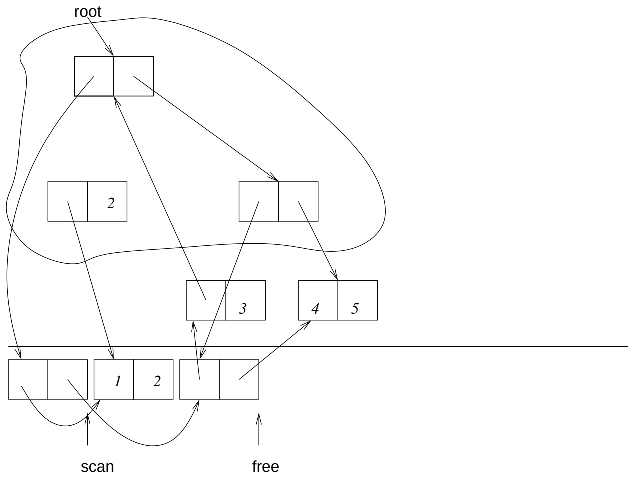

Figure 10: Illustration of $\mathsf{A_{FORW}}$

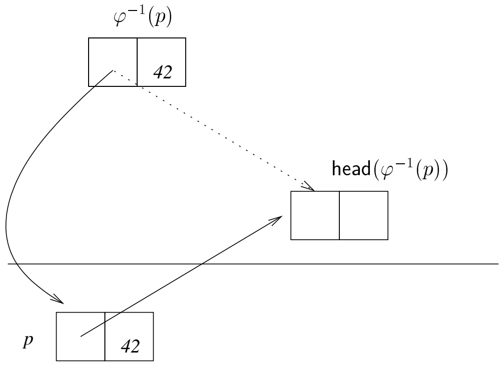

Figure 11: Illustration of a pointer in UNFIN

- For all the pointers p in UNFIN there are cells $p' \in \mathsf{ALIVE}$ which contained the contents of p before the run of the algorithm (p is just a copy of p' before p' had its first component updated, and if so, $(p', p) \in \varphi$). Therefore, there are q, q' such that $(p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge (p, q') \in \mathsf{tail} \circ \varphi^\dagger$ and such that p points to q and q' (that is, p points to q and $p + 4$ points to q'). We define

$$\begin{aligned}&\mathsf{A_{UNFIN}} \equiv \\ &\forall_* p \in \mathsf{UNFIN}.((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\ &\qquad\qquad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q'))\end{aligned}$$

  There is an illustration of this in Fig. 11, where we have used a "functional notation" involving the relations φ and head.

- When a pointer p is scanned, the components (q_1, q_2) of the cell it points to, are redirected to the values they correspond to under the bijection φ in both of the cases in the **while** loop, if they are pointers. If not, they are left untouched. Therefore, for all pointers p in FIN there are integers q, q' such that p points to q, q', $(p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)$, and $(p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^\dagger)$. So we define

$$\begin{aligned}&\mathsf{A_{FIN}} \equiv \\ &\forall_* p \in \mathsf{FIN}.((\exists q.\ (p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge p \mapsto q) * \\ &\qquad\qquad (\exists q'.\ (p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^\dagger) \wedge p + 4 \mapsto q')).\end{aligned}$$

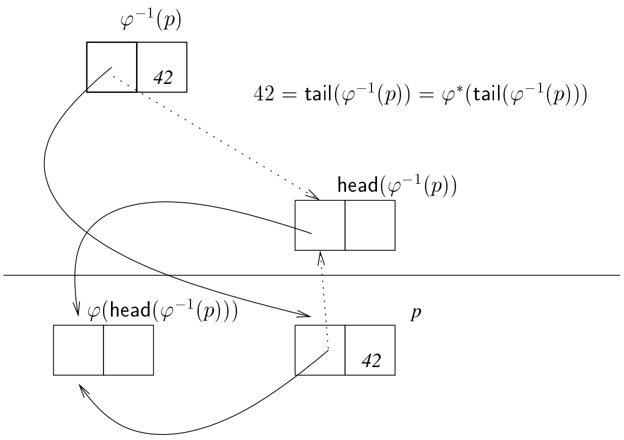

Figure 12: The situation for a pointer in FIN

Consult Fig. 12 for an illustration.

- The pointers $p \in \mathsf{FREE}$ are in the domain of the heap, so we set

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

Further, the union of the sets FORW and UNFORW is ALIVE. root is in FORW after the initialization, and φ is an isomorphism between FORW and BUSY. ALIVE has at most as many elements as NEW (since ALIVE is a subset of OLD which has the same size as NEW), and scan never exceeds free (and these are both pointers). The entities ALIVE, head and tail are not modified by the algorithm. Thus, the invariant of the algorithm is

$$\begin{aligned}
&I \equiv\\
&\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge\\
&\mathsf{root} \in \mathsf{FORW}\ \wedge \mathsf{scan} \leq \mathsf{free}\ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge\\
&(\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \mathsf{A}_{\mathsf{FREE}})
\end{aligned}$$

For the assertions $\mathsf{A}_{\_}$ defined above, we will abuse notation a little. Sometimes, we will need to consider the separating conjunction over the sets involved, except for one element which we will consider for separately. For example, $\mathsf{A}_{\mathsf{FORW}-x}$ will denote the assertion (compare with the definition of $\mathsf{A}_{\mathsf{FORW}}$)

$$\forall_* p \in (\mathsf{FORW} \ominus x). \ (\exists q^{\mathsf{int}}. \ (p, q) \in \varphi \wedge p \mapsto q, -).$$

This should not cause any confusion.

We will prove that the specification, we have outlined, holds in Section 7, and in Section 8, we will prove that this implies correctness of the algorithm.

7 Proving the Invariant

In this section, we will initiate the actual proof of our garbage collector. We have formulated a precondition $\mathsf{InitAss}$ and an invariant I, and in this section, we will show that

- I is established when the code before the **while**-loop is run in a state in which $\mathsf{InitAss}$ holds.
- The body of the **while**-loop preserves I, i.e., if I and the condition of the loop hold in a state, then after execution of the body from that state, I holds in the resulting state.

7.1 Establishing the Invariant

In this section we show that the initializing code preceding the **while** loop establishes I when run in a state satisfying the precondition from Section 6.2.1.

Therefore, let INIT and INIT^* be the code fragments

$$\begin{array}{llcll}
\mathsf{INIT} \equiv & t_1 := [\mathsf{root}]; & \qquad & \mathsf{INIT}^* \equiv & \mathsf{INIT}; \\
& t_2 := [\mathsf{root}+4]; & & & \mathsf{FORW} := \mathsf{FORW} \oplus \mathsf{root}; \\
& [\mathsf{free}] := t_1; & & & \mathsf{UNFORW} := \mathsf{UNFORW} \ominus \mathsf{root}; \\
& [\mathsf{free}+4] := t_2; & & & \varphi := \varphi \oplus (\mathsf{root}, \mathsf{free}); \\
& [\mathsf{root}] := \mathsf{free} & & & \mathsf{free} := \mathsf{free}+8;
\end{array}$$

We first infer a local specification for INIT, and then use this specification and the Frame Rule to obtain a global specification for INIT^*.

The specification below only mentions the locations that are affected by INIT.

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

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

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

is valid by the rule (66) for heap lookup, since the assertion $(\mathsf{root}, q) \in \mathsf{head}$ is pure. The first specification above is then valid by the Frame Rule. The second specification is valid by the same argument; here we use that $(\mathsf{root}, q') \in \mathsf{tail}$ is pure. For the third and the fourth specifications, we use the Frame Rule again, along with the rule for heap update (68). The implication follows from Remark 3.4 (we use that 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 (68) for update and the Frame Rule again.

From this local specification, we use the Frame Rule to infer a global specification for INIT^*.

$\mathsf{InitAss}$

$\Downarrow$

$$\{\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Disjoint}(\mathsf{OLD}, \mathsf{NEW}) \wedge \mathsf{Ptr}(\mathsf{maxFree}) \wedge \mathsf{root} \in \mathsf{ALIVE} \wedge \\ \mathsf{SbSet}(\mathsf{ALIVE}, \mathsf{OLD}) \wedge \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge \#\mathsf{NEW} = \#\mathsf{OLD} \wedge \\ \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \\ ((\forall_* p \in \mathsf{ALIVE}.\ ((\exists q.\ (p,q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q.\ (p,q') \in \mathsf{tail} \wedge p+4 \mapsto q')))* \\ (\forall_* p \in \mathsf{NEW}.\ p \mapsto -,-))\}$$

$$\mathsf{scan} := \mathsf{offset}; \mathsf{free} := \mathsf{offset}; \mathsf{FORW} := \varnothing; \\ \varphi := \varnothing; \mathsf{UNFORW} := \mathsf{ALIVE}$$

$$\{\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Disjoint}(\mathsf{OLD}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{ALIVE} \wedge \mathsf{Ptr}(\mathsf{maxFree}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \\ \mathsf{SbSet}(\mathsf{ALIVE}, \mathsf{OLD}) \wedge \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge \#\mathsf{NEW} = \#\mathsf{OLD} \wedge \\ \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \\ ((\forall_* p \in \mathsf{ALIVE}.\ ((\exists q.\ (p,q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q.\ (p,q') \in \mathsf{tail} \wedge p+4 \mapsto q')))* \\ (\forall_* p \in \mathsf{NEW}.\ p \mapsto -,-)) \wedge \\ \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FORW} = \varnothing \wedge \varphi = \varnothing \wedge \mathsf{UNFORW} = \mathsf{ALIVE}\}$$

$\Downarrow$

$$\{\mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{UNFORW} \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \\ \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\ \mathsf{I}_{pure} \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FORW} = \varnothing \wedge \varphi = \varnothing \wedge \\ (\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \mathsf{A}_{\mathsf{FREE}})$$

$\Downarrow$

$$\{\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{UNFORW} \wedge \\ \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\ \mathsf{I}_{pure} \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FORW} = \varnothing \wedge \varphi = \varnothing \wedge \\ ((\forall_* p \in (\mathsf{UNFORW} \ominus \mathsf{root}).\ ((\exists q.\ (p,q) \in \mathsf{head} \wedge p \mapsto q)* \\ (\exists q'.\ (p,q') \in \mathsf{tail} \wedge p+4 \mapsto q'))) * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \\ (\forall_* p \in (\mathsf{FREE} \ominus \mathsf{free}).\ p \mapsto -,-) * (\mathsf{free} \mapsto -,-)* \\ (\exists q.\ (\mathsf{root}, q) \in \mathsf{head} \wedge \mathsf{root} \mapsto q) * (\exists q'.\ (\mathsf{root}, q') \in \mathsf{tail} \wedge \mathsf{root}+4 \mapsto q'))\}$$

$$\mathsf{INIT}$$

$$\{\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{UNFORW} \wedge \\ \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\ \mathsf{I}_{pure} \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \mathsf{FORW} = \varnothing \wedge \varphi = \varnothing \wedge \\ ((\forall_* p \in (\mathsf{UNFORW} \ominus \mathsf{root}).\ ((\exists q.\ (p,q) \in \mathsf{head} \wedge p \mapsto q)* \\ \quad (\exists q'.\ (p,q') \in \mathsf{tail} \wedge p+4 \mapsto q')))* \\ \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \\ (\forall_* p \in (\mathsf{FREE} \ominus \mathsf{free}).\ p \mapsto -,-) * ((\mathsf{free} \mapsto t_1, t_2) * (\mathsf{root} \mapsto \mathsf{free}, -) \\ \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail}))\}$$

$\Downarrow$

$$\{\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{root} \in \mathsf{UNFORW} \wedge \\ \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\ \mathsf{I}_{pure} \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{free} = \mathsf{offset} \wedge \neg(\mathsf{root} \in \mathsf{FORW}) \wedge \neg((\mathsf{root}, \mathsf{free}) \in \varphi) \wedge \\ (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \\ ((\forall_* p \in (\mathsf{UNFORW} \ominus \mathsf{root}).\ ((\exists q.\ (p,q) \in \mathsf{head} \wedge p \mapsto q)* \\ \quad (\exists q'.\ (p,q') \in \mathsf{tail} \wedge p+4 \mapsto q')))* \\ \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \\ (\forall_* p \in (\mathsf{FREE} \ominus \mathsf{free}).\ p \mapsto -,-) * (\mathsf{free} \mapsto t_1, t_2) * (\mathsf{root} \mapsto \mathsf{free}, -))\}$$

$\mathsf{FORW} := \mathsf{FORW} \oplus \mathsf{root};$
$\mathsf{UNFORW} := \mathsf{UNFORW} \ominus \mathsf{root};$
$\varphi := \varphi \oplus (\mathsf{root}, \mathsf{free})$

$$\begin{array}{l}
\{\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{offset} = \mathsf{free} \wedge (\mathsf{root}, \mathsf{free}) \in \varphi \wedge \\
\mathsf{iso}(\varphi \ominus (\mathsf{root}, \mathsf{free}), \mathsf{FORW} \ominus \mathsf{root}, \mathsf{BUSY}) \wedge \mathsf{root} \in \mathsf{FORW} \wedge \mathsf{I}_{pure} \wedge \\
\mathsf{isUnion}(\mathsf{FORW} \ominus \mathsf{root}, \mathsf{UNFORW} \oplus \mathsf{root}, \mathsf{ALIVE}) \wedge \neg(\mathsf{root} \in \mathsf{UNFORW}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
((\forall_* p \in ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}).\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) * \\
(\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free}) \wedge p \mapsto q, -)) * \\
(\forall_* p \in \mathsf{FIN}.\ ((\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free}) \circledcirc (\mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free}))^\dagger) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \varphi \ominus (\mathsf{root}, \mathsf{free}) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free}))^\dagger) \wedge p + 4 \mapsto q'))) * \\
(\forall_* p \in \mathsf{UNFIN}.\ ((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free}))^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free}))^\dagger) \wedge p + 4 \mapsto q')) * \\
(\forall_* p \in (\mathsf{FREE} \ominus \mathsf{free}).\ p \mapsto -, -) * (\mathsf{root} \mapsto \mathsf{free}, -) * (\mathsf{free} \mapsto t_1, t_2))\}
\end{array}$$

