Local Reasoning about a Copying Garbage Collector

Lars Birkedal
The IT University of Copenhagen
birkedal@itu.dk

Noah Torp-Smith
The IT University of Copenhagen
noah@itu.dk

John C. Reynolds
Carnegie Mellon University
John.Reynolds@cs.cmu.edu

Abstract

We present a programming language, model, and logic appropriate for implementing and reasoning about a memory management system. We then state what is meant by correctness of a copying garbage collector, and employ a variant of the novel separation logics [18, 23] to formally specify partial correctness of Cheney's copying garbage collector [8]. Finally, we prove that our implementation of Cheney's algorithm meets its specification, using the logic we have given, and auxiliary variables [19].

Categories and Subject Descriptors

D.2.8 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, Logics of programs, Specification techniques*

General Terms

Realiability, Theory, Verification

Keywords

Separation Logic, Copying Garbage Collector, Local Reasoning

1 Introduction

Reasoning formally about low-level imperative programs that manipulate data structures involving pointers has proven to be very difficult, mainly due to a lack of reasoning principles that are adequate and simple at the same time. Recently, Reynolds, O'Hearn and others have suggested *separation logic* as a tool for reasoning about programs involving pointers; see [23] for a survey and historical remarks. In his dissertation, Yang showed that separation logic is a promising direction by giving an elegant proof of the non-trivial Schorr-Waite graph marking algorithm [28]. One of the key features making separation logic a promising tool is that it supports *local reasoning*: when specifying and reasoning about program fragments involving pointers, one may restrict attention to the "footprint" of the programs, that is, to just that part of memory that the program fragments read from or write to.

The aim of this paper is to further explore the idea of local reasoning and its realization in separation logic. To this end we prove the correctness of Cheney's copying garbage collector [8] via local reasoning in (an extension of) separation logic. There are several reasons why we have chosen to focus on Cheney's algorithm:

- The algorithm involves many imperative manipulations of data; in particular, it not only updates an existing data structure as the Schorr-Waite algorithm does, but *moves* data around. Moreover, it simultaneously treats the same data as a set of records linked by pointers and as an array of records.
- Cheney's algorithm copies any kind of data, including *cyclic data structures*; to the best of our knowledge, our correctness proof is the first such done via local reasoning for programs copying cyclic data structures.
- Variants of the algorithm are used in practice, e.g., in runtime systems for compilers for functional programming languages, so it is not a "toy"-example.
- To the best of our knowledge, there is no formal correctness proof of Cheney's algorithm in the literature.

Indeed, for the reasons above, proving correctness of a garbage collector, has been considered a key challenge for a long time. (In 1992 Nettles gave an informal proof of correctness of a copying garbage collector [17] and expressed a wish for a formal verification.)

There are two other motivating factors that we would like to call attention to. The first is that our analysis answers a question in the literature and thus paves the way for important future work that so far has been out of reach: In [7], local reasoning and separation logic for a garbage collected language is analyzed. An underlying garbage collector is presumed in the operational semantics of the language, inasmuch as a *partial pruning* and *α-renaming* (i.e., relocation) of the current state is allowed at any time during execution of a program. In [7] it is not mentioned *how* this pruning and renaming should be done, let alone proven that it is done correctly. A remark at the end of the paper expresses the desirability of such a proof — we provide one here. The analyses in [7] and the present paper are at two distinct levels: the former is at the level of a user language using a runtime system (a garbage collector), the latter is at the level of a runtime system providing operations for the user language (memory allocation and garbage collection). We believe these analyses pave the way for an investigation of the correctness

POPL'04 January 14 – 16, 2004, Venice, Italy

of *combinations* of user level programs and runtime systems, a goal that so far has been unfeasible. We present some preliminary ideas in this direction in Sec. 8.

The additional second motivating factor is that our analysis of garbage collection should be of use in connection with foundational proof-carrying code [2] and typed assembly language [14]. In these settings, a memory allocation (but no deallocation) construct is part of the instruction set and a memory management system is implicitly assumed. We believe that our correctness proof can contribute to mimicking the work of [14] in a more realistic setting, for more machine-like assembly languages.

1.1 Contributions and Methodology

In Sec. 2 we present our storage model and the syntax and semantics of assertions. Our storage model is very concrete and close to real machines; it treats locations as multiples of four. (It is assumed that all data other than locations are encoded as nonmultiples of four.) This is very similar to what is often used in real implementations of runtime systems for compilers. For simplicity we assume that heaps only consist of **cons**-cells, aligned such that the first component is always on a location divisible by eight; hence pointers (to **cons**-cells) are multiples of eight. As usual in separation logic, a *state* consists of a *stack* and a *heap*, where a *stack* is a finite map from variables to values, and a *heap* is a finite map from locations to integers. A new feature is that our values include finite sets and relations of pointers and integers, which are used to give the semantics of assertions and of auxiliary variables [19].

Our assertion language, presented in Sec. 2.3, is an extension of separation logic [23] with new assertion forms for finite sets and relations. These new forms are crucially used to express part of the specification of Cheney's algorithm; in particular the existence of an isomorphism between pointers to old cells and pointers to copies of the old cells. We believe this methodology of using sets and relations can be used more widely, to specify and prove correct other programs involving shared or cyclic structures. Indeed a somewhat similar approach is being used by Richard Bornat [5] to specify and verify an algorithm for copying directed acyclic graphs.

Moreover, we have extended the iterated separating conjunction [23] of separation logic to arbitrary finite sets. The assertion $\forall_* x \in m.\, A$ holds in a state s,h, if m denotes a finite set $\{p_1,\ldots,p_k\}$ and $A[p_1/x] * \cdots * A[p_k/x]$ holds in s,h (see Fig. 1 for a precise definition). As illustrated in Sec. 5, one can specify a program by dividing the locations it manipulates into disjoint sets, using the iterated separating conjunction, together with expressions for finite sets and relations, to express particular properties for each set of locations.

One could argue that it is a weakness of separation logic that we had to extend it with the above mentioned new constructs. Such extensions must be expected, however, when a young logic encounters a subtle algorithm. The real question is whether the resulting specification is a natural formalization of the programmer's understanding of the algorithm.

In Sec. 3 we define the syntax and semantics of the programming language used for implementing the garbage collector. It is a simple imperative programming language, with constructs for heap lookup and heap update (but no constructs for allocating or disposing of heap cells). The associated program logic is presented in Sec. 3.2. The program logic is mostly standard except for the new rules regarding sets and relations. We include the Frame Rule of separation logic, which makes local reasoning possible, as explained in Sec. 3.2.

In Sec. 4 we express what it means for a garbage collector to be correct. Our definitions are based on the analysis in [7] (already referred to above) and thus involve pruning and α-renaming of program states.

Cheney's algorithm and the specification of our implementation thereof are presented in Sec. 5; the implementation itself is included in Appendix A. We present an informal analysis of the algorithm and use it to derive a formal specification of an invariant. The pointers manipulated by the algorithm can be naturally divided into disjoint sets and thus it is natural to use the method of sets and relations together with the iterated separating conjunction mentioned before. The sets and relations are also used in another crucial way, namely to record the initial contents of the heap (before garbage collection). This makes it possible to relate the final heap (after garbage collection) to the initial heap and prove that the final heap is a garbage collected version of the initial heap.

