Toward a Formal Characterization of Garbage Collection

by

Danny House

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Toward a Formal Characterization of Garbage Collection

Danny House

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

Discussions of garbage collection rest on the intuition that

> *The result of the computation will be unchanged if we pause the machine, alter parts of memory that are not reachable via any chain of pointers, and allow the machine to resume.*

This statement uses a number of ill-defined terms, including *machine*, *computation*, *pause*, and *pointer*. In this thesis we give formal definitions for all of these concepts, and use them to give a formal interpretation of this fundamental intuition.

This thesis is dedicated to my wife, Lee-Anne, who has put up with the laziest of garbage collectors.

I would like to thank Marsha Chechik, who taught me that formal methods could be used for much more that just proofs of correctness, gave me support and encouragement, and showed concern for my future long before becoming my supervisor. I owe much to Ric Hehner, Albert Lai, Benet Devereux, and Arie Gurfinkel, who, by listening to my half-formed thoughts, helped give them shape. Finally, I thank the University of Toronto Formal Methods Reading Group for giving this part-time student a respite from the full-time tyranny of the urgent that is the working world.

Contents

Chapter 1

Introduction

1.1 What is Garbage Collection?

Although computer memory is no longer the scarce resource it once was, it is still easy for an otherwise working program to fail by consuming too much. The phrase *memory leak* is well known to programmers using languages that demand explicit allocation and deallocation of memory (*e.g.*,C). A memory leak arises whenever a program obtains more memory for each item of work, and does not return the memory after the task has been completed. A long running program that enters production with a memory leak will eventually fail. This happened to the dispatch system of the London Ambulance Service on November 4th, 1992 [1]. If there is a known request that causes a server to leak memory, the server is vulnerable to a denial-of-service attack. Memory leaks are difficult to find while testing, and failures due to memory leaks often resist reproduction, which complicates debugging.

Building working software without memory leaks in languages like C requires meticulous effort. Each allocation must be balanced with a deallocation. If deallocation occurs too soon, the program will fail. If the allocated buffer is associated with a work item that may follow one of many paths, then each path must end in a deallocation. This kind of software is hard to maintain.

Intuitively, *garbage* is allocated memory that will never again be used during the computation. *Garbage collection* (or *GC*) is a service provided by a variety of programming languages (*e.g.*, Java, Lisp, and ML). We refer to these languages as *garbage collecting languages* or *GCL*s. A garbage collector is any service that locates and recycles some garbage. In exchange for significant CPU and memory overhead, GC eliminates premature deallocation, reduces the number of ways memory leaks can occur, makes the code clearer and more concise, and saves a great deal of human effort. A language designer can use GC to hide some of the more troublesome aspects of pointers (*e.g.*, pointer arithmetic) behind an abstract view of memory.

1.2 Which Garbage Gets Collected?

The idea that memory can be recycled rests on the following informal intuition.

> *The result of the computation will be unchanged if we pause the machine, alter parts of memory that will never be used again, and allow the machine to resume.*

Trying to find all garbage leads to the halting problem via the construction in Rice's Theorem [11]. Suppose that, for any program, we can run it for a little while, pause it, point to a specific piece of allocated memory and answer the question "will this memory ever be used again?" Then from any program we can construct a new one that simply allocates a piece of memory, runs the initial program, and if it halts, accesses the memory. This answers the question "does the original program halt?" Thus, if we could recognize memory that will never again be used, we could solve Turing's halting problem.

Because it is impossible to identify all garbage, we must be satisfied with being able to find some, and hope that is enough. As the computer executes a program, it performs a sequence of actions. Each action is determined by the combination of a specific instruction and the current state of the computer. Some instructions directly name certain parts of the computer (*e.g.*, registers), and some use implicit or indirect names (*e.g.*, "the next word" or "the instruction that may be found by skipping ahead as many words as the next word indicates"). There is a sense in which each action only depends on and affects the things that are directly or indirectly named by the corresponding instruction. This leads to the intuition that any part of memory that cannot be named by any sequence of instructions must be garbage, and the corresponding assertion:

> *The result of the computation will be unchanged if we pause the machine, alter parts of memory that cannot be named, and allow the machine to resume.*

Due to the great flexibility of programming languages, it is often possible to compute a "name" (*e.g.*, by means of pointer arithmetic in C), so we run into the halting problem again. As before, if we can pause any program and ask "will this name be used again?" then we can construct a solution to the halting problem.

We can avoid this incarnation of the halting problem by explicitly limiting the methods of obtaining and manipulating names; many high-level languages restrict the user this way. Alternatively, we can allow the use of computed names with the understanding that bad things can happen when a name that does not refer to anything is used; this is a reasonable way to structure the internals of a garbage collecting language. An effective compromise is to identify a safe subset of the language - nothing can go wrong so long as the programmer sticks to this subset, and riskier techniques may only be used after taking appropriate precautions. The compromise could be implemented as a semi-opaque library or as a language-extension facility. In this thesis we follow the first approach and formally define the names and operations available to the user.