$\mathsf{free} := \mathsf{free} + 8$

$$\begin{array}{l}
\{\mathsf{Ptr}(\mathsf{free} - 8) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{scan} = \mathsf{offset} \wedge \\
\mathsf{offset} = \mathsf{free} - 8 \wedge (\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge \\
\mathsf{iso}(\varphi \ominus (\mathsf{root}, \mathsf{free} - 8), \mathsf{FORW} \ominus \mathsf{root}, \mathsf{BUSY} \ominus (\mathsf{free} - 8)) \wedge \mathsf{root} \in \mathsf{FORW} \wedge \mathsf{I}_{pure} \wedge \\
\mathsf{isUnion}(\mathsf{FORW} \ominus \mathsf{root}, \mathsf{UNFORW} \oplus \mathsf{root}, \mathsf{ALIVE}) \wedge \neg(\mathsf{root} \in \mathsf{UNFORW}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
((\forall_* p \in ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}).\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) * \\
(\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \wedge p \mapsto q, -)) * \\
(\forall_* p \in \mathsf{FIN}. \\
\quad ((\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \circledcirc (\mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q'))) * \\
(\forall_* p \in (\mathsf{UNFIN} \ominus (\mathsf{free} - 8)).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')) * \\
(\forall_* p \in ((\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8)).\ p \mapsto -, -) * (\mathsf{root} \mapsto (\mathsf{free} - 8), -) * \\
((\mathsf{free} - 8) \mapsto t_1, t_2))\}
\end{array}$$

$\Downarrow$

$$\begin{array}{l}
\{\mathsf{Ptr}(\mathsf{free} - 8) \wedge \mathsf{Ptr}(\mathsf{root}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{scan} = \mathsf{offset} \wedge \\
\mathsf{offset} = \mathsf{free} - 8 \wedge (\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge \\
\mathsf{iso}(\varphi \ominus (\mathsf{root}, \mathsf{free} - 8), \mathsf{FORW} \ominus \mathsf{root}, \mathsf{BUSY} \ominus (\mathsf{free} - 8)) \wedge \mathsf{root} \in \mathsf{FORW} \wedge \mathsf{I}_{pure} \wedge \\
\mathsf{isUnion}(\mathsf{FORW} \ominus \mathsf{root}, \mathsf{UNFORW} \oplus \mathsf{root}, \mathsf{ALIVE}) \wedge \neg(\mathsf{root} \in \mathsf{UNFORW}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
((\forall_* p \in ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}).\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) * \\
((\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \wedge p \mapsto q, -)) * \\
(\mathsf{root} \mapsto (\mathsf{free} - 8), -)) * \\
(\forall_* p \in \mathsf{FIN}. \\
\quad ((\exists q.\ (p, q) \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \circledcirc (\mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q'))) * \\
((\forall_* p \in (\mathsf{UNFIN} \ominus (\mathsf{free} - 8)).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')) * ((\mathsf{free} - 8) \mapsto t_1, t_2)) * \\
(\forall_* p \in ((\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8)).\ p \mapsto -, -))\}
\end{array}$$

In this derivation, the second step uses the derived rule (70) for assignment several times, and (52) to conclude $\mathsf{Ptr}(\mathsf{root})$. The third step uses the rule (2) for $\forall_*$ (to conclude $\mathsf{A}_{\mathsf{FORW}}, \mathsf{A}_{\mathsf{FIN}}$, and $\mathsf{A}_{\mathsf{UNFIN}}$) and (1) (for $\mathsf{A}_{\mathsf{UNFORW}}$ and $\mathsf{A}_{\mathsf{FREE}}$). This step also uses the set-theoretic rules (31), (7), (51), and (25). The fourth step uses the rule (5), and we use our "derived Frame Rule" (69) and our local specification from before to take the fifth step. The sixth step is a consequence of purity and the rules (61), (62), and the seventh follows from the rules in (71). The step for

$\mathsf{free} := \mathsf{free} + 8$ uses the rules in (72), and the last is just a matter of rewriting.

It is now our task to show that the invariant I follows from the conclusion in this derivation. Before we embark on this, we describe the method used to do it.

I can be viewed as a conjunction $I \equiv I_1 \wedge \cdots \wedge I_k$ of assertions, where some of the I_i's are pure and one is not pure; let us say that I_k is the impure part of I, and let us write I_k on the form $A_1 * \cdots * A_m$. Similarly, the conclusion in the derivation has the form $I'_1 \wedge \cdots \wedge I_{k'}$ where I'_i are pure for $i \in \{1, \cdots, k'-1\}$ and where $I_{k'} \equiv A'_1 * \cdots * A'_m$.

The strategy is to show each of $I_1, \ldots, I_{k-1}$ from $I'_1 \wedge \cdots \wedge I'_{k'-1}$ and each of the A_i from $I'_1 \wedge \cdots \wedge I'_{k'-1} \wedge A'_i$. This is sufficient by the remark after Lemma 3.8.

We therefore write down each of the conjuncts in I, starting with the pure parts.

- $\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY})$. This follows from

  $$\begin{array}{l}\mathsf{iso}(\varphi \ominus (\mathsf{root}, \mathsf{free} - 8), \mathsf{FORW} \ominus \mathsf{root}, \mathsf{BUSY} \ominus (\mathsf{free} - 8)) \wedge \mathsf{root} \in \mathsf{FORW} \wedge \\ (\mathsf{free} - 8) \in \mathsf{BUSY} \wedge (\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge \mathsf{Ptr}(\mathsf{free} - 8)\end{array}$$

  and (9). To be rigorous, we do not have $\mathsf{free} - 8 \in \mathsf{BUSY}$ *per se*. But we know $\mathsf{Ptr}(\mathsf{free} - 8) \wedge \mathsf{free} - 8 = \mathsf{offset}$, so by (49), we get $\mathsf{free} - 8 \in \mathsf{BUSY}$.

- $\mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE})$ follows from

  $$\mathsf{isUnion}(\mathsf{FORW} \ominus \mathsf{root}, \mathsf{UNFORW} \oplus \mathsf{root}, \mathsf{ALIVE}) \wedge \mathsf{root} \in \mathsf{FORW}$$

  and (29).

- $$\mathsf{root} \in \mathsf{FORW} \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{I}_{pure} \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW})$$

  follows from the same assertion that is a part of the conclusion above.

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

- $\mathsf{Ptr}(\mathsf{scan}) \wedge \mathsf{Ptr}(\mathsf{free})$. Follows from ordinary set manipulations (57), and from $\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{scan} = \mathsf{offset} \wedge \mathsf{Ptr}(\mathsf{free} - 8)$.

For the impure parts, we need more work. We will deal with each of the parts in the iterated separating conjunction $(\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \mathsf{A}_{\mathsf{FREE}})$ separately.

For UNFORW, we have

$$\begin{array}{l}\neg(\mathsf{root} \in \mathsf{UNFORW}) \wedge \\ (\forall_* p \in ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}). \\ \quad ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) \\ \Downarrow \\ ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}) = \mathsf{UNFORW} \wedge \\ (\forall_* p \in ((\mathsf{UNFORW} \oplus \mathsf{root}) \ominus \mathsf{root}). \\ \quad ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) \\ \Downarrow \\ (\forall_* p \in \mathsf{UNFORW}.\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) * (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) \\ \shortparallel \\ \mathsf{A}_{\mathsf{UNFORW}},\end{array}$$

where the two implications follow from (45) and (1), respectively.

For FORW,

$$
\begin{array}{l}
(\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge \mathsf{root} \in \mathsf{FORW} \wedge \\
((\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p,q) \in \varphi \ominus (\mathsf{root}, \mathsf{free} - 8) \wedge p \mapsto q, -)) * \\
(\mathsf{root} \mapsto \mathsf{free} - 8, -)) \\
\Downarrow \\
((\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p,q) \in \varphi \wedge p \mapsto q, -)) * \\
(\mathsf{root} \mapsto \mathsf{free} - 8, - \wedge (\mathsf{root}, \mathsf{free} - 8) \in \varphi)) \wedge \mathsf{root} \in \mathsf{FORW} \\
\Downarrow \\
((\forall_* p \in (\mathsf{FORW} \ominus \mathsf{root}).\ (\exists q.\ (p,q) \in \varphi \wedge p \mapsto q, -)) * \\
(\exists q.\ \mathsf{root} \mapsto q, - \wedge (\mathsf{root}, q) \in \varphi)) \wedge \mathsf{root} \in \mathsf{FORW} \\
\Downarrow \\
\forall_* p \in \mathsf{FORW}.\ (\exists q.\ (p,q) \in \varphi \wedge p \mapsto q, -) \\
\shortparallel \\
\mathsf{A}_{\mathsf{FORW}}
\end{array}
$$

The first implication follows from the rule for pure assertions in Remark 3.4, from (47), and Lemma 3.7. The third implication follows from (6).

For FIN, the easiest proof is to note that the condition $\mathsf{scan} = \mathsf{offset}$ in the conclusion above makes both of the assertions about FIN equivalent to emp. Another way is to use (47) and Lemma 3.7 to "extend" φ as it is done in the derivation for UNFIN below.

The following derivation takes care of FREE (we have implicitly used the rule (48)):

$$
\begin{array}{l}
\neg(\mathsf{free} - 8 \in \mathsf{FREE}) \wedge \\
\forall_* p \in ((\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8).\ p. \to -, -) \\
\Downarrow \\
\mathsf{FREE} = (\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8) \wedge \\
\forall_* p \in ((\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8).\ p. \to -, -) \\
\Downarrow \\
\forall_* p \in \mathsf{FREE}.\ p \mapsto -, - \\
\shortparallel \\
\mathsf{A}_{\mathsf{FREE}}
\end{array}
$$

The first implication here is an instance of (45), and the second implication follows from (1).

Finally, for UNFIN, we use (49) and get

$$
\begin{array}{l}
(\mathsf{root}, \mathsf{free} - 8) \in \varphi \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
\mathsf{Ptr}(\mathsf{free} - 8) \wedge (\mathsf{free} - 8) \in \mathsf{UNFIN} \wedge \\
((\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{free} - 8).\ ((\exists q.\ (p,q) \in \mathsf{head} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p,q') \in \mathsf{tail} \circ (\varphi \ominus (\mathsf{root}, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q'))) * \\
(\mathsf{free} - 8 \mapsto t_1, t_2)) \\
\Downarrow \\
(\mathsf{free} - 8, \mathsf{root}) \in \varphi^\dagger \wedge (\mathsf{root}, t_1) \in \mathsf{head} \wedge (\mathsf{root}, t_2) \in \mathsf{tail} \wedge \\
\mathsf{Ptr}(\mathsf{free} - 8) \wedge (\mathsf{free} - 8) \in \mathsf{UNFIN} \wedge \\
((\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{free} - 8).\ ((\exists q.\ (p,q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p,q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q'))) * \\
(\mathsf{free} - 8 \mapsto t_1) * ((\mathsf{free} - 8) + 4 \mapsto t_2)) \\
\Downarrow \\
(\mathsf{free} - 8, t_1) \in \mathsf{head} \circ \varphi^\dagger \wedge (\mathsf{free} - 8, t_2) \in \mathsf{tail} \circ \varphi^\dagger \wedge (\mathsf{free} - 8) \in \mathsf{UNFIN} \wedge \\
((\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{free} - 8).\ ((\exists q.\ (p,q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\ (p,q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q'))) * \\
(\mathsf{free} - 8 \mapsto t_1) * ((\mathsf{free} - 8) + 4 \mapsto t_2)) \\
\Downarrow
\end{array}
$$

$$
\begin{array}{l}
\quad (\mathsf{free} - 8) \in \mathsf{UNFIN} \wedge \\
\quad ((\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{free} - 8).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\qquad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q'))) * \\
\quad (\mathsf{free} - 8 \mapsto t_1 \wedge (\mathsf{free} - 8, t_1) \in \mathsf{head} \circ \varphi^\dagger) * \\
\quad ((\mathsf{free} - 8) + 4 \mapsto t_2 \wedge (\mathsf{free} - 8, t_2) \in \mathsf{tail} \circ \varphi^\dagger)) \\
\Downarrow \\
\quad (\mathsf{free} - 8) \in \mathsf{UNFIN} \wedge \\
\quad ((\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{free} - 8).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\qquad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q'))) * \\
\quad (\exists q.\ \mathsf{free} - 8 \mapsto q \wedge (\mathsf{free} - 8, q) \in \mathsf{head} \circ \varphi^\dagger) * \\
\quad (\exists q'.\ (\mathsf{free} - 8) + 4 \mapsto q' \wedge (\mathsf{free} - 8, q') \in \mathsf{tail} \circ \varphi^\dagger)) \\
\Downarrow \\
\quad \forall_* p \in \mathsf{UNFIN}.\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\qquad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')) \\
\quad \shortparallel \\
\quad \mathsf{A}_{\mathsf{UNFIN}}
\end{array}
$$

The first implication here uses (47), Lemma 3.7, (58), and Lemma 3.7. The second follows from elementary set theory (59), and the third implication follows from purity. Finally, the last implication is an instance of the rule (5).

This establishes that running INIT^* starting from a state that satisfies the precondition InitAss from Section 6.2.1 terminates in a state that satisfies the invariant I, as desired.

”

7.2 Maintaining the Invariant

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

$$
\begin{array}{l}
\{I \ \wedge\ \neg(\mathsf{scan} = \mathsf{free})\} \\
\quad \mathsf{BODY} \\
\{I\}
\end{array}
$$

holds, where BODY is the body of the **while** loop. As a first step, we note that BODY consists of two similar parts ScanCar and ScanCdr, one for each component 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 component of it is “finished”, whereas the other is about to be “scanned”. All other pointers involved are “settled”, i.e., belong to one of the sets mentioned in the invariant. The aim is thus to show that the following sequence of specifications holds.

$$
\begin{array}{l}
\{I \ \wedge\ \neg(\mathsf{scan} = \mathsf{free})\} \\
\quad \mathsf{ScanCar}; \\
\{I'\} \\
\quad \mathsf{ScanCdr}; \\
\quad \mathsf{scan} := \mathsf{scan} + 8 \\
\{I\},
\end{array}
\tag{73}
$$

where I' holds in the intermediate state where scan is “halfway between UNFIN and FIN”:

$$
\begin{array}{l}
I' \equiv \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\#\mathsf{ALIVE} \le \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \le \mathsf{free} \wedge \\
\mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
(\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN} \ominus \mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}} * \\
(\exists q.\ (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto q) \\
(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto q'))
\end{array}
$$

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

1. If x is a non-pointer, nothing happens.
2. If x is a pointer, we branch according to the value y of the first component of the cell pointed to by x.
   (a) If y is a pointer in NEW, the cell has already been copied, and the copy is located at y, so we can just update [scan] to y.
   (b) If y is a nonpointer or a pointer in ALIVE, the cell has not yet been copied. Therefore, we do so, and we also update the first component of the cell pointed to by x to a pointer to the copy (to mark that it has been copied), and we also update [scan] to point to the new copy.