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

In Sec. 6 we prove our implementation of Cheney's garbage collection algorithm correct. The proof proceeds by first showing that the proposed invariant from the previous section is indeed an invariant and then showing that the invariant suffices to conclude correctness. We present the key ideas of the proofs and in particular explain how local reasoning allows us to reason locally about different fragments of the algorithm, thus illustrating the power of the Frame Rule. Full proofs with all details can be found in the companion technical report [4].

In Sec. 7 we discuss some related work, besides what we have already discussed above, and finally we conclude and present some suggestions for future work in Sec. 8.

2 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 [18], but we extend the expression and assertion languages with finite sets and relations, new basic assertions about these, and the extension of the iterated separating conjunction to arbitrary finite sets.

2.1 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 $x, y, \ldots$ range over Var and assume a type-function

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

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. More formally, we define:

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

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

2.2 Expressions

We define the syntax and semantics for expressions of each of the types $\mathsf{int}, \mathsf{fs}, \mathsf{frp}, \mathsf{fri}, \mathsf{path}$. For expressions of type int we just present the syntax; the semantics is standard. For expressions of the remaining types, we just present the semantics as the syntax will be evident from the presentation of the semantics.

Expressions of type int are defined by the following grammar:

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

where $n \in \mathbb{Z}$ and $j \in \mathbb{N}\setminus\{0\}$. The semantics of $\#m^{\mathsf{fs}}$ is the number of elements in the finite set denoted by m^{fs} (see below). In order to avoid introducing an explicit type of boolean values, we use a standard encoding of truth values, where 0 denotes "false", and all other integers denote "true". Although the superscript that indicates the type is only meant to indicate the type of variables, we 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.

We use m to range over expressions of type fs. The semantics of an expression of type fs is a finite set of pointers. The operator $\oplus$ 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.

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

With this, we can formally define the semantics of expressions of the form $\#m^{\mathsf{fs}}$:

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

We use f to range over expressions of type frp. The semantics of an expression of type frp is a finite relation on pointers.

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

We use P to range over expressions of type path. The semantics for expressions of type path is straightforward, in that the denotation of an expression is equal to itself:

$$[\![P]\!]s = P.$$

To conclude our semantics for expressions, we give the semantics for expressions of type fri. We use g to range over such expressions. The $\circledcirc$ operator will be used to model the structure-preserving property of a garbage collector, inasmuch as it extends a relation with the identity on non-pointers before composing it with another relation (cf. Definition 6 later in this paper).

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

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

2.3 Assertions

The assertion language is an extension of separation logic [23] with new assertion forms for finite sets and relations. We just present the semantics of assertions; again the syntax is evident from the presentation of the semantics.

We use A, B, and D to range over assertions. The set $FV(A)$ of free variables for an assertion is defined as usual, but note that x is bound in $\forall_* x \in m.\ A$. Substitution $A[e/x]$ of the expression e for the variable x in the assertion A is defined in the standard way. We sometimes write an assertion A as $A(x)$ to emphasize that the variable x may occur free in A.

The formal semantics of assertions is given by a judgement of the form

$$s,h \Vdash A,$$

the intended meaning of which is that the assertion A holds in the state s,h. We require that $FV(A)\subseteq dom(s)$. The semantics is given in Fig. 1; here b ranges over the boolean expressions $e_1 \le e_2, e_1 = e_2$, and δ ranges over Types. In Fig. 1, 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 define the combined heap $h_1 * h_2$ by

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

A brief explanation of some of the clauses in Fig. 1 is appropriate here. The assertion forms emp, $e_1 \mapsto e_2$, $A * B$, and $A \mathrel{-\!\!*} B$ are

A	$s,h \Vdash A$ if and only if
b	$[\![b]\!]s \neq 0$
T	always
F	never
$\neg A$	$s,h \nVdash A$
$A \to B$	$s,h \Vdash A$ implies $s,h \Vdash B$
$A \wedge B$	$s,h \Vdash A$ and $s,h \Vdash B$
$A \vee B$	$s,h \Vdash A$ or $s,h \Vdash B$
$\forall x^{\delta}.\, A$	for all $v \in [\![\delta]\!]$, $s[x \mapsto v],h \Vdash A$
$\exists x^{\delta}.\, A$	for some $v \in [\![\delta]\!]$, $s[x \mapsto v],h \Vdash A$
$m_1 = m_2$	$[\![m_1]\!]s = [\![m_2]\!]s$
$(e_1,e_2) \in f^{\mathsf{frp}}$	$([\![e_1]\!]s,[\![e_2]\!]s) \in [\![f]\!]s$
$(e_1,e_2) \in g^{\mathsf{fri}}$	$([\![e_1]\!]s,[\![e_2]\!]s) \in [\![g]\!]s$

A	$s,h \Vdash A$ if and only if
$e_1 \mapsto e_2$	$h = \{([\![e_1]\!]s,[\![e_2]\!]s)\}$
emp	$dom(h) = \varnothing$
$A * B$	there are heaps h_1,h_2 such that $h_1 \# h_2$, $h_1 * h_2 = h, s,h_1 \Vdash A$, and $s,h_2 \Vdash B$
$A \twoheadrightarrow B$	$s,h*h' \Vdash B$ for all h' such that $h \# h'$ and $s,h' \Vdash A$
$\mathsf{Ptr}(e)$	$[\![e]\!]s \in \mathsf{Ptr}$
$\mathsf{PtrRg}(g,m)$	$\forall\,(p,q) \in [\![g]\!]s.\ q \in \mathsf{Ptr} \Rightarrow q \in [\![m]\!]s$
$m \perp m'$	$[\![m]\!]s \cap [\![m']\!]s = \varnothing$
$m_1 \subseteq m_2$	$[\![m_1]\!]s \subseteq [\![m_2]\!]s$
$\mathsf{Tfun}(f^{\mathsf{frp}},m)$	$\forall p \in [\![m]\!]s.\ \exists! n \in \mathbb{Z}.\ (p,n) \in [\![f]\!]s$
$\mathsf{Tfun}(g^{\mathsf{fri}},m)$	$\forall p \in [\![m]\!]s.\ \exists! n \in \mathbb{Z}.\ (p,n) \in [\![g]\!]s$