Garbage collection algorithms are generally described in terms of memory, pointers, fields, cells and roots. These are manipulated by the user via the `new` and `set` operations. `new` takes some description of the memory required and returns a pointer to a memory region that matches the description. We call the region indicated by `new` a *cell*, and view the pointer to it as its *name*. `set` takes the name of a cell and a pointer into a possibly different cell. The pointer indicates a field in the second cell at which `set` stores the name of the first cell. We view this pointer as the *field name*. There is a special set of cells known as *roots*. Once this simple informal interface has been accepted many subtle algorithms can be described and analyzed.

Some garbage can be identified by means of connected subgraphs. If the cells are viewed as the nodes of a graph, and the pointers in their fields are taken to define directed edges, then we can identify the smallest connected subgraph that contains the roots. The fundamental assumption of GC is that everything outside this subgraph is garbage. We avoid the halting problem by limiting ourselves to collecting only this kind of garbage.

1.3 Our Contribution

The use of many GC algorithms rests on the following informal intuition:

Intuition 1 *The result of the computation will be unchanged if we pause the machine, alter parts of memory that are not reachable via any chain of pointers, and allow the machine to resume.*

The contribution of this thesis is the formalization of the many fundamental concepts and assumptions that lie behind Intuition 1.

Here are just a few of the questions that we answer while formalizing our intuition:

1. What is a machine?

2. What is a computation?

3. What does it mean to pause and change a machine?

4. What is memory?

5. What is a pointer?

6. Which parts of memory are reachable via chains of pointers?

7. What is a machine with memory and pointers?

8. What does it mean to alter unreachable memory while the machine is paused?

9. Which computations count?

1.4 Our Garbage Collection Interface

The informal `new-set` interface is not sufficient for a rigorous approach to GC. Naming cells by means of pointers complicates the matter of distinguishing the roots. Using pointers as field names is an effective abstraction for describing GC algorithms, but does not lend itself to hiding the internal data structures from the user of a GC module, and thus obstructs our expressing the module's requirements. The `new-set` interface makes implicit assumptions about how to read pointer fields. The user wants to store and retrieve data, and many interesting GC algorithms are concerned with its representation, but the informal interface ignores data completely. The rest of this section gives an overview of our interface objects and functions for motivational purposes.

There are two kinds of names: *root names* and *pointer field names*. Treatments of GC often contain statements like "the roots are the machine registers and the global variables" [4, 8, 13, 18], or simply assert that some cells are roots [7, 8]; some do both. Formal treatments often assume that there is only one root [5] or identify some fixed set of roots [7]. Treatments that are part of a recipe for building an interpreter are likely to include portions of the stack [4, 18] or environment [15]. We distinguish root names from *root cells* and say that a root name is the name of a root cell. In order to support stacks, environments and algorithms that relocate cells in memory, our interface includes the functions `addRN` (for **addR**oot**N**ame) and `removeRN` (for **removeR**oot**N**ame). `addRN`(r)

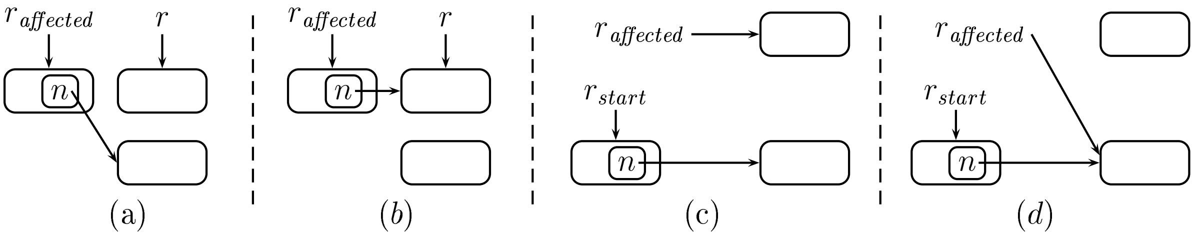

Figure 1.1: `setPF`$(r, n, r_{affected})$ transforms (a) into (b). `followPFN`$(r_{affected}, n, r_{start})$ transforms (c) into (d).

and `removeRN`(r) add or remove the root name r from an active set that corresponds to the stack or environment. (We defer discussion of the active set until Section 4.3.)

We have a `new` function that allocates cells. Instead of returning the name of the allocated cell, `new`(r) takes the root name r and associates the cell with it.

We assume there is some fixed set of pointer field names. In the context of most GC algorithm descriptions, reading a pointer field is trivial[1], and there are implicit assumptions about how to do it using some underlying language. We use `setPF` (for **setP**ointer**F**ield) and `followPFN` (for **followP**ointer**F**ield**N**ame). `setPF`$(r, n, r_{affected})$ causes the cell named by the root name r to be indicated by the pointer field named n of the cell named by the root name $r_{affected}$ (see Figure 1.1, a and b). `followPFN`$(r_{affected}, n, r_{start})$ follows the root name r_{start} to some cell, and looks in that cell's pointer field named n, whereby it finds a cell to associate with the root name $r_{affected}$ (see Figure 1.1, c and d).

The representation of data in GC systems is usually discussed in one of two ways: pretend it is irrelevant or give highly detailed and optimized structures to hold it. The data stored and retrieved by the user plays a critical role in our approach to garbage collection. We take the view that how the data and pointers are represented, and where they are stored, should be hidden from the user. Thus, the data returned by a GC service is its only observable feature. This is why our interface provides the `setData` and `getData` funtions.

[1]Sometimes a read-barrier is used to allow concurrent GC - in this case it is not at all trivial.