We first formalize the effect of the command $x := [\mathsf{scan}]$. We have

$$
\begin{array}{l}
\quad\{I \wedge \neg(\mathsf{scan} = \mathsf{free})\} \\
\Downarrow \\
\quad\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\quad \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
\quad \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
\quad (\exists q.\ (\mathsf{scan}, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{scan} \mapsto q) * \\
\quad (\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q'))\} \\
\qquad x := [\mathsf{scan}] \\
\quad\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\quad \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
\quad \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
\quad ((\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{scan} \mapsto x) * \\
\quad (\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q'))\} \\
\Downarrow \\
\quad\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\quad \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\quad \neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
\quad (\mathsf{scan} \mapsto x) * (\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q'))\}
\end{array}
$$

The first step here is due to (5), the specification step uses the rule (66) for lookup and the derived version (69) of the Frame Rule; the last step uses purity.

According to the rule of conditionals, there are two specifications to be shown, according to the outer **if**-branch in ScanCar. If we let I_x be the conclusion in the derivation above, the first of these is

$$
\begin{array}{l}
\quad\{I_x \wedge \neg(x \bmod 8 = 0)\} \\
\Downarrow \\
\quad\{I_x \wedge \neg\mathsf{Ptr}(x)\} \\
\qquad \textbf{skip} \\
\quad\{I'\}
\end{array}
\tag{74}
$$

The second specification we have to show for the outer **if**-branch contains an inner **if**-branch, so it also splits into two specifications. Before writing these down, we formalize the effect of the command $y := [x]$. We have

$$
\begin{array}{l}
\{I_x \wedge x \bmod 8 = 0\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x)\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge x \in \mathsf{ALIVE}\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \in \mathsf{FORW} \vee x \in \mathsf{UNFORW})\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow - \vee x \hookrightarrow -)\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow -)\} \\
\quad y := [x] \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y)\}
\end{array}
$$

The first of the implications uses (56), and the second follows from $(\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE})$, (40), and (41). The third follows from (22), and the fourth follows from (3). Finally the specification follows from (67).

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

- $$
\begin{array}{l}
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \bmod 8 = 0 \wedge \mathsf{offset} \leq y \leq \mathsf{maxFree}\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW}\} \\
\quad [\mathsf{scan}] := y \\
\{I'\}
\end{array}
\tag{75}
$$

- $$
\begin{array}{l}
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \bmod 8 = 0 \wedge \mathsf{offset} \leq y \leq \mathsf{maxFree})\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \in \mathsf{NEW})\} \\
\quad \mathsf{CopyCell}^* \\
\{I'\},
\end{array}
\tag{76}
$$

where

$$
\begin{array}{rcl}
\mathsf{CopyCell}^* & \equiv & t_1 := [x]; \\
& & t_2 := [x+4]; \\
& & [\mathsf{free}] := t_1; \\
& & [\mathsf{free}+4] := t_2; \\
& & [x] := \mathsf{free}; \\
& & [\mathsf{scan}] := \mathsf{free}; \\
& & \\
& & \mathsf{FORW} := \mathsf{FORW} \ \oplus \ x; \\
& & \mathsf{UNFORW} := \mathsf{UNFORW} \ \ominus \ x; \\
& & \varphi := \varphi \oplus (x, \mathsf{free}); \\
& & \\
& & \mathsf{free} := \mathsf{free} \ + \ 8
\end{array}
$$

We will handle these in their order of difficulty in the next sections. But first, we note a lemma for later use.

Lemma 7.1. *(The pure part of) I implies* $\mathsf{free} \leq \mathsf{maxFree}$.

PROOF: By (53) and the fact that BUSY and NEW are defined as intervals with a common start-point, we can just show that $\#\mathsf{BUSY} \leq \#\mathsf{NEW}$. But this follows from

$$\#\mathsf{BUSY} = \#\mathsf{FORW} \leq \#\mathsf{ALIVE} \leq \#\mathsf{NEW}$$

The first equality follows from $\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY})$ and (15), the first inequality follows from $\mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE})$, (27), and (51). The last inequality is an explicit part of I. □

7.2.1 If Nothing Happens

In this section, we show that the specification (74) from before holds. According to the rule for **skip** and the rule of consequence, that amounts to the following

Proposition 7.2. *The assertion*

$$A \equiv I_x \wedge \neg\mathsf{Ptr}(x)$$

implies I'.

PROOF: We have

$$\begin{array}{l}
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \;\wedge\; \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \;\wedge \mathsf{scan} \leq \mathsf{free} \;\wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \neg\mathsf{Ptr}(x) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\mathsf{scan} \mapsto x) * (\exists q'.\, (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q')) \\
\Downarrow \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \;\wedge\; \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \;\wedge \mathsf{scan} \leq \mathsf{free} \;\wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\mathsf{scan} \mapsto x) * (\exists q'.\, (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q')) \\
\Downarrow \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \;\wedge\; \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \;\wedge \mathsf{scan} \leq \mathsf{free} \;\wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\mathsf{scan} \mapsto x \wedge (\mathsf{scan}, x) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger})) * \\
(\exists q'.\, (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q')) \\
\Downarrow \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \;\wedge\; \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \;\wedge \mathsf{scan} \leq \mathsf{free} \;\wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\mathsf{I}_{pure} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\exists q.\, (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge \mathsf{scan} \mapsto q) * \\
(\exists q'.\, (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge \mathsf{scan} + 4 \mapsto q')) \\
\Downarrow \\
I'
\end{array}$$

The first implication follows from the fact that A implies $(\mathsf{scan}, x) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger})$ by (17). The second implication follows from purity. □

7.2.2 If We do not Copy

In this section, we show the specification (75). The steps to do this are as follows. First, we show that the precondition implies $x \in \mathsf{FORW}$, and use this to infer $(\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger})$. Then we use a local specification to infer the desired global specification.

Lemma 7.3. *The assertion*

$$A \equiv I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW}$$

implies $x \in \mathsf{FORW}$.

PROOF: $(\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(x) \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE})$ implies $x \in \mathsf{ALIVE}$ by (41) and (40), so A implies $x \in \mathsf{ALIVE}$. By (23),

$$x \in \mathsf{ALIVE} \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \neg(x \in \mathsf{UNFORW}) \rightarrow x \in \mathsf{FORW},$$

so we assume $x \in \mathsf{UNFORW}$ and derive a contradiction, i.e., we show $A \wedge (x \in \mathsf{UNFORW}) \rightarrow \mathsf{F}$.

By (39), it suffices to show (since F is a monotone assertion)

$$\begin{array}{l} \mathsf{A}_{\mathsf{UNFORW}} \wedge (x \hookrightarrow y) \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \\ \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \wedge x \in \mathsf{UNFORW} \rightarrow \mathsf{F} \end{array}$$

We have

$$\begin{array}{l}
\mathsf{A}_{\mathsf{UNFORW}} \wedge x \hookrightarrow y \wedge \mathsf{PtrRg}(\mathsf{headALIVE}) \wedge \mathsf{Ptr}(y) \wedge \\
\mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \wedge x \in \mathsf{UNFORW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{UNFORW}-x} * (\exists q.\ (x,q) \in \mathsf{head} \wedge x \mapsto q) * (\exists q'.\ (x,q') \in \mathsf{tail} \wedge x+4 \mapsto q')) \wedge \\
x \hookrightarrow y \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{UNFORW}-x} * (x+4 \mapsto -) * (\exists q.\ (x,q) \in \mathsf{head} \wedge x \mapsto q)) \wedge \\
x \hookrightarrow y \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{UNFORW}-x} * (x+4 \mapsto -) * ((x,y) \in \mathsf{head} \wedge x \mapsto y)) \wedge \\
x \hookrightarrow y \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{UNFORW}-x} * (x+4 \mapsto -) * (x \mapsto y)) \wedge \\
(x,y) \in \mathsf{head} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
(x,y) \in \mathsf{head} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Ptr}(y) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
y \in \mathsf{ALIVE} \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge y \in \mathsf{NEW} \\
\Downarrow \\
\mathsf{F}
\end{array}$$

The first of these implications follows from (5), the third follows from (38), the fourth from purity, the sixth from (40), and the last follows from (28). □

Lemma 7.4. *The assertion A from Lemma 7.3 implies* $(\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)$

PROOF: We use Lemma 7.3 and show $A \wedge x \in \mathsf{FORW} \rightarrow (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)$. By (18),

$$(\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge A \wedge (x,y) \in \varphi \rightarrow (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger),$$

so it suffices to show

$$A \wedge x \in \mathsf{FORW} \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \rightarrow (x,y) \in \varphi$$

Like before, we use (39) and show

$$\mathsf{A}_{\mathsf{FORW}} \wedge x \hookrightarrow y \wedge x \in \mathsf{FORW} \rightarrow (x,y) \in \varphi$$

We have

$$
\begin{array}{l}
\mathsf{A}_{\mathsf{FORW}} \wedge x \hookrightarrow y \wedge x \in \mathsf{FORW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{FORW}\ominus x} * (\exists q.\ (x,q) \in \varphi \wedge x \mapsto q, -)) \wedge x \hookrightarrow y \wedge x \in \mathsf{FORW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{FORW}\ominus x} * ((x,y) \in \varphi \wedge x \mapsto y, -)) \wedge x \hookrightarrow y \wedge x \in \mathsf{FORW} \\
\Downarrow \\
(x,y) \in \varphi
\end{array}
$$

In this derivation, we have first used (5), then (65), and finally purity. □

We turn to the local specification for this branch of the program. It is simple:

$$
\begin{array}{l}
\{\mathsf{scan} \mapsto - \wedge (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)\} \\
\quad [\mathsf{scan}] := y \\
\{\mathsf{scan} \mapsto y \wedge (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)\} \\
\Downarrow \\
\{\exists q.\ (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto q\}
\end{array}
\tag{77}
$$

The first step follows from the rule (68) for heap update and the rule of conjunction by since the second conjunct is pure.

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

$$
\begin{array}{l}
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW}\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW} \wedge (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)\} \\
\Downarrow \\
\{\mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\mathsf{scan} \mapsto - \wedge (\mathsf{scan}, y) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)) * \\
(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto q'))\} \\
\\
\quad [\mathsf{scan}] := y \\
\\
\{\mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge y \in \mathsf{NEW} \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \\
\mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\exists q.\ \mathsf{scan} \mapsto q \wedge (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)) * \\
(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto q'))\} \\
\Downarrow \\
I'
\end{array}
$$

For the first implication, we use Lemma 7.4, and for the second, we use purity. The specification step follows from our local specification (77), and our derived version of the Frame Rule (69). This implies the desired global specification (75).

We are now ready to address the specification (76) for the most complicated branch of $\mathsf{ScanCar}$. The code in the the inner **else** branch resembles that of INIT^*, and consequently, the proof of its correctness will also be similar to that.

7.2.3 If We Copy

We show that the specification (76) from the branching analysis is derivable. We split $\mathsf{CopyCell}^*$ into two parts:

$$\begin{array}{llll}
\mathsf{CopyCell} \equiv & t_1 := [x]; & \mathsf{Increment} \equiv & \varphi := \varphi \oplus (x, \mathsf{free}) \\
& t_2 := [x+4]; & & \mathsf{FORW} := \mathsf{FORW} \oplus x; \\
& [\mathsf{free}] := t_1; & & \mathsf{UNFORW} := \mathsf{UNFORW} \ominus x; \\
& [\mathsf{free}+4] := t_2; & & \mathsf{free} := \mathsf{free} + 8; \\
& [x] := \mathsf{free}; & & \\
& [\mathsf{scan}] := \mathsf{free}; & &
\end{array}$$

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

Lemma 7.5. *The assertion*

$$A \equiv I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \in \mathsf{NEW})$$

implies $x \in \mathsf{UNFORW} \wedge \neg(x \in \mathsf{FORW})$.

PROOF: First,

$$\begin{array}{l}
I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \in \mathsf{NEW}) \\
\Downarrow \\
(\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \\
\mathsf{Ptr}(x) \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \\
\Downarrow \\
x \in \mathsf{ALIVE} \wedge \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \\
\Downarrow \\
x \in \mathsf{FORW} \vee x \in \mathsf{UNFORW}
\end{array}$$

The second implication follows from (40) and (41), and the third is by (22).