A	$s,h \Vdash A$ if and only if
$\mathsf{iso}(f,m_1,m_2)$	$\forall p_1 \in M_1.\ \exists! p_2 \in M_2.\ (p_1,p_2) \in \phi\ \wedge\ \forall p_2 \in M_2.\ \exists! p_1 \in M_1.\ (p_1,p_2) \in \phi\ \wedge$ $\forall (p_1,p_2) \in \phi.\ p_1 \in M_1 \wedge p_2 \in M_2$, where $M_1 = [\![m_1]\!]s, M_2 = [\![m_2]\!]s, \phi = [\![f]\!]s$
$\mathsf{eval}(g,g',P^{\mathsf{path}},e,e')$	$(P = \varepsilon$ and $s,h \Vdash e = e')$, or $(P = P' \cdot \mathsf{head}$ and $\exists p \in \mathsf{Ptr}.\ s,h \Vdash \mathsf{eval}(g,g',P',e,p)$ and $s,h \Vdash (p,e') \in g)$, or $(P = P' \cdot \mathsf{tail}$ and $\exists p \in \mathsf{Ptr}.\ s,h \Vdash \mathsf{eval}(g,g',P',e,p)$ and $s,h \Vdash (p,e') \in g')$
$\mathsf{Reachable}(g,g',m,e)$	$[\![m]\!]s = \{p \in \mathsf{Ptr} \mid \exists P \in \mathsf{Path}.\ s,h \Vdash \mathsf{eval}(g,g',P,e,p)\}$
$\forall_* p \in m.\ A$	$\begin{cases} s,h \Vdash A[p_1/p] * \cdots * A[p_k/p], & \text{if } [\![m]\!]s = \{p_1,\ldots,p_k\} \\ s,h \Vdash \mathsf{emp} & \text{if } [\![m]\!]s = \varnothing \end{cases}$

Figure 1. Semantics of Assertions

taken from 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 $A \twoheadrightarrow B$ means that for all heaps h' disjoint from the current heap h, if A holds in h', the combination $h * h'$ will satisfy B. The assertions eval and Reachable concern evaluation of paths, and $\mathsf{PtrRg}(g,m)$ says that any pointer in the range of the relation denoted g is in the set denoted by m. Finally, $\forall_*$ is an *iterated separating conjunction*. Informally, if $s,h \Vdash \forall_* x \in m.\ A$, and if $[\![m]\!]s = \{p_1,\ldots,p_k\}$, then h can be split into disjoint heaps $h = h_1 * \cdots * h_k$ with $s,h_1 \Vdash A[p_1/x],\ldots,s,h_k \Vdash A[p_k/x]$. Note that the semantics is classical for the standard first-order logic fragment.

We use the following standard shorthand notations

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

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

For later use, we introduce some special classes of assertions. The definitions are taken from [28] and [23].

- 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',

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

  where $\subseteq$ indicates set-theoretic inclusion of graphs.

In the literature, monotone assertions are often called *intuitionistic* [23].

Remark 1.

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

Definition 1. We call an assertion A *valid* if, for all states s,h with $FV(A) \subseteq dom(s)$, we have $s,h \Vdash A$. We use $\Rightarrow$ to denote semantic validity, i.e., $A \Rightarrow B$ if $s,h \Vdash A$ implies $s,h \Vdash B$, for all states s,h.

One may easily verify that a number of useful assertion schemas are valid; below we present some of the more interesting ones (and omit obvious assertions about finite sets and relations, the standard rules for classical logic, and simple arithmetic).

First we present some rules for the iterated separating conjunction.

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

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

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

$$\begin{aligned} &(\forall_* x \in m.\ A) \wedge e \in m \to \\ &(\forall_* x \in (m \ominus e).\ A) * A[e/x] \end{aligned} \tag{4}$$

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

Next, we give rules involving our special operator $\circledcirc$ on relations.

$$(e_1,e_2) \in g \wedge \neg \mathsf{Ptr}(e_2) \rightarrow \forall x^{\mathsf{frp}}.\ (e_1,e_2) \in x \circledcirc g \tag{6}$$
$$(e_1,e_2) \in g \wedge (e_2,e_3) \in f \rightarrow (e_1,e_3) \in f \circledcirc g \tag{7}$$
$$(e_1,e_2) \in f \circledcirc g \rightarrow ((e_1,e_2) \in g \wedge \neg \mathsf{Ptr}(e_2)) \vee (\exists x.\ \mathsf{Ptr}(x) \wedge (e_1,x) \in g \wedge (x,e_2) \in f) \tag{8}$$
$$\mathsf{Tfun}(g,m) \wedge (e_2,e_1) \in g \wedge e_2 \in m \wedge \mathsf{Ptr}(e_1) \wedge (e_2,e_3) \in f \circledcirc g \rightarrow (e_1,e_3) \in f \tag{9}$$
$$((e_1,e_2) \in f \circledcirc g \wedge e_1 \in m' \wedge ((\exists x.\ (e_1,x) \in g \wedge x \in m) \vee \neg \mathsf{Ptr}(e_2)) \wedge \mathsf{Tfun}(f,m) \wedge \mathsf{Tfun}(g,m')) \rightarrow e_2 = e_3 \tag{10}$$

The following rules exploit that heaps are single-valued.

$$e \hookrightarrow e_1 \ \wedge\ e \hookrightarrow e_2 \rightarrow e_1 = e_2 \tag{11}$$

When $x \notin FV(e_1,e_2)$,

$$(e_1 \hookrightarrow e_2) \ \wedge\ ((\exists x.\ e_1 \mapsto x \ \wedge A(x)) * B) \rightarrow (e_1 \mapsto e_2 \ \wedge A[e_2/x]) * B \tag{12}$$

If B is pure and B' is monotone, then

$$A \wedge B \Rightarrow B' \text{ implies } (A * A') \wedge B \Rightarrow B' \tag{13}$$

The following lemmas can be applied when we reason about assertions involving $\forall_*$.

LEMMA 1. *Suppose $s,h \Vdash \forall_* x \in m.\ A$ and that $\forall x.\ x \in m \wedge A \rightarrow B$ is valid. Then $s,h \Vdash \forall_* x \in m.\ B$.*

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

LEMMA 2. *If D is a pure assertion, and if $D \wedge A \rightarrow A'$ and $D \wedge B \rightarrow B'$ are valid, then $D \wedge (A * B) \rightarrow D \wedge (A' * B')$ is valid.*

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

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

As an example of a rule that can be derived from the rules above, we get the following from (12) and monotonicity. When A is a monotone assertion and $x \notin FV(e_1,e_2)$,

$$\begin{array}{l} (e_1 \hookrightarrow e_2) \ \wedge\ ((\exists x.\ (e_1 \mapsto x \ \wedge A)) * B) \\ \Downarrow \\ (e_1 \mapsto e_2 \ \wedge A[e_2/x]) * B \\ \Downarrow \\ ((e_1 \mapsto e_2) * B) \ \wedge A[e_2/x] \end{array} \tag{14}$$

3 Programming 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 defined above to give a program logic for the language.

3.1 Syntax and Semantics

Definition 2. The syntax of the programming 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 or deallocating locations on the heap. It would be straightforward to add such constructs to the language, but we will not need them. In our specification and implementation of Cheney's algorithm we simply assume that the domain of the heap contains the necessary locations.

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 3. The relation $\rightsquigarrow$ on configurations is defined by a number of inference rules, most of which are completely standard. Thus we just present the rules for assignment, heap lookup, and heap update:

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

The semantics is easily seen to be deterministic.

Definition 4. 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 $C,s,h \rightsquigarrow^* s',h'$.

Other published definitions of the programming language used by separation logic use a special configuration called "abort" or "fault" instead of the concept of "stuck"; we are able to avoid this complication because we have restricted the programming language to a deterministic sublanguage.