Many informal algorithms deal with data-representation explicitly in order to optimize performance. Some sort data-types by size and keep corresponding pools of cells [20]. Some use the high-bits to distinguish small integers from pointers, so that roots can hold them directly [9]. Copy-collection algorithms [3] relocate the non-garbage data and retarget any pointers to it. We abstract all of these details by assuming there is some fixed set of data objects. If r is a root name, then `setData`(d, r) stores data object d in the cell indicated by r, and `getData`(r) returns the data object stored in that cell.

Example 1.4.1 (A Linked-List) *We use our GC interface to construct a linked list with data values 7, 6 and 5. We use two root names, r_1 and r_2, and a pointer field named* `next`.

`addRN`(r_1)	*Prepare two root-names to work with.*
`addRN`(r_2)	
`new`(r_1)	*Get a cell to hold the value 5.*
`setData`$(5, r_1)$	
`new`(r_2)	*Get a cell to hold the value 6.*
`setData`$(6, r_2)$	
`setPF`$(r_1,$ `next`$, r_2)$	*Make the cell with 6 point to the cell with 5.*
`new`(r_1)	*Get a cell to hold the value 7.*
`setData`$(7, r_1)$	
`setPF`$(r_2,$ `next`$, r_1)$	*Make the cell with 7 point to the cell with 6.*

At this point the first and second cells in the list are roots with names r_1 and r_2. See Figure 1.2. Now we fetch the third data value (5) from the linked list.

`followPFN`$(r_2,$ `next`$, r_1)$	*Follow r_1's* `next` *field so that r_2 indicates the second cell.*
`followPFN`$(r_2,$ `next`$, r_2)$	*Follow r_2's* `next` *field so that r_2 indicates the third cell.*
`getData`(r_2)	*Get the data from the third cell.*

At the end, r_1 indicates the cell with 7 and r_2 indicates the cell with 5, so the first and third cells are roots.

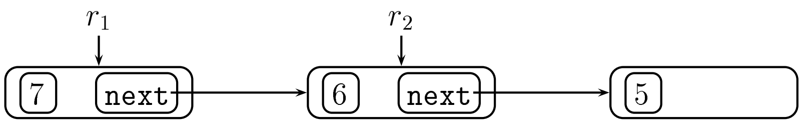

Figure 1.2: This is the result of constructing the linked list in Example 1.4.1.

1.5 Related Work

Although the volume of literature concerning Garbage Collection is huge (Jones provides an online GC bibliography [14] with more than 2500 entries), it is nearly all focused on the presentation and analysis of specific algorithms. The subset that uses formal methods is, in turn, almost entirely directed toward proving that specific algorithms meet two basic correctness criteria: all garbage eventually gets collected, and no non-garbage ever gets collected [5, 6, 12, 19]. Within this subset, Dijkstra *et al.*'s on-the-fly algorithm [7] is a popular subject. Attempts to model GC independently of implementation are rare indeed.

Goguen, Brooksby and Burstall [8] gives a formal treatment of incremental tracing for GC (first described by Dijkstra *et al.* [7]). Their formalization of memory in terms of a graph with actions has much in common with the model presented in this thesis. They define when a node is *action-accessible*, which, in our terms, is when there is a computation that depends on the node. They show that a node is reachable if and only if it is action-accessible.

Our work differs from [8] in emphasis and conclusion. In [8], the emphasis is on formalizing incremental tracing, and their conclusion is that it can be transparently added to a memory manager. They explicitly avoid any discussion of garbage collection. Our emphasis is finding a formal statement of the ways the memory state can be changed without affect on the computation; we are directly concerned with what makes garbage collection possible. Our conclusion is that garbage can be collected. As with most formalizations of existing techniques, our conclusions share one glaring common feature:

we both conclude that what is already known to work, does, in fact, work. This should not be taken as a criticism; it is simply the bane of anyone trying to construct logical foundations for successful methods.

Morrisett, Felleisen and Harper [17] models memory and pointers in a novel fashion. For them, memory is a set of variable/expression bindings in the manner of a LISP environment. The variables constitute the cell addresses, and what we call pointer field names correspond to the free variables in the expressions. Where we follow a pointer, they look up the binding for the variable name in the environment. They never say anything at all about roots! (Presumably, the free variables of the expression being evaluated are the roots.)

1.6 Organization of this Thesis

In Chapter 2, we describe background information the reader will require to follow the formalization.

In chapters 3, 4, and 5, we devote a section to answering each of the questions of Section 1.3 in the order presented there. In Section 3.1 and Section 3.2, we formalize our approach to machines and computations, using actions and viewpoints. We examine what it means to pause and change a machine in Section 3.3. In Section 4.1, we use our formalism to present a random access memory machine. In Section 4.2 through Section 4.4, we construct a memory machine that abstracts the memory models presented by many garbage collecting languages. Section 5.1 is devoted to stating what can and cannot be changed while the machine is paused. In Section 5.2, we find that Intuition 1 does not apply to all computations, and determine exactly which ones it does cover. Finally, a full formal statement of Intuition 1 appears in Section 5.3. In Chapter 6, we describe the role HOL played in writing this thesis.