So, as before, we assume $x \in \mathsf{FORW}$ and derive a contradiction, i.e., we show that $(x \in \mathsf{FORW}) \wedge A \to \mathsf{F}$. By (39) and Lemma 7.1, the following derivation is sufficient to establish this.

$$\begin{array}{l}
\mathsf{A}_{\mathsf{FORW}} \wedge x \hookrightarrow y \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \wedge x \in \mathsf{FORW} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{FORW}-x} * (\exists q.\ (x,q) \in \varphi \wedge x \mapsto q, -)) \wedge \\
x \hookrightarrow y \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{FORW}-x} * (x+4 \mapsto -) * (\exists q.\ (x,q) \in \varphi \wedge x \mapsto q)) \wedge \\
x \hookrightarrow y \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
(\mathsf{A}_{\mathsf{FORW}-x} * (x+4 \mapsto -) * ((x,y) \in \varphi \wedge x \mapsto y)) \wedge \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow
\end{array}$$

$$\begin{array}{l}
(\mathsf{A}_{\mathsf{FORW}-x} * (x+4 \mapsto -) * (x \mapsto y)) \wedge \\
(x,y) \in \varphi \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
(x,y) \in \varphi \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
y \in \mathsf{BUSY} \wedge \neg(y \in \mathsf{NEW}) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
y \in \mathit{Itv}(\mathsf{offset}, \mathsf{free}) \wedge \neg(y \in \mathit{Itv}(\mathsf{offset}, \mathsf{maxFree})) \wedge \mathsf{free} \leq \mathsf{maxFree} \\
\Downarrow \\
\mathsf{F}
\end{array}$$

The first of these implications follows from (5), the second is a matter of notation. The third implication is an instance of (38), and the next comes from purity. The sixth implication in the derivation follows from (10), and the last is by elementary set manipulations (60). This shows the desired result. $\square$

We now turn to the local specification for $\mathsf{CopyCell}$. As usual, it only involves the locations that are involved in the program fragment.

$$\begin{array}{l}
\{(\exists q.\ (x,q) \in \mathsf{head} \wedge x \mapsto q) * (\exists q'.\ (x,q') \in \mathsf{tail} \wedge x+4 \mapsto q')* \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-)\} \\
\quad t_1 := [x] \\
\{((x,t_1) \in \mathsf{head} \wedge x \mapsto t_1) * (\exists q.\ (x,q') \in \mathsf{tail} \wedge x+4 \mapsto q')* \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-)\} \\
\quad t_2 := [x+4] \\
\{((x,t_1) \in \mathsf{head} \wedge x \mapsto t_1) * ((x,t_2) \in \mathsf{tail} \wedge x+4 \mapsto t_2)* \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-)\} \\
\quad [\mathsf{free}] := t_1 \\
\{((x,t_1) \in \mathsf{head} \wedge x \mapsto t_1) * ((x,t_2) \in \mathsf{tail} \wedge x+4 \mapsto t_2)* \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,-)\} \\
\quad [\mathsf{free}+4] := t_2 \\
\{((x,t_1) \in \mathsf{head} \wedge x \mapsto t_1) * ((x,t_2) \in \mathsf{tail} \wedge x+4 \mapsto t_2)* \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,t_2)\} \\
\Downarrow \\
\{((x \mapsto t_1) * (x+4 \mapsto t_2) * (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,t_2))\wedge \\
(x,t_1) \in \mathsf{head} \wedge (x,t_2) \in \mathsf{tail}\} \\
\quad [x] := \mathsf{free} \\
\{((x \mapsto \mathsf{free}) * (x+4 \mapsto t_2) * (\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto t_1,t_2))\wedge \\
(x,t_1) \in \mathsf{head} \wedge (x,t_2) \in \mathsf{tail}\} \\
\quad [\mathsf{scan}] := \mathsf{free} \\
\{((x \mapsto \mathsf{free}) * (x+4 \mapsto t_2) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1,t_2))\wedge \\
(x,t_1) \in \mathsf{head} \wedge (x,t_2) \in \mathsf{tail}\} \\
\Downarrow \\
\{((x \mapsto \mathsf{free},-) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1,t_2))\wedge \\
(x,t_1) \in \mathsf{head} \wedge (x,t_2) \in \mathsf{tail}\}
\end{array} \tag{78}$$

The implication in the middle of this derivation is due to pureness. The other implication is just a matter of notation, and the rest of the steps use the rules for lookup (66) and update (68), along with the Frame Rule.

We are now ready to infer the desired global specification for $\mathsf{CopyCell}^*$ via the specification (78).

$$\begin{array}{l}
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \in \mathsf{NEW})\} \\
\Downarrow \\
\{I_x \wedge \mathsf{Ptr}(x) \wedge (x \hookrightarrow y) \wedge \neg(y \in \mathsf{NEW}) \wedge x \in \mathsf{UNFORW} \wedge \neg(x \in \mathsf{FORW})\} \\
\Downarrow \\
\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Ptr}(x) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \\
\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge x \in \mathsf{UNFORW} \wedge \neg(x \in \mathsf{FORW}) \wedge \\
\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \forall e.\ \neg((x,e) \in \varphi) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}-x} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}})* \\
(\mathsf{scan} \mapsto -) * (\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan}+4 \mapsto q')* \\
(\exists q.\ (x,q) \in \mathsf{head} \wedge x \mapsto q) * (\exists q'.\ (x,q') \in \mathsf{tail} \wedge x+4 \mapsto q') * (\mathsf{free} \mapsto -,-))\}
\end{array}$$

CopyCell

$\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge$
$\mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Ptr}(x) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge$
$\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge x \in \mathsf{UNFORW} \wedge \neg(x \in \mathsf{FORW}) \wedge$
$\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \forall e.\ \neg((x, e) \in \varphi) \wedge$
$((\mathsf{A}_{\mathsf{UNFORW}-x} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) *$
$(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto q') *$
$((x \mapsto \mathsf{free}, -) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1, t_2)) \wedge$
$(x, t_1) \in \mathsf{head} \wedge (x, t_2) \in \mathsf{tail}\}$

$\Downarrow$

$\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge$
$\mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Ptr}(x) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge$
$\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ \varphi^\dagger \wedge x \in \mathsf{UNFORW} \wedge \neg(x \in \mathsf{FORW}) \wedge$
$(x, t_1) \in \mathsf{head} \wedge (x, t_2) \in \mathsf{tail} \wedge$
$\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(x, \mathsf{free}) \in \varphi \wedge$
$((\mathsf{A}_{\mathsf{UNFORW}-x} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) *$
$(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge \mathsf{scan} + 4 \mapsto q') *$
$((x \mapsto \mathsf{free}, -) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1, t_2)))\}$

$\varphi := \varphi \oplus (x, \mathsf{free});$
$\mathsf{UNFORW} := \mathsf{UNFORW} \ominus x;$
$\mathsf{FORW} := \mathsf{FORW} \oplus x$

$\{\mathsf{iso}(\varphi \ominus (x, \mathsf{free}), \mathsf{FORW} \ominus x, \mathsf{BUSY}) \ \wedge \ \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge$
$\mathsf{isUnion}(\mathsf{FORW} \ominus x, \mathsf{UNFORW} \oplus x, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge$
$\mathsf{root} \in \mathsf{FORW} \ominus x \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Ptr}(x) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge$
$\neg(\mathsf{scan} = \mathsf{free}) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger \wedge x \in \mathsf{FORW} \wedge$
$\neg(x \in \mathsf{UNFORW}) \wedge (x, t_1) \in \mathsf{head} \wedge (x, t_2) \in \mathsf{tail} \wedge (x, \mathsf{free}) \in \varphi \wedge$
$((\forall_* p \in ((\mathsf{UNFORW} \oplus x) \ominus x).\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) *$
$(\forall_* p \in (\mathsf{FORW} \ominus x).\ (\exists q.\ (p, q) \in (\varphi \ominus (x, \mathsf{free})) \wedge p \mapsto q, -)) *$
$(\forall_* p \in \mathsf{FIN}.\ ((\exists q.\ (p, q) \in (\varphi \ominus (x, \mathsf{free})) \circledcirc (\mathsf{head} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger) \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in \varphi \ominus (x, \mathsf{free}) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger) \wedge p + 4 \mapsto q'))) *$
$(\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{scan}).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger) \wedge p + 4 \mapsto q')) *$
$(\forall_* p \in (\mathsf{FREE} \ominus \mathsf{free}).\ p \mapsto -, -) *$
$(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free}))^\dagger) \wedge \mathsf{scan} + 4 \mapsto q') *$
$(x \mapsto \mathsf{free}, -) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto t_1, t_2))\}$

$\mathsf{free} := \mathsf{free} + 8$

$\{\mathsf{iso}(\varphi \ominus (x, \mathsf{free} - 8), \mathsf{FORW} \ominus x, \mathsf{BUSY} \ominus (\mathsf{free} - 8)) \ \wedge \ \mathsf{Ptr}(\mathsf{free} - 8) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge$
$\mathsf{isUnion}(\mathsf{FORW} \ominus x, \mathsf{UNFORW} \oplus x, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge$
$\mathsf{root} \in \mathsf{FORW} \ominus x \ \wedge \mathsf{scan} \leq \mathsf{free} - 8 \ \wedge \mathsf{Ptr}(x) \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge$
$\neg(\mathsf{scan} = \mathsf{free} - 8) \wedge (\mathsf{scan}, x) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger \wedge x \in \mathsf{FORW} \wedge$
$\neg(x \in \mathsf{UNFORW}) \wedge (x, t_1) \in \mathsf{head} \wedge (x, t_2) \in \mathsf{tail} \wedge (x, \mathsf{free} - 8) \in \varphi \wedge$
$((\forall_* p \in ((\mathsf{UNFORW} \oplus x) \ominus x).\ ((\exists q.\ (p, q) \in \mathsf{head} \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in \mathsf{tail} \wedge p + 4 \mapsto q'))) *$
$(\forall_* p \in (\mathsf{FORW} \ominus x).\ (\exists q.\ (p, q) \in (\varphi \ominus (x, \mathsf{free} - 8)) \wedge p \mapsto q, -)) *$
$(\forall_* p \in \mathsf{FIN}.\ ((\exists q.\ (p, q) \in \varphi \ominus (x, \mathsf{free} - 8) \circledcirc (\mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in (\varphi \ominus (x, \mathsf{free} - 8)) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q'))) *$
$(\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger \wedge p \mapsto q) *$
$\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')) *$
$(\forall_* p \in ((\mathsf{FREE} \oplus (\mathsf{free} - 8)) \ominus (\mathsf{free} - 8)).\ p \mapsto -, -) *$
$(\exists q'.\ (\mathsf{scan}, q') \in \mathsf{tail} \circ (\varphi \ominus (x, (\mathsf{free} - 8)))^\dagger) \wedge \mathsf{scan} + 4 \mapsto q') *$
$(x \mapsto (\mathsf{free} - 8), -) * (\mathsf{scan} \mapsto (\mathsf{free} - 8)) * ((\mathsf{free} - 8) \mapsto t_1, t_2))\}$

The first implication follows from Lemma 7.5. The second follows from (5) and (16). The specification for $\mathsf{CopyCell}$ follows from our local specification (78) and (69). The implication immediately thereafter is a consequence of purity. The specification for the three auxiliary variables follows from the rules in (71), and the specification for the update of free follows from the rules for assignment and the rules in (72).

Like in Section 7.1, 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 x). Specifically, the pure part of I' follows from the pure part of the conclusion above by exactly the same argument as in Section 7.1, and the same argument goes for the separating conjunction over the sets FORW, UNFORW, FIN, and FREE, and for the location $\mathsf{scan} + 4$. Thus, if we let B be the pure part of the conclusion above, what is left to show is that

$$\begin{aligned}&B \wedge (((\mathsf{free} - 8) \mapsto t_1, t_2) * (\mathsf{scan} \mapsto \mathsf{free} - 8)*\\&(\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)).\\&((\exists q.\ (p, q) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger \wedge p \mapsto q)*\\&\quad(\exists q'.\ (p, q') \in \mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')))\end{aligned} \tag{79}$$

implies

$$\begin{aligned}&(\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{scan}).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q)*\\&\quad(\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')))*\\&(\exists q.\ (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge \mathsf{scan} \mapsto q)\end{aligned} \tag{80}$$

and that

$$\begin{aligned}&B \wedge\\&(\forall_* p \in \mathsf{FIN}.\\&((\exists q.\ (p, q) \in \varphi \ominus (x, \mathsf{free} - 8) \circledcirc (\mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p \mapsto q)*\\&(\exists q'.\ (p, q') \in (\varphi \ominus (x, \mathsf{free} - 8)) \circledcirc (\mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')))\end{aligned} \tag{81}$$

implies

$$\begin{aligned}&B \wedge\\&(\forall_* p \in \mathsf{FIN}.\ ((\exists q.\ (p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge p \mapsto q)*\\&\quad(\exists q'.\ (p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^\dagger) \wedge p + 4 \mapsto q')))\end{aligned} \tag{82}$$

The last follows from ordinary set manipulations (47) and Lemma 3.7.

For the first, we have

$$
\begin{array}{l}
(x, t_1) \in \mathsf{head} \wedge (x, t_2) \in \mathsf{tail} \wedge (x, \mathsf{free} - 8) \in \varphi \wedge \\
(\mathsf{scan}, x) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger \wedge \mathsf{Ptr}(x) \wedge \\
((\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)). \\
((\exists q.\, (p, q) \in \mathsf{head} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\, (p, q') \in \mathsf{tail} \circ (\varphi \ominus (x, \mathsf{free} - 8))^\dagger) \wedge p + 4 \mapsto q')) * \\
(\mathsf{scan} \mapsto \mathsf{free} - 8) * (\mathsf{free} - 8 \mapsto t_1, t_2)) \\
\Downarrow \\
(\mathsf{free} - 8, t_1) \in \mathsf{head} \circ \varphi^\dagger \wedge \\
(\mathsf{free} - 8, t_2) \in \mathsf{tail} \circ \varphi^\dagger \wedge (\mathsf{scan}, \mathsf{free} - 8) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge \\
(\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)). \\
((\exists q.\, (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\, (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')) * \\
(\mathsf{scan} \mapsto \mathsf{free} - 8) * ((\mathsf{free} - 8) \mapsto t_1) * ((\mathsf{free} - 8) + 4 \mapsto t_2)) \\
\Downarrow \\
(\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)). \\
((\exists q.\, (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\, (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')) * \\
(\mathsf{scan} \mapsto \mathsf{free} - 8 \wedge (\mathsf{scan}, \mathsf{free} - 8) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)) * \\
((\mathsf{free} - 8) \mapsto t_1 \ \wedge (\mathsf{free} - 8, t_1) \in \mathsf{head} \circ \varphi^\dagger) * \\
((\mathsf{free} - 8) + 4 \mapsto t_2 \wedge (\mathsf{free} - 8, t_2) \in \mathsf{tail} \circ \varphi^\dagger)) \\
\Downarrow \\
(\forall_* p \in ((\mathsf{UNFIN} \ominus \mathsf{scan}) \ominus (\mathsf{free} - 8)). \\
((\exists q.\, (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\, (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')) * \\
(\exists q.\, \mathsf{scan} \mapsto q \wedge (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)) * \\
(\exists q.\, (\mathsf{free} - 8) \mapsto q \wedge (\mathsf{free} - 8, q) \in \mathsf{head} \circ \varphi^\dagger) * \\
(\exists q'.\, (\mathsf{free} - 8) + 4 \mapsto q' \wedge (\mathsf{free} - 8, q') \in \mathsf{tail} \circ \varphi^\dagger)) \\
\Downarrow \\
(\forall_* p \in (\mathsf{UNFIN} \ominus \mathsf{scan}). \\
((\exists q.\, (p, q) \in \mathsf{head} \circ \varphi^\dagger \wedge p \mapsto q) * \\
\quad (\exists q'.\, (p, q') \in \mathsf{tail} \circ \varphi^\dagger \wedge p + 4 \mapsto q')) * \\
(\exists q.\, \mathsf{scan} \mapsto q \wedge (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger))
\end{array}
$$

The first implication follows from (47), ordinary composition of relations (59), the rule for the special relation composition $\circledcirc$ (18), and from Lemma 3.7. The second implication follows from purity, and the last implication from (6).