As is standard, we define $\mathrm{Mod}(C)$ 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.

3.2 Program Logic

Definition 5. Let A and B be assertions, and let C be a command. The *partial correctness specification* $\{A\}\ C\ \{B\}$ 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 refer to A as the *precondition* of the specification and to B as the *postcondition*.

We present a set of proof rules that are sound with respect to Def. 5. Since the rules regarding constructs from the simple **while**-language are standard [11], we only present rules regarding the heap.

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

$$\begin{array}{l}\{(\exists y.\ e \mapsto y \wedge A) \wedge x = e'\}\\ \quad x := [e]\\ \{e[e'/x] \mapsto x \wedge A[x/y]\}\end{array} \tag{15}$$

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

$$\begin{array}{l}\{\exists y.\ e \mapsto y\ \wedge\ A\}\\ \quad x := [e]\\ \{e \mapsto x \wedge A[x/y]\}\end{array} \tag{16}$$

Rule for heap update.

$$\{e_1 \mapsto -\}\ [e_1] := e_2\ \{e_1 \mapsto e_2\} \tag{17}$$

The Frame Rule

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

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

Derived Rule for Pure Assertions

$$\frac{\{A\}\ C\ \{B\}}{\{A\ \wedge\ A'\}\ C\ \{B \wedge\ A'\}}\ A' \text{ pure, } \mathrm{Mod}(C) \cap FV(A') = \varnothing$$

As an example of another useful derived rule, we note how pure assertions can move in and out of the Frame Rule: When A' is pure, and $\mathrm{Mod}(C) \cap FV(B') = \varnothing$,

$$\frac{\{A' \wedge A\}\ C\ \{B\}}{\{A' \wedge (A * B')\}\ C\ \{B * B'\}} \tag{18}$$

This follows from Remark 1 and the standard rule of consequence.

We then have the expected soundness result.

THEOREM 1. *If a specification $\{A\}\ C\ \{B\}$ is derivable by the rules above and the standard rules of Hoare logic, then $\{A\}\ C\ \{B\}$ holds.*

Example: The following rule for lookup can be derived from (16), the Frame Rule, and the standard rules of Hoare logic.

If $y \notin FV(A)$, and x and y are distinct variables,

$$\begin{array}{l}\{A \wedge (x \hookrightarrow -)\}\\ \quad y := [x]\\ \{A \wedge (x \hookrightarrow y)\}\end{array} \tag{19}$$

4 Expressing Garbage Collection

In this section we define what it means for a state to be a garbage collected version of another state. Our formulation builds upon the analysis in [7] and thus involves pruning and α-renaming of states.

A basic requirement is that the heaps of the two states are isomorphic. For simplicity, we assume that heaps only consist of **cons**-cells.

Definition 6. Let h and h' be heaps. Call h and h' *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. We refer to β as a *heap isomorphism*.

Of course, it is only necessary to have a heap isomorphism between the subheaps consisting of the *live* data, i.e., the data reachable from a given *root set*. For simplicity, we will assume that there is only one root cell.

Definition 7. Let s,h be a state with $\mathsf{root} \in dom(s)$. Pointer q is *reachable from pointer* p in the heap h if $p = q$ or if $h(p) = p_1$, $h(p+4) = p_2$, and q is reachable from p_1 or p_2 in h. The pointer p is called *reachable* in the state s,h if p is reachable from $s(\mathsf{root})$ in h. Finally, $prune(s,h) = s,g$, where $g \subseteq h$ is the subheap of h restricted to those pointers reachable in s,h.

Definition 8. Let s,h and s',h' be states. If β is a heap isomorphism between h and h', and $\beta(s(\mathsf{root})) = s'(\mathsf{root})$, then we call β a *state isomorphism*.

The formal notion of garbage collection is then as follows:

Definition 9. Let s,h and s',h' be states. We say that s',h' is a *garbage collected version* of s,h if there exists a state isomorphism $\beta : prune(s,h) \cong prune(s',h')$.

Definition 10. A command GC is a *correct garbage collector* if $GC,s,h \rightsquigarrow^* s',h'$ implies that s',h' is a garbage collected version of s,h.

5 Cheney's Algorithm

We implement and reason about *Cheney's Algorithm* [8]. The implementation of the algorithm and the associated memory allocator is given in Appendix A. It assumes two contigous "semi-heaps", $\mathsf{OLD} \equiv Itv(\mathsf{startOld},\mathsf{endOld})$ and $\mathsf{NEW} \equiv Itv(\mathsf{startNew},\mathsf{endNew})$ of equal size. The memory allocator attempts to allocate a **cons**-cell in OLD; if there is no space available in OLD, the garbage collector copies all cells in OLD reachable from root into NEW, and then the allocation resumes in NEW. The garbage collector is delimited by comments in the code in Appendix A; we refer to it as GC^*. Notice that the algorithm is aware of the locations of the live cells only (those locations reachable from root). In the spirit of local reasoning, our specification will therefore only involve the reachable pointers in OLD, called ALIVE, and not the remaining (unreachable) part of OLD.

Notice that the set $\{\phi, \mathsf{FORW}, \mathsf{UNFORW}\}$ is an *auxiliary variable set* for the implementation, in the sense of [19]. Thus, the assignments to these variables are not necessary for the program to work, but they ease the job of proving properties about the program. We could have chosen to existentially quantify these variables and omit them from the program, but the reasoning becomes clearer when the program modifies the auxiliary variables explicitly.

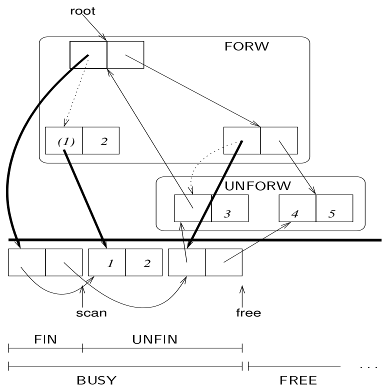

Figure 2. A state of execution

5.1 The Precondition

Before execution of GC^*, we assume that the following assertion holds.

$$\begin{aligned}&\mathsf{InitAss} \equiv\\ &\mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{maxFree}) \wedge \#\mathsf{ALIVE} \leq \#\mathsf{NEW} \wedge\\ &(\mathsf{ALIVE} \perp \mathsf{NEW}) \wedge \mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge\\ &\mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge\\ &\mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{root} \in \mathsf{ALIVE} \wedge\\ &((\forall_* y \in \mathsf{ALIVE}.\ ((\exists z.\ (y,z) \in \mathsf{head} \wedge y \mapsto z) *\\ &\quad (\exists z'.\ (y,z') \in \mathsf{tail} \wedge y+4 \mapsto z')))*\\ &(\forall_* y \in \mathsf{NEW}. y \mapsto -,-))\end{aligned}$$

Informally, the assertion, among other things, expresses that ALIVE is the set of pointers reachable from root, and that head and tail are relations recording the initial contents of the reachable cells, as can be seen from the iterated separating conjunction over the set ALIVE.