We draw conclusions, point out limitations of our work, and suggest ideas for further research in Chapter 7. In Appendix A, we go over the lessons we learned about working with HOL, and provide warnings and advice to prospective users of HOL. Appendix B provides a glossary of the symbols used in this thesis.

Chapter 2

Background

$\mathbb{B} = \{\top, \bot\}$ is the set of Boolean values, with $\top$ being true, and $\bot$ being false. $\mathbb{N} = \{0, 1, ...\}$ is the set of natural numbers.

We assume an understanding of sets. We use $\emptyset$ to denote the empty set. Whenever we refer to some set and do not explicitly state otherwise, it may or may not be infinite. Throughout this chapter, S and T denote arbitrary sets. $\wp(S)$ denotes the set of all subsets of S. $|S|$ denotes the cardinality of S; it need not be a finite cardinality.

We find `insert` and `delete` to be useful set operations.

Definition 2.1 (`insert` and `delete`) *For any object x,*

$$\begin{aligned} x \texttt{ insert } S &= \{x\} \cup S, \textit{ and} \\ S \texttt{ delete } x &= S - \{x\} \\ &= \{y \mid y \in S \wedge y \neq x\}. \end{aligned}$$

We assume an understanding of functions. $S \to T$ is the set of total functions from S to T (there are no partial functions in this thesis). The expression $f : S \to T$ states that f is a total function from S to T, or equivalently, that f is in the set $S \to T$. We say that S is the domain of f, and T is its co-domain.

Definition 2.2 ($\Im$ and $\Re$) *If $f : S \to T$ and $S_0 \subseteq S$, then the image of S_0 under f is*

$$\Im(f, S_0) = \{y \mid \exists x : S_0 \cdot y = f(x)\},$$

and the range of f *is* $\Re(f) = \Im(f, S)$.

We find the `let` ... `in` and `if` ... `then` ... `else` constructs useful.

Definition 2.3 **(`let` ... `in`)** *When* x *is a variable and* t_1 *and* t_2 *are terms, we use*

$$\texttt{let } x = t_1 \texttt{ in } t_2$$

as syntactic sugar for $(\lambda x \cdot t_2)t_1$.

Definition 2.4 **(`if` ... `then` ... `else`)** *When* P *is a predicate and* u, t_1 *and* t_2 *are terms,*

$$u = \texttt{if } P \texttt{ then } t_1 \texttt{ else } t_2$$

is an abbreviation for

$$\{P \wedge (u = t_1)\} \vee \{\neg P \wedge (u = t_2)\}.$$

Notice that we have used '{' and '}' for describing sets and as alternatives for '(' and ')'. We feel that the benefit of making heavily parenthesised expressions easier to read outweighs any confusion that this ambiguity may cause.

We often wish to construct new functions from old by changing their value at just one argument. To do this, we use the `update` abbreviation.

Definition 2.5 **(`update`)** *If* f *is a function from* S *to* T, $x \in S$, *and* $y \in T$, *then*

$$\texttt{update}(f, x, y)$$

is an abbreviation for

$$\lambda z : S \cdot \texttt{if } (z = x) \texttt{ then } y \texttt{ else } f(z).$$

We assume an understanding of relations. A relation R on S and T is a subset of $S \times T$. If $s \in S$ and $t \in T$, we write either $R(s, t)$ or sRt to indicate that the pair, $\langle s, t\rangle$ is in R, and say that s is related by R to t.

When $S = T$, we say that R is a relation on S, and we can view S and R as defining a directed graph in which S is the set of nodes, and the elements of R are the edges. We define the transitive closure of R to be the smallest relation, $\widehat{R}$, such that $R \subseteq \widehat{R}$ and

$$\forall x, y, z : S \cdot x\widehat{R}y \wedge y\widehat{R}z \Rightarrow x\widehat{R}z$$

We say that all elements of S are reachable from themselves, and t is reachable from s in R if $s\widehat{R}t$. The next definition allows us to say that an element is reachable under R from a subset of S.

Definition 2.6 (`reachable`) *If S is a set, then*

$$\texttt{reachable} : (S \times S \to \mathbb{B}) \times \wp(S) \times S \to \mathbb{B}$$

is defined by

$$\texttt{reachable}(R, S', t) = t \in S' \vee \exists s : S' \cdot s\widehat{R}t.$$

We provide the next definition to support the use of natural induction in proofs about reachability.

Definition 2.7 (`reachableInN`) *If S is a set, then we define*

$$\texttt{reachableInN} : (S \times S \to \mathbb{B}) \times \wp(S) \times S \times \mathbb{N} \to \mathbb{B}$$

by

$$\begin{aligned}
&\texttt{reachableInN}(R, S', x, 0) = x \in S', \mathit{and} \\
&\texttt{reachableInN}(R, S', x, n+1) \\
&\qquad = \texttt{reachableInN}(R, S', x, n) \vee \exists p : S \cdot \texttt{reachableInN}(R, S', p, n) \wedge R(p, x).
\end{aligned}$$

Here are two simple theorems about reachability.