We have therefore obtained (79) $\Rightarrow$ (80), and we can conclude that the first part of the specification (73) 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.

7.2.4 After ScanCdr

In this subsection, 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 completely analogous to that of ScanCar. Following the lines in that proof, one obtains the specification

$$
\begin{array}{l}
\{I'\} \\
\quad \mathsf{ScanCdr} \\
\{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\exists q.\ (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
(\exists q'.\ (\mathsf{scan}, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p \mapsto q'))\}
\end{array}
$$

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

$$
\begin{array}{l}
\{A\} \\
\quad \mathsf{scan} := \mathsf{scan} + 8 \\
\{I\}
\end{array}
$$

holds. By the rule for assignment and the rules (72) for intervals, we get

$$
\begin{array}{l}
\quad \{A\} \\
\Downarrow \\
\quad \{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\quad \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\quad \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \neg(\mathsf{scan} = \mathsf{free}) \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FREE}}) * \\
\quad (\forall_{*} p \in \mathsf{FIN}.\ ((\exists q.\ (p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p + 4 \mapsto q'))) * \\
\quad (\forall_{*} p \in (\mathsf{UNFIN} \ominus \mathsf{scan}).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge p + 4 \mapsto q'))) * \\
\quad (\exists q.\ (\mathsf{scan}, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (\mathsf{scan}, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p \mapsto q'))\} \\
\quad\quad \mathsf{scan} := \mathsf{scan} + 8 \\
\quad \{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\quad \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} - 8 \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\quad \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan} - 8) \wedge \neg(\mathsf{scan} - 8 = \mathsf{free}) \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} \mathsf{A}_{\mathsf{FREE}}) * \\
\quad (\forall_{*} p \in (\mathsf{FIN} \ominus (\mathsf{scan} - 8)).\ ((\exists q.\ (p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p + 4 \mapsto q'))) * \\
\quad (\forall_{*} p \in ((\mathsf{UNFIN} \oplus (\mathsf{scan} - 8)) \ominus \mathsf{scan} - 8).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge p + 4 \mapsto q'))) * \\
\quad (\exists q.\ (\mathsf{scan} - 8, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (\mathsf{scan} - 8, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p \mapsto q'))\} \\
\Downarrow \\
\quad \{\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge\ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \\
\quad \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} - 8 \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \\
\quad \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan} - 8) \wedge \neg(\mathsf{scan} - 8 = \mathsf{free}) \wedge \\
\quad ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} \mathsf{A}_{\mathsf{FREE}}) * \\
\quad ((\forall_{*} p \in (\mathsf{FIN} \ominus (\mathsf{scan} - 8)).\ ((\exists q.\ (p, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p + 4 \mapsto q'))) * \\
\quad (\exists q.\ (\mathsf{scan} - 8, q) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge p \mapsto q) * \\
\quad (\exists q'.\ (\mathsf{scan} - 8, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge p \mapsto q')) * \\
\quad (\forall_{*} p \in ((\mathsf{UNFIN} \oplus (\mathsf{scan} - 8)) \ominus \mathsf{scan} - 8).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge p \mapsto q) * \\
\quad\quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^{\dagger} \wedge p + 4 \mapsto q'))))\}
\end{array}
$$

The first step uses (5), and the specification uses the rule for assignment and those in (72). The last step is a simple rewriting.

Like in Section 7.1 and 7.2.3, 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 $\mathsf{Ptr}(\mathsf{scan}) \wedge \mathsf{scan} \leq \mathsf{free}$. But this follows from $\mathsf{Ptr}(\mathsf{scan} - 8) \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \neg(\mathsf{scan} - 8 = \mathsf{free}) \wedge \mathsf{scan} - 8 \leq \mathsf{free}$.

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

$$\begin{array}{l} I''_p \wedge \\ (\forall_* p \in ((\mathsf{UNFIN} \oplus (\mathsf{scan} - 8)) \ominus \mathsf{scan}).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge p \mapsto q) * \\ \quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^{\dagger}) \wedge p + 4 \mapsto q')) \end{array}$$

implies

$$\begin{array}{l} \forall_* p \in ((\mathsf{UNFIN} \oplus (\mathsf{scan} - 8)) \ominus \mathsf{scan}).\ ((\exists q.\ (p, q) \in \mathsf{head} \circ \varphi^{\dagger} \wedge p \mapsto q) * \\ \quad (\exists q'.\ (p, q') \in \mathsf{tail} \circ \varphi^{\dagger}) \wedge p + 4 \mapsto q') \end{array}$$

and that

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

implies

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

For the first of these, the implication follows from (1), Lemma 3.7, and (47), since $\neg((\mathsf{scan} - 8) \in \mathsf{UNFIN})$ implies $(\mathsf{UNFIN} \oplus (\mathsf{scan} - 8)) \ominus (\mathsf{scan} - 8) = \mathsf{UNFIN}$ (recall $\mathsf{UNFIN} \equiv Itv(\mathsf{scan}, \mathsf{free})$). The second implication follows from Lemma 3.7, (47), and (6), since $\mathsf{scan} - 8 \in \mathsf{FIN} \equiv Itv(\mathsf{offset}, \mathsf{scan})$.

This means that the specification (73) holds. In order to conclude correctness of the garbage collector, we show that I and the fact that we exit the **while**-loop imply that the heaps before and after execution of the garbage collector are heap-isomorphic in the sense of Definition 5.2.

8 Sufficiency of the Invariant

In this section, we will prove that the invariant I is strong enough to conclude that the relation φ can be viewed as a weak isomorphism between the initial heap and the heap after execution of the garbage collector.

The overall strategy is this. We first show that all cells that were reachable before execution are moved from UNFORW into FORW by the algorithm, so we can conclude that $\mathsf{ALIVE} = \mathsf{FORW}$ after the execution. This fact is then used to establish a correspondence between φ, head, and our "special composition" $\circledcirc$. This will be the key to show that after execution, the denotation of φ is indeed a weak heap isomorphism.

Lemma 8.1.

$$I \wedge \mathsf{scan} = \mathsf{free} \wedge \exists P^{\mathsf{path}}.\ \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, p) \to p \in \mathsf{FORW}$$

PROOF: We show that

$$\forall P.\ \forall p.\ (\mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, p)) \to ((I \wedge \mathsf{scan} = \mathsf{free}) \to p \in \mathsf{FORW})$$

by the induction principle (36) on evaluation of paths. We just show

$$\begin{array}{ll} \text{(B)} & \mathsf{eval}(\mathsf{head}, \mathsf{tail}, \varepsilon, \mathsf{root}, p) \wedge I \wedge \mathsf{scan} = \mathsf{free} \to p \in \mathsf{FORW} \\ \text{(IS)} & (\forall \overline{p}.\ \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, \overline{p}) \wedge I \wedge \mathsf{scan} = \mathsf{free} \to \overline{p} \in \mathsf{FORW}) \wedge \\ & \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P \cdot \mathsf{head}, \mathsf{root}, p) \wedge I \wedge (\mathsf{scan} = \mathsf{free}) \to p \in \mathsf{FORW} \end{array}$$

For (B), we have

$$
\begin{array}{l}
\mathsf{eval}(\mathsf{head}, \mathsf{tail}, \varepsilon, \mathsf{root}, p) \wedge I \wedge \mathsf{scan} = \mathsf{free} \\
\Downarrow \\
\mathsf{root} = p \wedge I \wedge \mathsf{scan} = \mathsf{free} \\
\Downarrow \\
\mathsf{root} = p \wedge \mathsf{root} \in \mathsf{FORW} \\
\Downarrow \\
p \in \mathsf{FORW}
\end{array}
$$

The first implication follows from (32).

For (IS), we will introduce some shorthand notations. First, we let I_p be the pure part of I:

$$
\begin{array}{l}
I_p \equiv \\
\mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \ \wedge \ \mathsf{isUnion}(\mathsf{FORW}, \mathsf{UNFORW}, \mathsf{ALIVE}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge \\
\mathsf{root} \in \mathsf{FORW} \ \wedge \mathsf{scan} \leq \mathsf{free} \ \wedge \mathsf{Disjoint}(\mathsf{ALIVE}, \mathsf{NEW}) \wedge \mathsf{I}_{pure} \wedge \mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan})
\end{array}
$$

Second, there will be a variable q with $q \in \mathsf{FIN}$ involved in our proof. Therefore, we define

$$
\begin{array}{l}
I_q \equiv \\
(\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}-q} * \mathsf{A}_{\mathsf{UNFIN}} * \mathsf{A}_{\mathsf{FREE}}) * \\
(\exists q'.\, (q, q') \in \varphi \circledcirc (\mathsf{tail} \circ \varphi^{\dagger}) \wedge q + 4 \mapsto q')
\end{array}
$$

Then we have

$$
\begin{array}{l}
(\forall \overline{p}.\, \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, \overline{p}) \wedge I \wedge \mathsf{scan} = \mathsf{free} \to \overline{p} \in \mathsf{FORW}) \wedge \\
(\mathsf{eval}(\mathsf{head}, \mathsf{tail}, P \cdot \mathsf{head}, \mathsf{root}, p) \wedge I \wedge \mathsf{scan} = \mathsf{free}) \\
\Downarrow \\
(\forall \overline{p}.\, \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, \overline{p}) \wedge I \wedge \mathsf{scan} = \mathsf{free} \to \overline{p} \in \mathsf{FORW}) \wedge \\
(\exists p'.\, \mathsf{Ptr}(p') \wedge \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, p') \wedge (p', p) \in \mathsf{head} \wedge I \wedge \mathsf{scan} = \mathsf{free}) \\
\Downarrow \\
(\exists p'.\, \mathsf{Ptr}(p') \wedge p' \in \mathsf{FORW} \wedge (p', p) \in \mathsf{head} \wedge I \wedge \mathsf{scan} = \mathsf{free}) \\
\Downarrow \\
\exists p'.\, \exists q.\, p' \in \mathsf{FORW} \wedge q \in \mathsf{BUSY} \wedge I \wedge \mathsf{scan} = \mathsf{free} \wedge (p', p) \in \mathsf{head} \wedge \\
\mathsf{Ptr}(p') \wedge (p', q) \in \varphi \wedge \mathsf{BUSY} = \mathsf{FIN} \\
\Downarrow \\
\exists p'.\, \exists q.\, p' \in \mathsf{FORW} \wedge q \in \mathsf{FIN} \wedge I \wedge \mathsf{scan} = \mathsf{free} \wedge (p', p) \in \mathsf{head} \wedge \\
\mathsf{Ptr}(p') \wedge (p', q) \in \varphi \\
\Downarrow \\
\exists p'.\, \exists q.\, p' \in \mathsf{FORW} \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\
(I_q * (\exists r.\, (q, r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge q \mapsto r)) \wedge \\
\mathsf{scan} = \mathsf{free} \wedge (p', p) \in \mathsf{head} \wedge \mathsf{Ptr}(p') \wedge (p', q) \in \varphi \\
\Downarrow
\end{array}
$$

$$
\begin{array}{l}
\exists p'.\, \exists q.\, p' \in \mathsf{FORW} \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\
(I_q * (\exists r.\, (q, r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger})) \wedge q \mapsto r) \wedge \\
\mathsf{scan} = \mathsf{free} \wedge (p', p) \in \mathsf{head} \wedge \mathsf{Ptr}(p') \wedge (p', q) \in \varphi \wedge (q, p) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \\
\mathsf{Tfun}(\mathsf{head} \circ \varphi^{\dagger}, \mathsf{BUSY}) \\
\Downarrow \\
\exists p'.\, \exists q.\, p' \in \mathsf{FORW} \wedge q \in \mathsf{FIN} \wedge I_p \wedge \exists r.\, (q, r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^{\dagger}) \wedge (I_q * (q \mapsto -)) \wedge \\
\mathsf{scan} = \mathsf{free} \wedge (p', p) \in \mathsf{head} \wedge \mathsf{Ptr}(p') \wedge (p', q) \in \varphi \wedge (q, p) \in \mathsf{head} \circ \varphi^{\dagger} \wedge \\
\mathsf{Tfun}(\mathsf{head} \circ \varphi^{\dagger}, \mathsf{BUSY}) \\
\Downarrow \\
\exists r.\, (p, r) \in \varphi \ \wedge \mathsf{iso}(\varphi, \mathsf{FORW}, \mathsf{BUSY}) \\
\Downarrow \\
p \in \mathsf{FORW}
\end{array}
$$

The first implication uses the rule (33) for eval, the third uses one of the rules for isos (12) and the definition of BUSY and FIN as intervals, the fifth uses our usual splitting of iterated separating conjunctions (5). Implication number six uses elementary set manipulations (58), (59), and rules for isomorphisms and total functions (13), (11). The seventh implication uses purity, and the last-but-one uses the rules for $\circledcirc$ (19). Finally, the last implication takes advantage of (10). This proves the lemma. $\square$

We can now argue that upon exit from the **while** loop, $\mathsf{ALIVE} = \mathsf{FORW}$.

Lemma 8.2.

$$I \wedge \mathsf{scan} = \mathsf{free} \rightarrow \mathsf{FORW} = \mathsf{ALIVE}$$

PROOF: First, I implies $\mathsf{SbSet}(\mathsf{FORW}, \mathsf{ALIVE})$ by (27). By (54), it then suffices to show $I \wedge \mathsf{scan} = \mathsf{free} \rightarrow \mathsf{SbSet}(\mathsf{ALIVE}, \mathsf{FORW})$. To this end, we can use (30) and show that for all p, $I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \rightarrow p \in \mathsf{FORW}$. We have

$$\begin{array}{l}
I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \\
\Downarrow \\
p \in \mathsf{ALIVE} \wedge \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge I \wedge \mathsf{scan} = \mathsf{free} \\
\Downarrow \\
\exists P.\ \mathsf{eval}(\mathsf{head}, \mathsf{tail}, P, \mathsf{root}, p) \wedge I \wedge \mathsf{scan} = \mathsf{free} \\
\Downarrow \\
p \in \mathsf{FORW}
\end{array}$$

The second implication here uses the rules for $\mathsf{Reachable}$ (35), and the third uses Lemma 8.1. $\square$

The following theorem is the key to showing that our invariant I is strong enough to conclude that φ is a weak heap isomorphism.