5.2 The Invariant

To exhibit an invariant of the **while**-loop, we consider Fig. 2, which is a snapshot of a state during execution. Only the reachable cells in OLD (the part of the heap above the bold horizontal line) are shown. Three of the cells in ALIVE have been modified at this stage: their first component have been updated with *forwarding pointers*; these have a bolder appearance in the figure. The original contents of these first components are indicated with dotted lines and parenthesized numbers. The pointers in ALIVE naturally divide into two sets:

- UNFORW: The pointers in ALIVE that point to cells not yet modified by the algorithm.
- FORW: The pointers in ALIVE that point to cells that have their first component overwritten with a pointer in NEW.

The algorithm proceeds by *scanning* all the cells in between the scan and free pointers, that is, scan always points to the next cell to be scanned. If the cell being scanned contains a non-pointer, then the scan pointer is incremented; if the cell being scanned contains a pointer p in UNFORW, then the cell pointed to by p is copied and a forwarding pointer is placed in the original cell; if the cell being scanned contains a pointer in FORW, then the cell pointed to has already been copied and we simply update the scanned cell. We use the auxiliary variables ϕ, FORW, and UNFORW to keep track of the forwarding pointers, and to record the live cells that have been already copied into NEW. When a cell is copied from OLD to NEW, the corresponding pointer is moved from UNFORW to FORW, and ϕ is updated.

The pointers in NEW can be divided into the following three sets:

- $\mathsf{FIN} \equiv Itv(\mathsf{offset}, \mathsf{scan})$: The pointers in NEW that have been scanned. These are not modified further by the algorithm.
- $\mathsf{UNFIN} \equiv Itv(\mathsf{scan}, \mathsf{free})$: The pointers in NEW that have not been scanned. These point to the original contents of cells pointed to by pointers in ALIVE.
- $\mathsf{FREE} \equiv Itv(\mathsf{free}, \mathsf{maxFree})$: The pointers in NEW that are available for allocation.

The five sets are illustrated in Fig. 2. Note that FIN, UNFIN and FREE are intervals, whereas this is not the case for FORW and UNFORW in general. We observe that there is a one-to-one correspondence, ϕ, between the pointers in FORW and those in $\mathsf{BUSY} \equiv \mathsf{FIN} \cup \mathsf{UNFIN} = Itv(\mathsf{offset}, \mathsf{free})$. This bijection will turn out to be the heap isomorphism we are looking for.

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

$$\begin{aligned}&I_{pure} \equiv\\ &\mathsf{iso}(\phi, \mathsf{FORW}, \mathsf{BUSY}) \wedge (\mathsf{ALIVE} = \mathsf{FORW} \cup \mathsf{UNFORW}) \wedge\\ &\mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root}) \wedge (\mathsf{ALIVE} \perp \mathsf{NEW}) \wedge\\ &\mathsf{PtrRg}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{PtrRg}(\mathsf{tail}, \mathsf{ALIVE}) \wedge\\ &\mathsf{Tfun}(\mathsf{head}, \mathsf{ALIVE}) \wedge \mathsf{Tfun}(\mathsf{tail}, \mathsf{ALIVE}) \wedge\\ &(\#\mathsf{ALIVE} \leq \#\mathsf{NEW}) \wedge (\mathsf{root} \in \mathsf{FORW}) \wedge (\mathsf{scan} \leq \mathsf{free}) \wedge\\ &\mathsf{Ptr}(\mathsf{free}) \wedge \mathsf{Ptr}(\mathsf{scan}) \wedge \mathsf{Ptr}(\mathsf{offset}) \wedge \mathsf{Ptr}(\mathsf{maxFree})\end{aligned}$$

Note in particular the conjunct $\mathsf{iso}(\phi, \mathsf{FORW}, \mathsf{BUSY})$ expressing that ϕ is a bijection, and the conjunct

$$\mathsf{Reachable}(\mathsf{head}, \mathsf{tail}, \mathsf{ALIVE}, \mathsf{root})$$

expressing that ALIVE is the set of pointers reachable via head and tail from root, i.e., the set of cells that were live initially. The rest of I_{pure} simply records basic facts about the relationship between the various sets and pointers.

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

The cells pointed to by pointers in UNFORW have not been modified by the algorithm; hence they are described by head and tail, and we thus define

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

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

$$\mathsf{A}_{\mathsf{FORW}} \equiv \forall_* y \in \mathsf{FORW}.\ (\exists z.\ (y,z) \in \phi \wedge y \mapsto z, -).$$

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

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

The cells in FIN have been scanned. The case-distinction between pointers and non-pointers during scanning is captured by the oper-

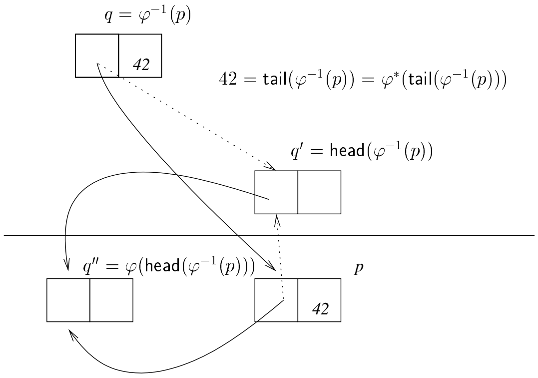

Figure 3. The situation for a pointer p in FIN

ator $\circledcirc$. We define

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

To understand $\mathsf{A}_{\mathsf{UNFIN}}$ and $\mathsf{A}_{\mathsf{FIN}}$, it is helpful to consider Fig. 3 in which we use a functional notation for functional relations. The pointer $p \in \mathsf{FIN}$ is the address of the rightmost bottommost cell. Before p was scanned, it held the original contents of a cell pointed to by a pointer $q \in \mathsf{ALIVE}$. When that cell was copied, it had its first component overwritten with the forwarding pointer p; this is recorded by ϕ, hence $(p,q) \in \phi$. The original contents of the cell pointed to by q is recorded by head and tail, so letting q' denote the address of the rightmost uppermost cell in Fig. 3, we have $(q,q') \in \mathsf{head}$, hence $(p,q') \in \mathsf{head} \circ \phi^{\dagger}$. Before the cell pointed to by p was scanned, it had q' in its first component. Now, by scanning the first component in the cell pointed to by p, we copy the cell pointed to by q' (if necessary), and update the component we are scanning to point to the address of the copy of that cell. Denoting the address of the copy by q'', we then have $(q',q'') \in \phi$, by the definition of ϕ, and therefore, $(p,q'') \in \phi \circ (\mathsf{head} \circ \phi^{\dagger})$.

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

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

Summing up, the invariant of the algorithm is

$$I \equiv I_{pure} \wedge (\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}} * \mathsf{A}_{\mathsf{FREE}}).$$

6 Proofs

In this section we prove that our implementation of Cheney's garbage collection algorithm is correct. We present the key ideas of the proof and refer to the companion technical report [4] for further details.

The proof proceeds in two stages. First we show that I from Sec. 5.2 is indeed an invariant, and then we use the invariant to show that the algorithm is correct in the sense of Definition 10.