Theorem 2.1 (inductive form for `reachable`) *If S is a set, R is a relation on S, $S' \subseteq S$, and $x \in S$, then*

$$\texttt{reachable}(R, S', x) = \exists n : \mathbb{N} \cdot \texttt{reachableInN}(R, S', x, n).$$

Theorem 2.2 (relationship between `reachable` and $\subseteq$) *If S is a set, R is a relation on S, $x \in S$ and $S', S'' \subseteq S$, then*

$$S' \subseteq S'' \land \texttt{reachable}(R, S', x) \Rightarrow \texttt{reachable}(R, S'', x).$$

We assume an understanding of lists. We use [ ] to denote the empty list. If $h \in S$ and t is a list of elements of S, then $\texttt{cons}(h, t)$ is the list formed by prefixing t with h. It is convenient to abbreviate $\texttt{cons}(h, t)$ to $h :: t$. S^* is the set of all lists of elements from S.

If l_1 and l_2 are lists, then `append` (l_1, l_2) is the list obtained by concatenating l_1 and l_2 so that l_1 is a prefix of the result.

We assume an understanding of records (tuples with named fields). Suppose R is the set of all records with fields named $N_1, ..., N_m$, and field i has type T_i. Then R is isomorphic to $T_1 \times ... \times T_m$. If $r \in R$, then we use the expression $r.N_i$ to denote the value stored in the field named N_i of the record r. The field names may be viewed as labels for the projection functions. Let π_i be the usual projection of $T_1 \times ... \times T_m$ onto T_i. Then $r.N_i = \pi_i(r)$.

New records may be obtained from old using the `with` ... $\leftharpoondown$ construct. Suppose $\{N_{k_1}, ..., N_{k_n}\} \subseteq \{N_1, ..., N_m\}$ with the N_{k_i} pairwise distinct, and e_i denotes an object of type T_{k_i} for $i = 1, ..., n$. Then

$$r' = r \texttt{ with } N_{k_1} \leftharpoondown e_1, ..., N_{k_n} \leftharpoondown e_n$$

states that the record r' is the same as r except at the fields $\{N_{k_1}, ..., N_{k_n}\}$, and it satisfies $r'.N_{k_i} = e_i$ for $i = 1, ..., n$.

Chapter 3

Machines

3.1 What is a Machine?

There is not yet a definitive answer to this question, either in general or for the subclass of computing machines. Instead, we have many models of machines, none of which is suitable for all purposes. Some pairs of models are interchangeable in the sense that they can be used to simulate each other. Some pairs are strictly incomparable, in that each model permits some computation that the other does not.

Some computing machines are used to control parts of the world by responding to a stream of inputs with a stream of outputs, while others are used to evaluate mathematical expressions. Some, such as large database systems, build complex internal states that must persist across power failures, so there is a sense in which these machines are started and never stopped. Still others return to their initial state every fraction of a second. There are machines that always respond to the same input in the same way, and those that exhibit some degree of non-determinacy.

The intuition being followed here is that a machine is some kind of object that sits in a partially-hidden initial state until the user performs some action upon it. The action causes the machine to perform some hidden actions, passing through some sequence of hidden states, and finally enter a new (or possibly the original) partially-hidden state according to some hidden deterministic rule. We assume that it is easy to tell when the machine has finished reacting to the user's action, by which point it has stopped changing state. It then stays in that state until another action is performed. The machine is used by performing a sequence of actions upon it, viewing some aspects of the state that follows the last action, and interpreting that observation as the answer to some question.

Example 3.1.1 (Calculator) *Consider a simple calculator. The only part of its state that is not hidden is the number display. The actions consist of pushing various buttons. The hidden actions and states are quite complex, but they pass almost instantaneously. At the end, the user observes the numbers displayed and interprets them as the value of the expression that the sequence of button-pushes represented.*

Example 3.1.2 (Telephone) *Consider the use of a simple telephone. The machine is the entire phone system. The actions consist of lifting the handset and pushing various buttons. The only observable part of the system state is the position of the handset and the audio signal coming from the speaker. At the end of a sequence of button pushes, the user listens to the speaker and interprets the sounds as the answer to a wide range of questions, such as "Is my friend's phone busy?".*

To formalize this, we take a *machine* to consist of some *states*, *actions*, *observables*, and *viewpoints*. Prior to being used, the machine is in a special state called the *initial state*. After an action is performed, the machine is in a state that depends only on the action and the previous state. Thus, we model actions as automorphisms on the set of states. After performing some actions, the user looks at the machine from some viewpoint and sees some observable aspect of the machines state. We capture this as a set of viewpoints, each of which is a function from the set of states to the set of observables. Because this notion of machine does not cover everyone's intuition, we call our machines *action-view machines* when there is any chance of ambiguity.

Definition 3.1.1 (Action-View Machine, AVM) *An* action-view machine*, or* AVM*, is a 5-tuple,* $\langle \mathbb{S}, s_0, \mathbb{A}, \mathbb{O}, \mathbb{V} \rangle$*, where*

- $\mathbb{S}$ *is the set of* states*,*
- s_0 *is the* initial state*,*
- $\mathbb{A} \subseteq (\mathbb{S} \to \mathbb{S})$ *is the set of* actions*,*
- $\mathbb{O}$ *is a set of* observables*, and*
- $\mathbb{V} \subseteq (\mathbb{S} \to \mathbb{O})$ *is the set of* viewpoints*.*