Theorem 8.3.

$$I \wedge \mathsf{scan} = \mathsf{free} \rightarrow (p \in \mathsf{ALIVE} \wedge (p, q) \in \varphi \rightarrow (q \hookrightarrow r \leftrightarrow (p, r) \in \varphi \circledcirc \mathsf{head}))$$

PROOF: Let

$$\begin{array}{l}
I_1 \equiv I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (p, q) \in \varphi \wedge q \hookrightarrow r, \\
I_2 \equiv I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (p, q) \in \varphi \wedge (p, r) \in \varphi \circledcirc \mathsf{head}.
\end{array}$$

We then show that $I_1 \rightarrow (p, r) \in \varphi \circledcirc \mathsf{head}$ and $I_2 \rightarrow q \hookrightarrow r$. For the first of these, we have

$$
\begin{array}{l}
\quad I_1 \\
\Downarrow \\
\quad I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \hookrightarrow r \\
\Downarrow \\
\quad I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \hookrightarrow r \wedge q \in \mathsf{BUSY} \\
\Downarrow \\
\quad I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \hookrightarrow r \wedge q \in \mathsf{FIN} \\
\Downarrow \\
\quad \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \hookrightarrow r \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\
\quad (I_q * (\exists q'.\, (q,q') \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge q \mapsto q')) \\
\Downarrow \\
\quad \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\
\quad (I_q * ((q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge q \mapsto r)) \\
\Downarrow \\
\quad \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\
\quad (q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge (I_q * (q \mapsto r)) \\
\Downarrow \\
A \left\{ \begin{array}{l} (\mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\ (q,r) \in \mathsf{head} \circ \varphi^\dagger \wedge \neg \mathsf{Ptr}(r) \wedge (I_q * (q \mapsto r))) \end{array} \right. \\
\quad \vee \\
B \left\{ \begin{array}{l} (\mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{FORW} \wedge (p,q) \in \varphi \wedge q \in \mathsf{FIN} \wedge I_p \wedge \\ \exists p'.\, ((q,p') \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(p') \wedge (p',r) \in \varphi) \wedge (I_q * (q \mapsto r))) \end{array} \right.
\end{array}
$$

We have used the same I_p and I_q as in the proof of Lemma 8.1. The first implication follows from Lemma 8.2, the second follows from (10). The third implication follows from the definitions of FIN and BUSY and $\mathsf{scan} = \mathsf{free}$. The fourth follows from (5), and the fifth follows from single-valuedness (38). The sixth is a consequence of purity, and the last follows from the rules for $\circledcirc$ (20).

We now have to show that each of the two parts of the disjunction $A \vee B$ in the conclusion above implies $(p,r) \in \varphi \circledcirc \mathsf{head}$. For A,

$$
\begin{array}{l}
\quad A \\
\Downarrow \\
\quad (p,q) \in \varphi \wedge (q,r) \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(p) \wedge \neg \mathsf{Ptr}(r) \\
\Downarrow \\
\quad (p,r) \in \mathsf{head} \circ \varphi^\dagger \circ \varphi \wedge \neg \mathsf{Ptr}(r) \\
\Downarrow \\
\quad (p,r) \in \mathsf{head} \wedge \neg \mathsf{Ptr}(r) \\
\Downarrow \\
\quad (p,r) \in \varphi \circledcirc \mathsf{head}
\end{array}
$$

The first and second steps here use elementary set theory (52), (59). The third uses that the application of an isomorphism with its transpose yields the identity (55), and the last step uses the rules for $\circledcirc$ (17). For B,

$$
\begin{array}{l}
\quad B \\
\Downarrow \\
\quad \exists p'.\, \mathsf{Ptr}(p') \wedge (q,p') \in \mathsf{head} \circ \varphi^\dagger \wedge (p,q) \in \varphi \wedge \mathsf{Ptr}(p) \wedge (p',r) \in \varphi \\
\Downarrow \\
\quad \exists p'.\, (p,p') \in \mathsf{head} \circ \varphi^\dagger \circ \varphi \wedge (p',r) \in \varphi \wedge \mathsf{Ptr}(p') \\
\Downarrow \\
\quad \exists p'.\, (p,p') \in \mathsf{head} \wedge \mathsf{Ptr}(p') \wedge (p',r) \in \varphi \\
\Downarrow \\
\quad (p,r) \in \varphi \circledcirc \mathsf{head}
\end{array}
$$

The first step here uses the fact that sets only contain pointers (52), the second step uses the rule for ordinary composition of relations (59), and again we note the identity yielded by composing an isomorphism with its transpose in the third step (55). The last implication os due to (18). This establishes $I_1 \to (p,r) \in \varphi \circledcirc \mathsf{head}$.

For the other half, we start with the following implication which follows from (20).

$$
\begin{array}{ll}
 & I_2 \\
 & \Downarrow \\
A\,\{ & (I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (p,q) \in \varphi \wedge (p,r) \in \mathsf{head} \wedge \neg \mathsf{Ptr}(r)) \\
 & \vee \\
B\,\{ & (I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (p,q) \in \varphi \wedge \exists p'.\, \mathsf{Ptr}(p') \wedge (p,p') \in \mathsf{head} \wedge (p',r) \in \varphi)
\end{array}
$$

Again, we have to do a case study, i.e., show that each of the parts in the disjunction $A \vee B$ above implies $q \hookrightarrow r$. We have

$$
\begin{array}{l}
A \\
\Downarrow \\
I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (q,p) \in \varphi^\dagger \wedge (p,r) \in \mathsf{head} \wedge \neg \mathsf{Ptr}(r) \wedge q \in \mathsf{BUSY} \\
\Downarrow \\
I \wedge \mathsf{scan} = \mathsf{free} \wedge (q,r) \in \mathsf{head} \circ \varphi^\dagger \wedge \neg \mathsf{Ptr}(r) \wedge q \in \mathsf{FIN} \\
\Downarrow \\
I \wedge \mathsf{scan} = \mathsf{free} \wedge (q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge q \in \mathsf{FIN} \\
\Downarrow \\
(I_q * (\exists \overline{p}.\, (q,\overline{p}) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge q \mapsto \overline{p})) \wedge \\
I_p \wedge \mathsf{scan} = \mathsf{free} \wedge (q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \\
\Downarrow \\
\exists \overline{p}.\, (I_q * (q \mapsto \overline{p})) \wedge I_p \wedge \mathsf{scan} = \mathsf{free} \wedge \\
(q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge (q,\overline{p}) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \\
\Downarrow \\
\exists \overline{p}.\, (I_q * q \mapsto \overline{p}) \wedge r = \overline{p} \\
\Downarrow \\
I_q * q \mapsto r \\
\Downarrow \\
q \hookrightarrow r
\end{array}
$$

The first implication here follows from elementary set theory (58), and from (10). The second follows from the definition of FIN and BUSY, from $\mathsf{scan} = \mathsf{free}$, and from (59). The third implication is a consequence of the rule (17). The fourth implication follows from the usual splitting of iterated separating conjunctions (5), and the fifth from purity. The sixth follows from the rules for $\circledcirc$ (21), and the last implication is just a matter of notation.

Finally,

B
$\Downarrow$
$\exists p'.\, I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (q,p) \in \varphi^\dagger \wedge \mathsf{Ptr}(p') \wedge$
$(p,p') \in \mathsf{head} \wedge (p',r) \in \varphi \wedge q \in \mathsf{BUSY}$
$\Downarrow$
$\exists p'.\, I \wedge q \in \mathsf{FIN} \wedge (q,p') \in \mathsf{head} \circ \varphi^\dagger \wedge \mathsf{Ptr}(p') \wedge (p',r) \in \varphi$
$\Downarrow$
$\exists p'.\, (I_q * (\exists \overline{p}.\, (q,\overline{p}) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge q \mapsto \overline{p})) \wedge I_p \wedge (q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)$
$\Downarrow$
$\exists p', \overline{p}.\, (I_q * q \mapsto \overline{p}) \wedge I_p \wedge (q,r) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger) \wedge (q,\overline{p}) \in \varphi \circledcirc (\mathsf{head} \circ \varphi^\dagger)$
$\Downarrow$
$(I_q * q \mapsto \overline{p}) \wedge r = \overline{p}$
$\Downarrow$
$I_q * q \mapsto r$
$\Downarrow$
$q \hookrightarrow r$

This derivation follows the same pattern as the previous one. This concludes the proof of the theorem. □

Corollary 8.4. *Suppose s, h is a state satisfying the precondition* InitAss *mentioned in Section 6.2.1, and assume $GC^*, s, h \rightsquigarrow s', h'$ Then $s'(\varphi)$ is a weak heap isomorphism between the part of h whose domain is $s(\mathsf{ALIVE})$ and the part of h' whose domain is $s'(\mathsf{FORW})$.*

PROOF: This proof will take place at the semantical level. Write β for $s'(\varphi)$ (and note that I implies that β is a bijection $\beta : s'(\mathsf{FORW}) = s'(\mathsf{ALIVE}) = s(\mathsf{ALIVE}) \overset{\sim}{\to} s'(\mathsf{BUSY})$), hd for $s(\mathsf{head}) = s'(\mathsf{head})$, tl for $s(\mathsf{tail}) = s'(\mathsf{tail})$, and al for $s(\mathsf{ALIVE}) = s'(\mathsf{ALIVE})$ (and note that $h(p) = hd(p)$ and $h(p+4) = tl(p)$ for all $p \in al$). What we need to show is that for all $p \in al$,

$$h'(\beta(p)) = \beta^*(h(p)) \quad (83)$$
$$h'(\beta(p)+4) = \beta^*(h(p+4)) \quad (84)$$

We just show (83). If $h'(\beta(p)) = r$, there is a q such that

$$s', h' \Vdash I \wedge \mathsf{scan} = \mathsf{free} \wedge p \in \mathsf{ALIVE} \wedge (p,q) \in \varphi \wedge q \hookrightarrow r$$

By Theorem 8.3, this means $s', h' \Vdash (p,r) \in \varphi \circledcirc \mathsf{head}$, and this means $\beta^*(hd(p)) = r$ and since $hd(p) = h(p)$, we get $h'(\beta(p)) = r$ implies $\beta^*(h(p)) = r$, so $h'(\beta(p)) = \beta^*(h(p))$, as desired. □

We are now in a position to take the last step and conclude that our garbage collector is correct. Intuitively, we can just use Corollary 8.4 and instantiate ALIVE with $dom(prune(h))$ if (s,h) in the state in which we run the algorithm (the result in Corollary 8.4 is valid for all values of ALIVE). To formalize this, we need a precise formulation of pruning. Thus, we define an auxiliary reachability predicate which depends on the heap (and not two relations). This, in turn, is most easily done by introducing a new notion of evaluation of paths that only depends on the heap.

Definition 8.5. We add the assertion forms

$$\mathsf{Pairheap}, \quad \mathsf{eval}'(P^{\mathsf{path}}, e_1^{\mathsf{int}}, e_2^{\mathsf{int}}), \quad \text{and} \quad \mathsf{Reachable}'(m^{\mathsf{fs}}, e^{\mathsf{int}})$$

to the assertion language of Section 3.3, and their semantics is given by the following clauses:

$$
\begin{array}{lll}
s,h & \Vdash & \mathsf{Pairheap}\ \text{iff} \\
& & \quad \forall p \in dom(h). \\
& & \qquad p \bmod 8 = 0 \rightarrow p+4 \in dom(h) \wedge \\
& & \qquad p \bmod 8 = 4 \rightarrow p-4 \in dom(h) \\
s,h & \Vdash & \mathsf{eval}'(P^{\mathsf{path}}, p^{\mathsf{int}}, e^{\mathsf{int}})\ \text{iff} \\
& & \quad (P = \varepsilon \text{ and } s,h \Vdash p = e),\ \text{or} \\
& & \quad (P = P' \cdot \mathsf{head} \text{ and } \exists p' \in \mathsf{Ptr}.\ s,h \Vdash \mathsf{eval}'(P',p,p') \\
& & \qquad \text{and } s,h \Vdash p' \hookrightarrow e),\ \text{or} \\
& & \quad (P = P' \cdot \mathsf{tail} \text{ and } \exists p' \in \mathsf{Ptr}.\ s,h \Vdash \mathsf{eval}'(P',p,p') \\
& & \qquad \text{and } s,h \Vdash p'+4 \hookrightarrow e). \\
s,h & \Vdash & \mathsf{Reachable}(m^{\mathsf{fs}}, e^{\mathsf{int}})\ \text{iff} \\
& & \quad s,h \Vdash \mathsf{Pairheap} \text{ and} \\
& & \quad [\![m]\!]s = \{p \in Ptr \mid \exists P \in \mathsf{Path}.\ s,h \Vdash \mathsf{eval}'(P,e,p)\}
\end{array}
$$

We can now conclude

Theorem 8.6. *Let* (s,h) *be a state such that*

$$s,h \Vdash \mathsf{Reachable}(\mathsf{ALIVE}, \mathsf{root}) \ \wedge \ \mathsf{InitAss}.$$

Then, if $GC^*, s, h \rightsquigarrow^* s', h'$, (s',h') *is a garbage collected version of* (s,h).

PROOF: We note that, by the definition of pruning, $s,h \Vdash \mathsf{Reachable}(\mathsf{ALIVE}, \mathsf{root})$ is equivalent to $s(\mathsf{ALIVE}) = dom(prune(h))$. Then use Corollary 8.4. ☐

We have therefore completed our task, and we may conclude

Theorem 8.7. *Our implementation of Cheney's algorithm meets the required specification, which implies that there exists a weak heap isomorphism between the initial state and the terminal state of execution. Therefore, it can safely be used as a garbage collector.*

We note that the proof has been almost entirely formal. All the proofs in Section 7 and most of the proofs in Section 8 used only the proof rules from Section 3.4, and no semantical arguments. Only the last steps needed a semantical argument, since we had to reason about the situation the heap before and after execution.