We will sometimes need to consider the iterated separating conjunction over one of the sets from Section 5.2, except for one element. We therefore, for example, write $\mathsf{A}_{\mathsf{FORW-x}}$ for the assertion

$$\forall_* y \in (\mathsf{FORW} \ominus \mathsf{x}).\ (\exists z.\ (y,z) \in \phi \wedge y \mapsto z, -).$$

6.1 *I* is an Invariant

We prove two specifications:

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

where INIT is the code before the **while** loop, and BODY is the body of the loop. The proof of the first specification is similar to part of the proof of the second specification, and we therefore omit it. Observe that BODY consists of two similar parts, $\mathsf{ScanCar}$ and $\mathsf{ScanCdr}$ (delimited by comments in the code). We show the specification for $\mathsf{ScanCar}$, the one for $\mathsf{ScanCdr}$ is similar. After $\mathsf{ScanCar}$, the cell pointed to by scan is in a "mixed state", where the first component is finished and the second is about to be scanned. Therefore, the specification for $\mathsf{ScanCar}$ is $\{I\ \wedge \mathsf{scan} \neq \mathsf{free}\}\ \mathsf{ScanCar}\ \{I'\}$, where

$$
\begin{aligned}
&I' \equiv\\
&I_{pure} \wedge ((\mathsf{A}_{\mathsf{UNFORW}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{FREE}} * \mathsf{A}_{\mathsf{UNFIN-scan}}) *\\
&\quad (\exists z.\ (\mathsf{scan}, z) \in \phi \circledcirc (\mathsf{head} \circ \phi^{\dagger}) \wedge \mathsf{scan} \mapsto z) *\\
&\quad (\exists z'.\ (\mathsf{scan}, z') \in \mathsf{tail} \circ \phi^{\dagger} \wedge \mathsf{scan}+4 \mapsto z')).
\end{aligned}
$$

Observe that $\mathsf{ScanCar}$ has three branches. We only present the proof of the specification corresponding to the most interesting branch, $\mathsf{CopyCell}^*$ (delimited by comments in the code), in which a cell is copied from OLD to NEW. Thus, we prove the specification

$$
\begin{aligned}
&\{I \wedge \mathsf{scan} \neq \mathsf{free} \wedge \mathsf{scan} \hookrightarrow \mathsf{x} \wedge \mathsf{Ptr}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \wedge \neg(\mathsf{y} \in \mathsf{NEW})\}\\
&\quad \mathsf{CopyCell}^*\\
&\{I'\}
\end{aligned}
\tag{20}
$$

To this end, we first show that the precondition implies the assertion $\mathsf{x} \in \mathsf{UNFORW}$. Next we prove a local specification for the heap manipulations in $\mathsf{CopyCell}^*$, and finally we apply the Frame Rule to infer the required specification (20). This illustrates the power of the Frame Rule.

LEMMA 3. *The assertion*

$$
\begin{aligned}
&I \wedge \mathsf{scan} \neq \mathsf{free} \wedge \mathsf{scan} \hookrightarrow \mathsf{x} \wedge \mathsf{Ptr}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \wedge \neg(\mathsf{y} \in \mathsf{NEW})\\
&\to \mathsf{free} \leq \mathsf{maxFree} \wedge \mathsf{x} \in \mathsf{UNFORW}
\end{aligned}
$$

is valid.

Using this lemma we have the following derivation, the last step of which uses (4):

$$
\begin{aligned}
&\{I \wedge \mathsf{scan} \neq \mathsf{free} \wedge \mathsf{scan} \hookrightarrow \mathsf{x} \wedge \mathsf{Ptr}(\mathsf{x}) \wedge \mathsf{x} \hookrightarrow \mathsf{y} \wedge \neg(\mathsf{y} \in \mathsf{NEW})\}\\
&\Downarrow\\
&\{I \wedge \mathsf{x} \in \mathsf{UNFORW} \wedge \mathsf{scan} \in \mathsf{UNFIN} \wedge \mathsf{free} \in \mathsf{FREE}\}\\
&\Downarrow\\
&\left\{\begin{array}{l}
I_{pure} \wedge\\
((\mathsf{A}_{\mathsf{UNFORW-x}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN-scan}} * \mathsf{A}_{\mathsf{FREE-free}}) *\\
(\exists z'.\ (\mathsf{scan}, z') \in \mathsf{tail} \circ \phi^{\dagger} \wedge \mathsf{scan}+4 \mapsto z') *\\
(\exists z.\ (\mathsf{x}, z) \in \mathsf{head} \wedge \mathsf{x} \mapsto z) *\\
(\exists z'.\ (\mathsf{x}, z') \in \mathsf{tail} \wedge \mathsf{x}+4 \mapsto z') *\\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-))
\end{array}\right\}
\end{aligned}
$$

It is straightforward to show the following local specification using the rules for heap lookup (16), heap update (17), the rule for pure

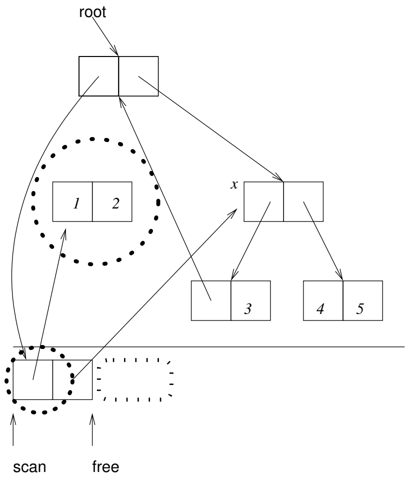

Figure 4. Footprint of CopyCell

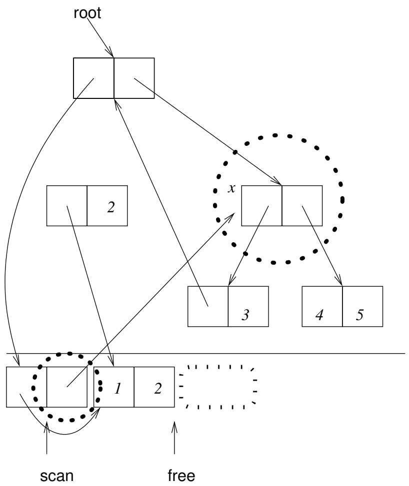

Figure 5. Footprint of ScanCdr

assertions from Remark 1, and the Frame Rule.

$$
\begin{array}{l}
\left\{\begin{array}{l}
(\exists z.\ (\mathsf{x},z) \in \mathsf{head} \wedge \mathsf{x} \mapsto z) * \\
(\exists z'.\ (\mathsf{x},z') \in \mathsf{tail} \wedge \mathsf{x}+4 \mapsto z') * \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-)
\end{array}\right\} \\
\quad \mathsf{t}_1 := [\mathsf{x}]; \mathsf{t}_2 := [\mathsf{x}+4]; \\
\quad [\mathsf{free}] := \mathsf{t}_1; [\mathsf{free}+4] := \mathsf{t}_2; \\
\quad [\mathsf{x}] := \mathsf{free}; [\mathsf{scan}] := \mathsf{free} \\
\left\{\begin{array}{l}
((\mathsf{x} \mapsto \mathsf{free},-) * (\mathsf{scan} \mapsto \mathsf{free}) * \\
(\mathsf{free} \mapsto \mathsf{t}_1,\mathsf{t}_2)) \wedge \\
(\mathsf{x},\mathsf{t}_1) \in \mathsf{head}\ \wedge\ (\mathsf{x},\mathsf{t}_2) \in \mathsf{tail}
\end{array}\right\}
\end{array}
\tag{21}
$$