It is convenient to choose one fixed but arbitrary AVM for the rest of this thesis, and call it $\mathbb{M}$. In doing so, we have chosen fixed but arbitrary components, and reserve the five symbols, $\mathbb{S}$, s_0, $\mathbb{A}$, $\mathbb{O}$, and $\mathbb{V}$, to refer to them. Thus,

$$\mathbb{M} = \langle \mathbb{S}, s_0, \mathbb{A}, \mathbb{O}, \mathbb{V} \rangle.$$

Example 3.1.3 (The Abacus) *We describe an action-view machine corresponding to a modern Japanese Abacus with* 15 *rods, numbered* 0 *through* 14*. Its states are the positions of all the beads when they are being used properly (*i.e.*, slid all the way up or all the way down, never in between), so there are* 10^{15} *states. The initial state is when all the beads are at the bottom (the state interpreted as the value* 0*). An action,* $\mathtt{add_a}(j, x)$ *on the abacus consists of adding* j *(a value from* 0 *through* 9*) to rod* x *(which may cause a carry onto higher rods). The observables are the integers from* 0 *to* $10^{15} - 1$*. In this AVM we have a viewpoint,* $\mathtt{v_a}(j, k)$*, for each interval of rods from* j *to* k *with* $0 \leq j \leq k < 15$*. If* $\mathtt{rod_a}(j)$ *is the number of beads in the up position on rod* j*,* $(0 \leq j < 15)$*, then*

$$\mathtt{v_a}(j, k) = \sum_{i=j}^{k} \mathtt{rod_a}(j) 10^{i-j}.$$

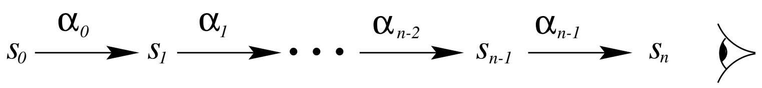

Figure 3.1: The machine starts in its initial state, actions are performed, the machine moves through a sequence of states, and the user looks at the machine.

3.2 What is a Computation?

We will follow the intuition present in statements like "use the slide-rule to compute the answer". Thus, using a machine (as described above) is the same as performing a computation. To perform a computation, we start with the machine in its initial state, perform some actions, then look at the machine from some viewpoint. It is our view that the result of a computation is specified by three things: a machine, a list of actions, and a viewpoint (see Figure 3.1).

To perform a computation is to do everything required to obtain its result. Thus, the computation itself is all the work performed by the user and all of the state changes (hidden or visible) within the machine. Since this work is specified by the same things that specify the result of the computation, we will abstract a little and say a computation is a machine, a list of actions, and a viewpoint. Whenever the machine is clear from context, we say that a computation is just a list of actions and a viewpoint.

When a machine is in state s and we perform action α, the machine enters the state $\alpha(s)$. When we perform a computation, the machine starts in its initial state, s_0, and we carry out a sequence of actions, α_0, ..., α_n. This causes the machine to move from s_0 to the state $\alpha_n(...\alpha_0(s_0))$. Finally, we look at the machine from viewpoint v, and obtain the result $v(\alpha_n(...\alpha_0(s_0)))$.

But the ellipsis is not formally defined, and Intuition 1 suggests that we must quantify over all computations. To deal with this, we make the sequence of actions an object in our model, and introduce the `do` and `compute` functions.

Definition 3.2.1 (do and compute) *To handle lists of actions,*

$$\texttt{do} : \mathbb{S} \times \mathbb{A}^* \rightarrow \mathbb{S}$$

is defined by

$$\texttt{do}(s, [\,]) = s, \text{ and}$$
$$\texttt{do}(s, \alpha :: t) = \texttt{do}(\alpha(s), t),$$

while

$$\texttt{compute} : \mathbb{S} \times \mathbb{A}^* \times \mathbb{V} \rightarrow \mathbb{O}$$

is defined by

$$\texttt{compute}(s, l, v) = v(\texttt{do}(s, l)).$$

Example 3.2.1 (The Abacus, continued) *Recall the abacus from Example 3.1.3. Here is one way we can use our abacus to compute $27 + 56$. The list of actions is*

$$[\texttt{add}_\texttt{a}(7,3), \texttt{add}_\texttt{a}(2,4), \texttt{add}_\texttt{a}(6,3), \texttt{add}_\texttt{a}(5,4)].$$

The viewpoint is $\texttt{v}_\texttt{a}(3,5)$. This computation uses rod 3 for the 1's place. The third action, $\texttt{add}_\texttt{a}(6,3)$, causes a carry from rod 3 onto rod 4.

Now we are able to quantify over computations using expressions similar to this:

$$\forall l : \mathbb{A}^*, v : \mathbb{V} \cdot P(l, v).$$

(Here P is some predicate, and we assume the machine is known from context). We also have a lemma relating `compute`, `append`, and `do`.

Lemma 3.2.1 *If s is a state, v a viewpoint, and l_1 and l_2 are lists of actions, then*

$$\texttt{compute}(s, \texttt{append}(l_1, l_2), v) = \texttt{compute}(\texttt{do}(s, l_1), l_2, v).$$

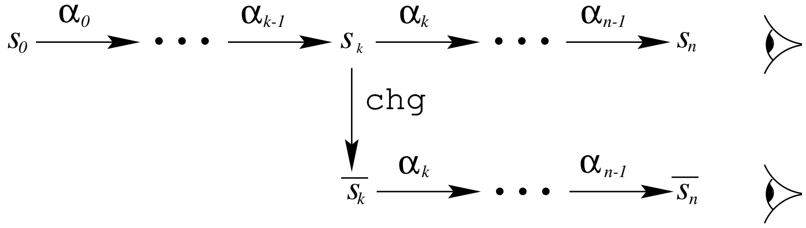

Figure 3.2: The upper eye sees the result of the original computation. The lower eye sees the result when the machine was paused and changed in the middle of the computation. $l_1 = [\alpha_0, ..., a_{k-1}]$ and $l_2 = [ak, ..., a_{n-1}]$.