9 Related and Future Work

In this section, we discuss work that relates to this paper. We begin by describing some of the literature on Separation Logic, and then we discuss some work on more type-theoretic approaches to the problems regarding reasoning about low-level programs. After this, we will give an account of some work in the related field of Proof Carrying Code, and then we describe other proofs of garbage collectors. Next, we briefly describe some earlier approaches to the problem treated in this paper, and finally we give pointers for future work.

9.1 Separation Logic

In this section, we will give a brief account of the work in Separation Logic that is mostly relevant for this paper. Obviously, the introductory papers [Rey00], [IO01], [ORY01] on **BI** as a program logic and the Frame Rule [IO01] (also mentioned in Section 2) are relevant, since they laid the foundation for this work, but we will not go into details about that work here. We focus on work with more technical resemblance to ours.

The $\forall_*$ connective was suggested by Reynolds, and is a generalization of the $\odot$ connective which he introduced in a course on Separation Logic [Rey03]. The $\odot$ connective was used to prove programs involving arrays, so it was sufficient to reason about intervals of pointers. In his slides, Reynolds proves several results, for example correctness of implementations of a cyclic buffer and quicksort using $\odot$. In our setting, intervals are not adequate, since the sets FORW and UNFORW are not intervals.

As mentioned in the Introduction, Yang was the first to use Separation Logic to prove a "non-toy" program in his thesis [Yan01]. The Schorr-Waite algorithm is interesting in its own right, since it can be used as the marking part of a mark-and-sweep garbage collector. The implementation uses extra "bit" fields (mark and check), whereas we do not use such fields. Although most of the proof is formalized, there are some "semantical holes" in the proof (e.g. Lemma 80), where the deduction is not justified by logical rules, but rather by a semantical argument. We have been aiming at using logical rules as much as possible in our proof. We have followed Yang's idea of using properties of special classes of assertions. He used pure and strictly exact assertions; we have also used other classes of assertions (monotone and domain-exact) and have used properties that these assertions enjoy.

There is a difference in methods between our proof and other proofs of programs that use Separation Logic. In [Rey00], a predicate $\mathsf{dlist}\ \alpha(\mathsf{i},\mathsf{i}',\mathsf{j},\mathsf{j}')$, which says that the mathematical sequence α as a list is represented in the heap, is introduced, and a program for deleting zero-valued elements from the list is proved correct. In [ORY01], a predicate $\mathtt{tree}(\tau, p)$ saying that the mathematical tree τ is represented in the current heap is used. Yang has similar predicates $markedlistR(L, E)$ and $spansR(\tau, E)$ in his proof. The basic pattern is the same in all these situations: the predicates all say something like "a mathematical structure (a tree, a list, etc.) is represented in the current heap". Proofs of programs then proceed by local reasoning and one proves that small changes correspond to small changes in the mathematical structure. Finally, one concludes that an isomorphism between the mathematical structure exists after execution of the program, and infers that the program is correct.

An earlier version of this paper used the same approach. It used a predicate which stated that a *graph* was represented in the heap. But since graphs are not defined inductively (like lists and trees), it was not possible to use this predicate to do the same kind of "reasoning on part of the structure" that is done with the inductively defined structures mentioned above. This proof became mostly semantical, since the reasoning took place at the level of graphs (it was a further complication that the original graph is destroyed during execution of the program, so there was no immediate isomorphism which could be used as an invariant). Since all information about the heap was stored in the graph, the use of local reasoning became eluded.

Our current proof does not use the approach with a mathematical structure being represented in the heap. The information about the structure is encoded in finite sets and relations and the iterated separating conjunction instead. We believe that this approach can also be used in proofs of other graph algorithms, but the verification of this claim is deferred to future work.

9.2 Type Theoretic Approaches

We give descriptions of type-theoretic approaches to the problem of proving safety of programs involving heaps.

Typed Assembly Language In the paper [MWCG99] by Morrisett *et al.*, a *Typed Assembly Language* (TAL) is introduced. TAL is based on a conventional assembly language, but with certain type annotations placed in its syntax. This induces a notion of well-formedness, which in turn guarantees desirable properties such as *subject reduction* (the operational semantics preserves well-formedness), *type safety* (well-formed programs do not get stuck), and *progress* (well-formed programs terminate). Further, it is shown how one can compile a variant λ^{F} of the polymorphic λ-calculus into TAL. For each of the steps in the compilation, it is shown that the translation is type-correct, and this means that the target program is well-formed, if the source program was. A problem in TAL is, as mentioned in Section 1, that the language has a `malloc` instruction and assumes an infinite amount of memory, which makes it unrealistic. A proof of a garbage collector might make the setting in [MWCG99] more realistic, if one expands all calls to `malloc` to a call to a runtime system like the one sketched in Section 5 (but this is deferred to future work).

Alias Types When allocating memory in TAL, the allocated cells are "stamped" with the types they can be initialized with. In the setting of *Alias Types* [SWM00], memory is handled more uniformly, and the *constraints* that ensure the safety of programs are simpler than the well-formedness conditions of TAL. The intention of constraints is to describe the "shape" of the heap and, for example, a function can only be called when the heap conforms to the constraints of that function. Again, there is a notion of a well-formedness (of a pair of a program and a store), and a corresponding soundness theorem stating that if such a pair is well-formed, execution of the program does not get stuck. The main problem is the requirement of the static description of memory described above. This means that all aliasing must be described in the types at all times. It might be useful to introduce framing of some sort in this work.

Hierarchical Storage In [AJW03], Ahmed, Jia, and Walker combine ideas from the Ambient Logic [CG00] by Cardelli and Gordon, from **BI**, and from region calculi [TT94] to develop a logic for reasoning about about hierarchical storage (a slightly modified version of the logic, where the structure on locations is less explicit, appears in [AW03]). They give a storage model based on this, along with a programming language to mutate stores. In this language, terms have types that are based on the new logic. The standard results (preservation of types by the operational language, progress of the semantics on well-formed states) are proved, but the usefulness of the language is not demonstrated (only a complex example of a simple operation is shown). It is, however, interesting to see a type-theoretic approach which utilize the ideas from **BI** / Separation Logic to give descriptions of storage. It will be interesting to see the offspring of this work, as it might narrow some theoretical gaps between Separation Logic and type-theory.

Capabilities The last work we will mention in our "survey" of type-theoretic approaches to safe memory management is the paper [CWM99] on capabilities by Crary, Walker, and Morrisett. The authors propose a Calculus of Capabilities, which is also an extension of the region calculus. The main feature is that regions are annotated with a *capability*, and the capabilities indicate whether it is safe to deallocate a region, and it is only safe to evaluate a term, if a given set of capabilities is held. For example, to read or allocate a region r, it is necessary to hold the capability to access r. Naturally, an effort must be made to address to the problem of aliasing, and the capability system tracks aliasing by adding *multiplicities* to capabilities, and a region r is unique (so r is the only region variable denoting its particular region) if it has unique multiplicity; this is denoted r^1. By default, all regions have unique capability, but one can create references to them inside `let`-declarations, thereby increasing the multiplicity. Then it is only safe to deallocate a region in a capability which says that the region is unique. With this system, one can guarantee that there are no memory leaks, and also the "usual" properties (type soundness, subject reduction, progress) hold. As the authors point out, the aliasing system is not very fine-grained: a region is either unique, or it may alias other regions, and this reduces flexibility, since there is no difference between a region that has two references to it and one with 17 (say) references to it. Further, it is not possible to give a "frame-like" rule and do local reasoning, as pointed out in the paper.

9.3 Proof Carrying Code

In their papers [NL96] and [Nec97], Lee and Necula introduced *Proof Carrying Code* (PCC) as a mechanism to determine safe execution (by a *code consumer*) of code provided by a *code producer*. The code producer publishes a *safety policy*, which, loosely speaking, is a protocol which describes allowed operations and the invariants that must hold when a program uses the operations. The producer then generates a program *along with a proof of its correctness*, and the consumer will then validate the proof before executing the program, since this guarantees safety of the program. A pre- and a postcondition is given for a program, and the code producer can insert invariants at strategic points in the program. A verification condition (VC) is then automatically generated for each in the program, and it is the duty of the producer to exhibit a (machine-checkable) proof for each of these steps. If the VC for the postcondition is provable, the program is safe to execute. The derivations in the papers are based on the Edinburgh Logical Framework (LF) and use a type-based approach.

Appel's approach to the idea of PCC [App01] objects to the idea of the VC generator used in [NL96], [Nec97], and also generalizes to other approaches to verification than the type-based one, by allowing proofs to be constructed and verified from the basics of mathematical logic (with no type-axioms); hence the name *Foundational Proof Carrying Code* (FPCC). This makes it more flexible (since it allows for other types of arguments than type-theoretic ones), and since only a minimal number of axioms are used in the formalization, it is also safer, since a smaller *Trusted Computing Base* (TCB) is used. For proving the safety of programs, the paper takes the same approach as in [MWCG99], namely to to use a type-preserving compiler, and thus we also get here that typability ensures safety. The disadvantage is that the proofs are harder to construct and often involve complex semantic models for types.

As a reaction to the complex proofs that are often necessary in FPCC, Hamid, *et al.* have devised a Syntactic Approach to FPCC [HST^{+}02]. A global invariant is used, and the method used to prove safety is to show that the invariant holds initially, is preserved by all "steps" the machine takes, and that the invariant implies that the "safety policy" of the machine is not violated. As in TAL [MWCG99] and Alias Types [SWM00], this depends on a notion of well-formedness. This is guaranteed by compiling a variant of TAL, called FTAL (Featherweight Typed Assembly Language) into the machine language. FTAL is then shown to have properties like TAL (progress, preservation), and it is shown that the operational semantics of FTAL is in harmony with the step semantics of the machine language (via a relation $\Rightarrow$). This yields the simple invariant $Inv(S)$ saying that "there exists a type-correct FTAL-program P such that $P \Rightarrow S$", and this ensures safety. Again, a problem is that FTAL, like TAL, has a `malloc` instruction and supposes an infinite amount of memory.

Many of the ideas from Proof Carrying Code would be interesting to use in the context of Separation Logic. For example, the idea of distributing a proof of safety along with a program could readily be analyzed for Hoare-style logics, if we have an encoding of such proofs. Nipkow's group in Munich have developed a framework for formally proving programs in traditional Hoare Logic with arrays [vON02], and an extension to Separation Logic is on its way. Whether this is a good platform for a safety protocol for proofs in Separation Logics remains to be seen.

9.4 Proofs of Garbage Collectors

Type Preserving Garbage Collectors In their POPL'01 article [WA01], Wang and Appel illustrate a way to transform (type-safe) programs into a form where they explicitly call a function which acts as a garbage collector for the program. Since this is a well-formed function, the garbage collector is type-safe and preserves the type safety of the program being garbage collected. This gives a better performance than pure region approaches, and can reduce the size of the TCB for PCC since one does not have to trust the garbage collector. The garbage collector implemented in this framework is of the same type (stop & copy) as the one treated in this paper, and the treatment of garbage collection starts after a CPS-conversion, a closure-conversion, and a region annotation [TT94] of the original program. The main problem is, of course, that of sharing and forwarding pointers, and this is dealt with by two tricks. First, a runtime check is used to check whether two (or more) region variables are bound to the same region in memory, allowing safe deallocation. Second, it is assumed that objects have an extra field for forwarding pointers, and this causes the need for casting between types. A problem is that no account is taken for cyclic data structures. This is not needed, since the language being collected, does not support the creation of these, but it is a drawback, and it is not clear how their technique could be used to garbage collect more low-level languages, for example like that of [COB02].

In contrast, our proof requires no extra fields for forwarding pointers, we can collect cyclic structures, and the sharing problem is, of course, dealt with by the connectives from (our version of) Separation Logic. However, it is interesting to see a type-theoretic approach to safe garbage collection, since it addresses the assumption of an infinite amount of storage that is implicit in the work mentioned before.

Typed Regions In [MS02], Monnier and Shao combine the methodologies of region calculi and Alias Types, and obtain a language in which they can write a type preserving garbage collector which can deal with cyclic structures. The collector they implement is a *generational* one [JL97, chp. 7], so more advanced than the one we have implemented. They have two generations (implemented by regions) and one region used for ref-cells. The combination of the two methodologies mentioned results in a language, where, for example, the types of terms are terms from the Calculus of Inductive Constructions [PM93]. The type of a pointer holds both the location it points to (like in TAL), and also some information (called the *intended type*) of the object it points to, so much information is stored in the type system. Regions are typed with functions mapping the objects they hold (along with their intended type) to their actual type. Our language is simpler, but of course, we have complex formulas to keep track of in our proof. How much complexity that will be added if we take on the challenge of proving a generational collector is a question which we defer to future work.

Comparison Between Type-theoretic and Logical Approaches: The work in [SWM00] and [MWCG99] encompass a generic way to prove properties of a programs: Exhibit a programming language with a sufficiently expressive type system, and define a suitably strong notion of well-formedness. Then prove that this notion of well-formedness implies that programs do not get stuck, and are therefore safe. The type-theoretic approach thus stems from the slogan

> "Well-typed programs do not go wrong"

When the programming languages involved involve heap manipulations, new ideas have to be introduced in the type systems, making them at the same time more complex and more expressive. This can for example be seen in the papers mentioned above. The idea is then that if a program is well-formed (and thereby safe), one can "strip" the program of types, and the resulting program is also safe.

In contrast, the approach used to deal with the problems that come with pointers and aliasing in Separation Logic, has been to extend the logic with new basic predicates and connectives that capture relevant properties about pointers. Then programs are proved with new proof rules that use these connectives, and the slogan that well-specified programs do not go wrong (see Section 2) is invoked to conclude that programs do not read or modify heap locations that they do not have access to. The Frame Rule is important to mention here, since it helps keep proofs of the critical heap operations lightweight. A rule like this might help make the rules in the type-theoretic approaches more tractable.

In the type-based work described here, the properties one can infer about a program using these approaches are limited to *safety*; the program does not write to or dereference dangling pointers (this is captured by the fact that programs do not get stuck). In comparison, proofs in program logics often also prove *correctness*: not only does a program not get stuck, but it also does what we expect it to do. In the case of the garbage collector, it is good to know that it does not core-dump, but it is also vital for the user program that the structure of its memory is preserved. On the other hand, one can prove other kinds of results using type theory that are hard to attack using a Hoare-logic based approach. For example, type theoretic arguments are used intensively to prove correctness of translations between languages in [MWCG99]. It is not clear how one could prove this kind of result using program logics.