We write CopyCell for the code fragment in (21). Notice how the specification (21) only mentions the footprint of CopyCell, i.e., those locations that are either dereferenced or updated by CopyCell. This is illustrated in Fig. 4, in which the footprint is shown using dotted circles. After executing CopyCell on the heap depicted in Fig. 4, the resulting heap is the one depicted in Fig. 5, in which the footprint of the following ScanCdr is shown. After the execution of ScanCdr, the resulting heap will be the one depicted in Fig. 2. From (21) we can continue the derivation from before.

$$
\begin{array}{l}
\left\{\begin{array}{l}
I_{pure} \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}-\mathsf{x}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\
(\exists z'.\ (\mathsf{scan},z') \in \mathsf{tail} \circ \phi^{\dagger} \wedge \mathsf{scan}+4 \mapsto z') * \\
(\exists z.\ (\mathsf{x},z) \in \mathsf{head} \wedge \mathsf{x} \mapsto z) * \\
(\exists z'.\ (\mathsf{x},z') \in \mathsf{tail} \wedge \mathsf{x}+4 \mapsto z') * \\
(\mathsf{scan} \mapsto -) * (\mathsf{free} \mapsto -,-))
\end{array}\right\} \\
\quad \mathsf{CopyCell} \hfill (22) \\
\left\{\begin{array}{l}
I_{pure} \wedge \\
((\mathsf{A}_{\mathsf{UNFORW}-\mathsf{x}} * \mathsf{A}_{\mathsf{FORW}} * \mathsf{A}_{\mathsf{FIN}} * \mathsf{A}_{\mathsf{UNFIN}-\mathsf{scan}} * \mathsf{A}_{\mathsf{FREE}-\mathsf{free}}) * \\
(\exists z'.\ (\mathsf{scan},z') \in \mathsf{tail} \circ \phi^{\dagger} \wedge \mathsf{scan}+4 \mapsto z') * \\
((\mathsf{x} \mapsto \mathsf{free},-) * (\mathsf{scan} \mapsto \mathsf{free}) * (\mathsf{free} \mapsto \mathsf{t}_1,\mathsf{t}_2)) \wedge \\
(\mathsf{x},\mathsf{t}_1) \in \mathsf{head}\ \wedge\ (\mathsf{x},\mathsf{t}_2) \in \mathsf{tail})
\end{array}\right\} \\
\quad \mathsf{FORW} := \mathsf{FORW} \oplus \mathsf{x};\ \mathsf{UNFORW} := \mathsf{UNFORW} \ominus \mathsf{x}; \\
\quad \phi := \phi \oplus (\mathsf{x},\mathsf{free});\ \mathsf{free} := \mathsf{free}+8 \\
\{I''\} \\
\Downarrow \\
\{I'\}
\end{array}
$$

where I'' is an assertion similar to I', which we will not state here for reasons of space. The first specification in (22) follows from (21) and the derived Frame Rule (18). The proof of the second specification in (22) uses the standard rules for assignment and the implication $I'' \Rightarrow I'$ uses Lemma 2. To apply Lemma 2, one must show a few implications (see the preconditions of Lemma 2); we include one of them here:

$$
\begin{array}{l}
(\mathsf{x},\mathsf{free}-8) \in \phi \wedge \mathsf{x} \in \mathsf{FORW} \wedge \\
((\forall_* y \in (\mathsf{FORW} \ominus \mathsf{x}). \\
\quad (\exists z.\ (y,z) \in \phi \ominus (\mathsf{x},\mathsf{free}-8) \wedge y \mapsto z,-)) * \\
(\mathsf{x} \mapsto \mathsf{free}-8,-)) \\
\Downarrow \\
((\forall_* y \in (\mathsf{FORW} \ominus \mathsf{x}).\ (\exists z.\ (y,z) \in \phi \wedge y \mapsto z,-)) * \\
(\mathsf{x} \mapsto \mathsf{free}-8,- \wedge (\mathsf{x},\mathsf{free}-8) \in \phi)) \wedge \mathsf{x} \in \mathsf{FORW} \\
\Downarrow \\
((\forall_* y \in (\mathsf{FORW} \ominus \mathsf{x}).\ (\exists z.\ (y,z) \in \phi \wedge y \mapsto z,-)) * \\
(\exists z.\ \mathsf{x} \mapsto z,- \wedge (\mathsf{x},z) \in \phi)) \wedge \mathsf{x} \in \mathsf{FORW} \\
\Downarrow \\
\forall_* y \in \mathsf{FORW}.\ (\exists z.\ (y,z) \in \phi \wedge y \mapsto z,-) \\
= \\
\mathsf{A}_{\mathsf{FORW}}
\end{array}
\tag{23}
$$

The first implication in (23) uses the rule for pure assertions in Remark 1. The third uses the rule (5). The remaining implications needed to conclude I' via Lemma 2 can be found in the technical report [4].

This completes our outline of the formal proof of the specification

$$\{I \wedge \neg(\mathsf{scan} = \mathsf{free})\}\ \mathsf{BODY}\ \{I\}.$$

6.2 Sufficiency of Invariant

We show that the invariant suffices to prove our implementation correct. The following lemma expresses that the algorithm moves all reachable cells from UNFORW to FORW.

LEMMA 4.

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

This lemma is proved using an obvious induction principle for paths.

The next lemma expresses that upon completion of the algorithm, FORW is equal to ALIVE.

LEMMA 5. $I \wedge \mathsf{scan} = \mathsf{free} \rightarrow \mathsf{FORW} = \mathsf{ALIVE}$.

This is easily shown using Lemma 4.

The formal proof of the following theorem is a bit more involved and can be found in [4]. Informally, it expresses that, when we exit the **while** loop, all pointers in ALIVE have been scanned and thus satisfy the condition that p in Fig. 3 satisfies.

THEOREM 2.

$$I \wedge \mathsf{scan} = \mathsf{free} \rightarrow (e \in \mathsf{ALIVE} \wedge (e, e_1) \in \phi \rightarrow (e_1 \hookrightarrow e_2 \leftrightarrow (e, e_2) \in \phi \circledcirc \mathsf{head})).$$

The proof of this uses Lemma 5. Finally, we can prove the main theorem of the paper.

THEOREM 3. *Let (s,h) be a state such that*

$$s, h \Vdash \mathsf{InitAss}.$$

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

PROOF (sketch): We argue that for all $p \in s(\mathsf{ALIVE})$,

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

where $\beta = s'(\phi)$ is the relation denoted by ϕ after execution. If $r = h'(\beta(p))$, then r satisfies the condition of the r in Theorem 2. Therefore, $s', h' \Vdash (p, r) \in \phi \circledcirc \mathsf{head}$, and since head recorded the original contents of p and $\circledcirc$ models the $(-)^*$ construction, this precisely means that $r = \beta^*(h(p))$. ☐

Since our specification does not mention the unreachable part of OLD, we also conlude that these cells are not copied by the program; thus our implementation only copies cells reachable from root.