3.3 What Does it Mean to Pause and Change a Machine?

Once again we consult our intuition, and conclude that to pause the machine is to stop performing actions on it, with an expectation that we will resume later. After resuming and completing the computation, we get the same result as if we had not paused. There is no reason to expect this intuition to apply to all machines, but it does hold for AVMs.

If the computation is given by the list of actions $\texttt{append}(l_1, l_2)$ and the viewpoint v, then we model the concept of pausing after l_1 with the notion of evaluating the term $\texttt{do}(s_0, l_1)$. Allowing the machine to resume corresponds to substituting the value of this term for x in the expression $\texttt{compute}(x, l_2, v)$. Then Lemma 3.2.1 may be interpreted as saying that pausing and resuming does not change the result of the computation. If we change the state from $\texttt{do}(s_0, l_1)$ to $\overline{s}$ while the machine is paused, allowing it to resume after the change corresponds to substituting $\overline{s}$ for x.

In Figure 3.2, the pause occurs when the machine is in state s_k. Then the state is changed to $\overline{s_k}$, and the machine is allowed to resume running. If the change has no effect on the computation, then $v(s_n) = v(\overline{s_n})$. We say that, at the point after l_1, changing to $\overline{s_k}$ preserves the result of the computation specified by $\texttt{append}(l_1, l_2)$ and v.

Definition 3.3.1 (`preservesResult`) *The predicate*

$$\texttt{preservesResult} : \mathbb{S} \times \mathbb{A}^* \times \mathbb{S} \times \mathbb{A}^* \times \mathbb{V} \to \mathbb{B}$$

is defined by

$$\texttt{preservesResult}(s, l_1, \overline{s}, l_2, v) = \{\texttt{compute}(s, \texttt{append}(l_1, l_2), v) = \texttt{compute}(\overline{s}, l_2, v)\}.$$

Example 3.3.1 (The Abacus, continued) *In Example 3.2.1, we computed* $27 + 56$ *using the actions*

$$[\texttt{add}_{\texttt{a}}(7, 3), \texttt{add}_{\texttt{a}}(2, 4), \texttt{add}_{\texttt{a}}(6, 3), \texttt{add}_{\texttt{a}}(5, 4)]$$

and the viewpoint $\texttt{v}_{\texttt{a}}(3, 5)$. *At any point in this computation, we could alter the state by moving some of the beads on rod* 0. *Since rod* 0 *does not contribute to the value returned by this viewpoint, the altered state would preserve the result of the computation.*

The following lemma tells us how `preservesResult`, `append`, and `do` are related.

Lemma 3.3.1 *If* v *is a viewpoint,* l_1, l_2 *and* l_3 *are lists of actions, and* s *and* $\overline{s}$ *are states, then*

$$\texttt{preservesResult}(s, \texttt{append}(l_1, l_2), \overline{s}, l_3, v) = \texttt{preservesResult}(\texttt{do}(s, l_1), l_2, \overline{s}, l_3, v).$$

We expect a garbage collector to transform the current memory state, so we model it with a function, δ, from one memory state to another. In the course of formalizing Intuition 1, we assert that if δ does not change reachable memory, then the new state preserves the result. This is the sense in which it is *safe* to use the garbage collector. Since δ selects a new state after the actions in l_1 have been performed, we see that $\overline{s} = \delta(\texttt{do}(s_0, l_1))$. For the computation specified by $\texttt{append}(l_1, l_2)$ and v, we require that δ satisfy

$$\texttt{preservesResult}(s_0, l_1, \delta(\texttt{do}(s_0, l_1)), l_2, v).$$

If this holds for every computation, then we say that it is always safe to pause and change the machine using δ. This leads to the following definition.

Definition 3.3.2 (`pcSafe` - Pause and Change is Safe) *The predicate*

$$\texttt{pcSafe} : (\mathbb{S} \to \mathbb{S}) \to \mathbb{B}$$

is defined by

$$\texttt{pcSafe}(\delta) = \forall l_1, l_2 : \mathbb{A}^*, v : \mathbb{V} \cdot \texttt{preservesResult}(s_0, l_1, \delta(\texttt{do}(s_0, l_1)), l_2, v).$$

This definition can be decomposed in the following useful manner. The 'pausing' part of this notion is closely related to achievability (this is often referred to as reachability, but we use that word elsewhere, and wish to avoid confusion). The 'changing while preserving' part is related to the question, "when can two states be distinguished?" We now formalize achievability and indistinguishability, and use them to restate the pause-and-change-is-safe criteria.

We say that a state is *achievable* in an AVM if there is a sequence of actions that cause the machine to move there from its initial state.