9.5 "Logical" Proofs of Garbage Collectors

As is described above, the challenge of proving correctness / safety of garbage collectors has been an interesting challenge in academia. Apart from the type-theoretic approaches described above, there have also been a number of attempts that were not based on type-theoretic arguments. We will briefly describe these here. Most of the collectors that have been verified in this way have been "mark-and-sweep" collectors [JL97, chp. 4], where the main part of the algorithm "marks" those locations that are reachable, so that a "sweep" of dead cells can safely take place.

To the author's knowledge, the first attempt of a proof of a garbage collector was published in [DLM$^+$78], where the problem of garbage collection "was selected as one of the most challenging – and, hopefully, most instructive! – problems". The language being garbage collected is essentially the same as in our setting, but the proof given is informal, and merely seems to give hints as to how one might give a formal proof. Other informal proofs (of essentially the same algorithm, only the number of colors used in the marking phase vary) have been published by Ben-Ari [BA84] and Pixley [Pix88]. The informal approach was defended by Ben-Ari:

> So as not to obscure the main ideas, the exposition is limited to the critical facets. A mechanically verifiable proof would need all sorts of trivial invariants.

Unfortunately, the proofs in both expositions turn out to be fallacious, as demonstrated in [Rus94]. In this exposition, Russinoff has explored how great a level of details that is needed for a complete formalization of a proof of a garbage collector, and has verified his proof of both safety (no live nodes are deallocated) and liveness (all dead cells are eventually deallocated) in a mechanical program verification system. It must be noted, however, that the proof does not operate on a "real" heap, but rather on a graph. The proof consists of 22 lemmas (which work like invariants) that are proven in the verification system, using more than a hundred sublemmas (that are, however, not documented). As mentioned below, an implementation of our proof into a theorem prover is deferred to future work. But it will be interesting to compare this with that of [Rus94], although the two garbage collectors in question are fundamentally different.

9.6 Future Work

In this section, we give pointers for future research.

In order to ensure that the proof herein is flawless, it is desirable to further formalize the proof, for example using a logical framework like ISABELLE [NPW02]. My experiences with ISABELLE are limited to an introductory course, but it seems that it would be a good place to start formalizing, and as mentioned before, initiatives to integrate Separation Logic in ISABELLE have already been taken. The main challenges in this regard are at least twofold. First, the extensions of Separation Logic with finite sets and relations, must be formalized in ISABELLE, in extension of the implementation underway. Second, we have used some structural rules in our proof that are not directly part of the idea used in the implementation of Separation Logic in ISABELLE. An assertion is just encoded as a semantic function from its free variables to the domain $\{\mathbf{true}, \mathbf{false}\}$, and there is no notion of syntax-directed ideas such as purity and domain-exact assertions in the upcoming implementation.[1] This might turn out to be an even more challenging problem than the first one.

In object-oriented programming languages, the notions of *modules* and their *interfaces* are important. Modules might utilize other modules' interfaces and this creates an important abstraction, since the programmer is free to choose how to *implement* the module. However, if the language in which the module is implemented has explicit access to memory, it is not clear what the criteria are for two implementations to be equivalent (and therefore interchangeable). Some work has already been done in this regard. Already in [Hoa72], Hoare outlined a notion of equivalence of data representations, and Mitchell have given a good tutorial on the subject in [Mit91], but there was no heap involved in their languages. In more recent work, Banerjee and Naumann have devised a (semantic) criterion which guarantees that two implementations of a module are indistinguishable [BN02], and Clarke, Noble, and Potter have given syntactic criteria to ensure the same goal [CNP01]. Finally, Reddy and Yang have devised rules from Separation Logic to give a categorical setting where one can show correctness of data representations that involve heaps [RY03].

In [Pym02], [OPY03], [OP99], [Yan01], [Cal02], categorical models for **BI** are given. In particular, it is shown how one can model the logic with a topos of presheaves, exploiting the DCC-structure that these possess. Since we have extended the logic with finite sets and relations, and in particular, with the new connective $\forall_*$, it needs to be explored what categorical machinery is necessary to model these new features.

As mentioned in Section 9.1, our approach differs from other proofs of programs in Separation Logic inasmuch as we use sets and relations and the $\forall_*$ connective. It will be interesting to explore the applicability of this approach by proving other graph algorithms correct. This could then be used to prove correctness of other garbage collectors.

There is a need for transforming the ideas from Separation Logic from sequential languages into a setting where parallel and communicating systems use shared resources. Some initial steps have been taken at Queen Mary, University of London in this regard. It will be interesting to follow and take part in this development.

10 Conclusion

Yang proved correctness of the Schorr-Waite graph marking algorithm, which involves non-trivial pointer manipulations to maintain an "internal stack" that keeps track of the depth-first marking that this algorithm performs. Apart from that, the proofs of programs in the literature that use Separation Logic have involved not very complex programs, like reversing a list, deleting a tree, etc. By proving correctness of Cheney's garbage collection algorithm, we have exhibited a second witness to the power of Separation Logic. We are sure that more will follow.

We have used an extension of "traditional" separation logic to formally prove correctness of a simple copying garbage collector. We have aimed to make the proof entirely "formal", that is, we wanted to use a set of proof rules rather than arguing at the semantical level. Although the last part of the argument that our invariant implies existence of a weak heap isomorphism is semantical, we believe that we have obtained this goal to a great extent; for example, the proof that the invariant is maintained by the **while**-loop in the program is entirely formal.

The benefits of using separation logic is best illustrated in the proof of CopyCell in Section 7.2.3. The local specification involving the critical heap manipulations was proved using simple rules, and after that, we used the

[1] According to private correspondences with Tobias Nipkow.

Frame Rule to insert the local specification a global one. If we were to do this in traditional Hoare logic using predicates like the version of Reachable from Section 2, it would be a nightmare, and would surely involve semantical arguments.

In the course of proving the invariant, we have added more and more “set-theoretic” conjuncts to it in order to make the argument go through. As a consequence, the invariant has become rather large and complex-looking; this can be seen as a flaw in the proof, but one may also view it as a confirmation of Ben-Ari’s statement which was quoted in Sec. 9.5: in the course of proving correctness of a garbage collector, there *are* a lot of little details that must be in place.

An earlier version of the proof used functions and bijections instead of the current relations. This reduced the size of the invariant, but we had to take into account that the functions might not always be defined, and as a consequence, the term language and its properties became unpleasant. The solution with relations yields a cleaner term language that is also more intuitive to reason about. As mentioned before, we believe that this logic (or a variant thereof) can be used as a basis for proving correctness of other graph algorithms and garbage collectors.

References

[AJW03] A. Ahmed, L. Jia, and D. Walker, *Reasoning about hierarchical storage*, Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003) (Ottawa, Canada), June 2003, To Appear.

[App92] A. Appel, *Compiling with continuations*, ch. 16, Cambridge University Press, 1992.

[App01] A. W. Appel, *Foundational proof carrying code*, LICS'01, 16th Annual IEEE Symposium on Logic in Computer Science, 2001.

[AW03] A. Ahmed and D. Walker, *The logical approach to stack typing*, Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003) (New Orleans, LA, USA), January 2003, pp. 74 – 85.

[BA84] M. Ben-Ari, *Algorithms for on-the-fly garbage collection*, ACM Transactions of Principles on Programming Languages and Systems **6** (1984), no. 3, 333 – 344.

[BN02] A. Banerjee and D. A. Naumann, *Representation independence, confinement and access control [extended abstract]*, POPL 02, 2002.

[Cal02] C. Calcagno, *Semantic and logical properties of stateful programming*, Ph.D. thesis, DISI, Università di Genova, 2002.

[CG00] L. Cardelli and A. Gordon, *Anytime, anywhere: Modal logics for mobile ambients*, Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 2000, pp. 365 – 377.

[Che70] C. J. Cheney, *A nonrecursive list compacting algorithm*, Communications of the ACM **13** (1970), no. 11.

[CNP01] D. G. Clarke, J Noble, and J. M. Potter, *Simple ownership types for object containment*, Proc. European Conference on Object-Oriented Programming, June 2001.

[COB02] C. Calcagno, P. O'Hearn, and R. Bornat, *Program logic and equivalence in the presence of garbage collection*, To appear in Theoretical Computer Science, 2002.

[CWM99] K. Crary, D. Walker, and G. Morrisett, *Typed memory management in a calculus of capabilities*, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (San Antonia, TX, USA), January 1999, pp. 262–275.

[DLM$^+$78] E. W. Dijkstra, L. Lamport, A. J. Martin, G. S. Scholten, and E. M. F. Steffens, *On-the-fly garbage collection: an exercise in cooperation*, Communications of the ACM **21** (1978), no. 11, 966 – 975.

[Hoa69] C. A. R. Hoare, *An axiomatic approach to computer programming*, Communications of the ACM **12** (1969), no. 583, 576 – 580.

[Hoa72] ______, *Proof of correctness of data representations*, Acta Informatica **1** (1972), 271–281.

[HST$^+$02] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, *A syntactic approach to foundational proof carrying code*, Tech. Report YALEU/DCS/TR-1224, Yale University, New Haven, CT, USA, January 2002.

[IO01] S. Ishtiaq and P. W. O'Hearn, *BI as an assertion language for mutable data structures*, Principles of Programming Languages (London), vol. 28, ACM - SIGPLAN, 2001.

[JL97] R. Jones and R. D. Linz, *Garbage collection – algorithms for automatic dynamic memory management*, ch. 1 – 7, John Wiley & Sons, 1997.

[Mit91] J. C. Mitchell, *On the equivalence of data representations*, Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy (V. Lifschitz, ed.), Academic Press, 1991.

[MS02] S. Monnier and Z. Shao, *Typed regions*, Tech. Report YALEU/DCS/TR-1242, Dept. of Computer Science, Yale University, New Haven, CT, October 2002.

[MWCG99] G. Morrisett, D. Walker, K. Crary, and N. Glew, *From system F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), no. 3, 527 – 568.

[Nec97] G. C. Necula, *Proof-carrying code*, Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97) (Paris), January 1997, pp. 106–119.

[NL96] G. C. Necula and P. Lee, *Safe kernel extensions without run-time checking*, 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA (Berkeley, CA, USA) (USENIX, ed.), USENIX, 1996, pp. 229–243.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — a proof assistant for higher-order logic*, LNCS, vol. 2283, Springer, 2002.

[OG76] S. Owicki and D. Gries, *An axiomatic proof technique for parallel programs*, Acta Informatica **6** (1976), no. 4, 319 – 340.

[OP99] P. O'Hearn and D. J. Pym, *The logic of bunched implications*, Bulletin of Symbolic Logic **5** (1999), no. 2.

[OPY03] P. O'Hearn, D. J. Pym, and H. Yang, *Possible worlds and resources: The semantics of BI*, To Appear in Theoretical Computer Science, 2003.

[ORY01] P. W. O'Hearn, J. C. Reynolds, and H. Yang, *Local reasoning about programs that alter data structures*, Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001 (Berlin), Lecture Notes in Computer Science, Springer-Verlag, 2001.

[PhD01] *The phd curriculum at the IT University of Copenhagen*, Internet page, 2001, Available at `http://www.it-c.dk/Internet/research/phd/ac_board/`.

[Pix88] C. Pixley, *An incremental garbage collection algorithm for multimutator systems*, Distributed Computing **3** (1988), no. 1, 41 – 50.

[PM93] C. Paulin-Mohring, *Inductive definitions in the system Coq - rules and properties*, Proceedings of Typed Lambda Calculi and Applications (M. Bezem and J.-F. Grothe, eds.), Lecture Notes in Computer Science, no. 664, Springer Verlag, 1993.

[Pym02] D. Pym, *The semantics and proof theory of the logic of bunched implications*, Applied Logics Series, vol. 26, Kluwer, 2002.

[Rey00] J. C. Reynolds, *Intuitionistic reasoning about shared mutable data structures*, Millenial Perspectives in Computer Science (J. Davies, B. Roscoe, and J. Woodcock, eds.), Palgrave, Houndsmill, Hampshire, 2000, pp. 303 – 321.

[Rey02a] ______, *Future directions*, 2002, Slides from the Course Reasoning about Low-Level Programming Languages held at Carnegie Mellon University, Spring 2002. Available from the course's home page.

[Rey02b] ______, *An introduction to separation logic*, 2002, Slides from the Course Reasoning about Low-Level Programming Languages held at Carnegie Mellon University, Spring 2002. Available from the course's home page.

[Rey02c] ______, *Separation logic: A logic for shared mutable data structures*, Proceedings of Logic in Computer Science (Copenhagen), vol. 17, IEEE, July 2002, pp. 55 – 74.

[Rey03] ______, *Iterated separating conjunction*, 2003, Slides from the Course Reasoning about Low-Level Programming Languages held at Carnegie Mellon University, Spring 2002. Available from the course's home page.

[Rus94] D. M. Russinoff, *A mechanically verified incremental garbage collector*, Formal Aspects of Computing **6** (1994), 359 – 390.

[RY03] U. S. Reddy and H. Yang, *Correctness of data representations involving heap data structures*, Proc. of the 12th European Symposium on Programming, ESOP 2003 (P. Degano, ed.), Springer Verlag, 2003, pp. 223 – 237.

[SWM00] F. Smith, D. Walker, and G. Morrisett, *Alias types*, European Symposium on Programming, March 2000.

[TT94] M. Tofte and J.-P. Talpin, *Implementing the call-by-value lambda-calculus using a stack of regions*, Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, January 1994, pp. 188–201.

[vON02] D. von Oheimb and T. Nipkow, *Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited*, Formal Methods – Getting IT Right (FME'02) (Lars-Henrik Eriksson and Peter Alexander Lindsay, eds.), LNCS, vol. 2391, Springer, 2002, pp. 89–105.

[WA01] D. Wang and A. W. Appel, *Type preserving garbage collectors*, Proceedings of the 28th annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'01) (London), January 2001, pp. 166 – 178.

[Yan01] H. Yang, *Local reasoning for stateful programs*, Ph.D. thesis, University of Illinois, 2001.