7 Related Work

There has been several proposals for using types to manage the problem of reasoning about programs that manipulate imperative data structures [9, 25, 1, 20]. They are based on the idea that well-typed programs do not go wrong, but they are not aimed at giving proofs of *correctness*. In the work [9] on capabilities, traditional region calculus [26] is extended with an annotation of a capability to each region, and this gives criteria to decide when it is safe to deallocate a region. In the setting of alias types [25], a static notion of constraint is used describe the shape of the heap, and this is used to decide when it is safe to execute a program. In the work [1] on hierarchical storage, ideas from BI [22] and region calculi are used to give a type system with structure on locations. In [20] Petersen et. al. propose to use a type theory based on ordered linear logic as a foundation for defining how data is laid out in memory. The type theory in [20] builds upon a concrete allocation model such as the one provided by Cheney's copying garbage collector.

The first attempt of a formal correctness proof of a garbage collector was published in [10], where the problem "was selected as one of the most challenging – and hopefully, most instructive! – problems". The proof given there is informal and merely gives an idea of how to obtain a formal proof. Other informal proofs were published in [3] and [21]. The fact that a "mechanically verifiable proof would need all kinds of trivial invariants" was used to justify the informal approach. Russinoff [24] explored how great a detail that was needed for a formal proof and demonstrated that the proofs in [3] and [21] are fallacious. Moreover, Russinoff gave a correctness proof of an abstract version of a mark-and-sweep collector, which, however, did not use local reasoning.

In their work on a type preserving garbage collector, Appel and Wang [27] transform well-typed programs into a form where they call a function, which acts as a garbage collector for the program. This function is designed such that it is well-typed in the target language, and thus is safe to execute. The approach of Appel and Wang guarantees safety, but not correctness of the garbage collector, and there is no treatment of cyclic data structures, since the user language does not create cyclic data structures. Monnier and Shao [13] combine ideas from region calculi and alias types in their work on typed regions and propose a programming language with a type system expressive enough to type a garbage collector, which is type preserving, generational, and handles cyclic data structures.

Recently, there has been a lot of work on Proof Carrying Code [16, 15]. The basic idea of a code producer submitting a proof of safety along with a program could, of course, be transferred to low-level programming languages, like the one used with separation logic. Nipkow's research group in Munich has developed a framework for formally verifying programs in traditional Hoare logic with arrays [12], and an extension to separation logic is underway. This would allow one to verify our correctness proof mechanically and perhaps to ship that proof along with proofs of programs using the garbage collector.

8 Conclusion and Future Work

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

We have extended separation logic with sets and relations, generalized the iterated separating conjunction and shown how these features can be used to specify naturally and prove correct an algorithm involving movement of cyclic data structures. We believe the methods used herein are of wider use and future work should include further experimentation with other subtle algorithms, such as those analyzed in [6] (and also, Bornat's methods might be applicable to Cheney's algorithm).

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

Future work also includes studying how to specify and prove correct combinations of user level programs and runtime systems, as mentioned in the introduction. In his work on Foundational Proof Carrying Code [2], Appel suggests compiling high-level languages into the Typed Assembly Language [14]. Our work offers an alternative to this. We suggest compiling types from high-level languages into *garbage insensitive predicates*, in the sense of [7], and using our memory allocator and garbage collector as an implementation of the `malloc` operation of TAL. By the nature of garbage insensitive predicates, we would have $\{P\}\ GC\ \{P\}$ for these predicates, for any correct garbage collector GC, and thus predicates resulting from type-safety guarantees would be preserved by the garbage collector, as desired.

Acknowledgments

The authors wish to thank Peter O'Hearn, Richard Bornat, Cristiano Calcagno, Henning Niss, and Martin Elsman for insightful discussions. Lars Birkedal's and Noah Torp-Smith's research was partially supported by Danish Natural Science Research Council Grant 51–00–0315 and Danish Technical Research Council Grant 56–00–0309. John Reynolds's research was partially supported by an EPSRC Visiting Fellowship at Queen Mary, University of London, by National Science Foundation Grant CCR-0204242, and by the Basic Research in Computer Science Centre of the Danish National Research Foundation.

A Implementation of Cheney's Algorithm

```
alloc(l,n1,n2) {
  if (free < maxFree)
    [free] := n1;
    [free+4] := n2;
    free := free+8;
    l := free−8
  else
    if (offset = startOld) then
      offset := startNew;
      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
        // CopyCell* begins
          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
        // CopyCell* ends
        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;
    alloc(l,n1,n2)
  fi
}
```

B References

[1] A. Ahmed, L. Jia, and D. Walker. Reasoning about hierarchical storage. In *Proc. LICS'03*, 2003.

[2] A. W. Appel. Foundational proof carrying code. In *Proc. LICS'01*, 2001.

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

[4] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Correctness of a garbage collector via local reasoning. Technical report TR-2003-30, The IT University of Copen-

hagen, Copenhagen, Denmark, July 2003. Available at http://www.itu.dk/English/research/publications/.

[5] R. Bornat. Correctness of copydag via local reasoning. Private Communication, Mar 2003.

[6] R. Bornat. Local Reasoning, Separation and Aliasing. Submitted to the SPACE'04 workshop. Available at http://www.cs.mdx.ac.uk/staffpages/r_bornat.

[7] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557 – 581, 2003.

[8] C. J. Cheney. A nonrecursive list compacting algorithm. *Comm. ACM*, 13(11), November 1970.

[9] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. of POPL'99*, pages 262–275, 1999.

[10] 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. *Comm. ACM*, 21(11):966 – 975, 1978.

[11] C. A. R. Hoare. An axiomatic approach to computer programming. *Comm. ACM*, 12(583):576 – 580, 1969.

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

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

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

[15] G. C. Necula. Proof-carrying code. In *Proc. POPL '97*, pages 106–119, 1997.

[16] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI'96*, pages 229–243, Berkeley, CA, USA, 1996.

[17] S. Nettles. A Larch specification of copying garbage collection. Technical Report CMU-CS-92-219, Carnegie Mellon University, Pittsburgh, PA 15213, Dec 1992.

[18] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL 2001*, Berlin, 2001.

[19] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319 – 340, 1976.

[20] L. Petersen, R. Harper, K. Crary, and F. Pfenning. A type theory for memory allocation and data layout. In *POPL'03*, January 2003.

[21] C. Pixley. An incremental garbage collection algorithm for multimutator systems. *Distributed Computing*, 3(1):41 – 50, 1988.

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

[23] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS'02*, pages 55 – 74, 2002.

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

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

[26] M. Tofte and J.-P. Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proc. POPL'94*, pages 188–201, 1994.

[27] D. Wang and A. W. Appel. Type preserving garbage collectors. In *Proc. POPL'01*, pages 166 – 178, 2001.

[28] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.