Definition 3.3.3 (`achievable`) *The predicate*

$$\texttt{achievable} : \mathbb{S} \to \mathbb{B}$$

is defined by

$$\texttt{achievable}(s) = \exists l : \mathbb{A}^* \cdot (\texttt{do}(s_0, l) = s).$$

If one viewpoint gives different observations for different states, then the states are trivially distinguishable. Even if all of the viewpoints agree on two states, there may be an action that takes them to distinguishable states. We only consider states to be indistinguishable when we always get the same observations after the same manipulations. We call this computational or observational equivalence.

Definition 3.3.4 (`computationEq`) *The predicate*

$$\texttt{computationEq} : \mathbb{S} \times \mathbb{S} \to \mathbb{B}$$

is defined by

$$\texttt{computationEq}(s_1, s_2) = \forall l : \mathbb{A}^*, v : \mathbb{V} \cdot \{\texttt{compute}(s_1, l, v) = \texttt{compute}(s_2, l, v)\}.$$

Example 3.3.2 (The Colorful-Abacus) *We extend the set of states for the abacus introduced in Example 3.1.3 by allowing the beads to be colored either black or white. The initial state of the* colorful-abacus *is the state where all the beads are black and at the bottom. The colorful-abacus has the same actions, observables and viewpoints as described in Example 3.1.3. Notice that result of any action or viewoint is independent of the color of the beads.*

A state-to-state function for the colorful-abacus will satisfy `pcSafe` *if and only if it leaves all the beads in the same position. We can pause anywhere in any computation, paint any beads black or white, as we wish, and resume without affecting the outcome of the computation.*

Since there are no actions that cause beads to change color, a state of the colorful-abacus is achievable if and only if all of the beads are black. Two states are computation equivalent if and only if all the beads are in the same position.

The criteria for computation equivalence may be restated using `preservesResult`.

Lemma 3.3.2 *If s_1 and s_2 are states, then*

$$\texttt{computationEq}(s_1, s_2) = \forall l : \mathbb{A}^*, v : \mathbb{V} \cdot \texttt{preservesResult}(s_1, [\,], s_2, l, v).$$

We use these last two definitions to rewrite `pcSafe`. The new form is easier to work with because it does not need `append` and uses fewer quantifiers. More to the point, the quantifiers that reappear when these definitions are expanded have reduced scope, which tends to simplify proofs that use the following theorem.

Theorem 3.3.1 *If $\delta : \mathbb{S} \to \mathbb{S}$, then*

$$\texttt{pcSafe}(\delta) = \forall s : \mathbb{S} \cdot \texttt{achievable}(s) \Rightarrow \texttt{computationEq}(s, \delta(s)).$$

Chapter 4

Memory

4.1 What is Memory?

In Funk & Wagnalls [2], the first definition for *memory* is

> *The mental function or capacity of recalling or recognizing previously learned behavior or past experience.*

We take memory to be a machine that provides the function of recalling a severely limited type and quantity of past experiences. In the next two examples, we present an action-view machine similar to the usual model of physical computer memory.

Example 4.1.1 (The State of a Random Access Memory Machine) *At any given instant, the state of a real computer's RAM may be comfortably thought of as a function from some range of integers into another range of integers. We will be slightly more abstract, and describe a RAM state as being a function from the set of* addresses*, A_r, to the set of* data *objects, D, so the set of states is $\mathbb{S}_r = (A_r \to D)$.*

In order to cast memory as an AVM, we need an initial state. For our purposes, any element of $\mathbb{S}_r$ will do. $\mathtt{ram}_0$ denotes the initial state of our RAM machine.

Notice that we have reused the symbol $\mathbb{S}$ from Definition 3.1.1. In this instance it carries the r subscript ro to remind the reader that it is the set of states for the RAM machine. Since we use a different set of addresses later, this set is similarly decorated. This mnemonic device is used throughout this thesis: objects with similar roles have the same base symbol, and a distinguishing subscript.

The actions, viewpoints and observables are the interface to the machine. It is often convenient to define families of actions and viewpoints by means of action-valued and viewpoint-valued functions. We call such functions interface-functions, and their formal arguments interface-parameters. The elements of the domains of the interface-parameters are called interface-objects.

Definition 4.1.1 (Interface-Function, Interface-Parameter, and Interface-Object) *If $\mathbb{M} = \langle \mathbb{S}, s_0, \mathbb{A}, \mathbb{O}, \mathbb{V} \rangle$ is an AVM, then an interface-function for $\mathbb{M}$ is a function for which the range is a subset of either $\mathbb{A}$ or $\mathbb{V}$. When one is given by means of the form* "$f : D_0 \times ... \times D_n \rightarrow R$ is defined by $f(p_0, ..., p_n) = ...$" *(where the co-domain R is either $\mathbb{A}$ or $\mathbb{V}$), the p_i are the interface-parameters, and the elements of $D_0 \cup ... \cup D_n$ are the interface-objects.*

Example 4.1.2 (The Interface to a Random Access Memory Machine) *An action in our RAM machine updates the data value at one address. When the input memory state is $m_{in} \in \mathbb{S}_r$, and the action is setting the data value at a_0 to d, then the output state, $m_{out} \in \mathbb{S}_r$, is defined by*

$$m_{out}(a) = \texttt{update}(m_{in}, a_0, d).$$

We can describe the set of all actions using the interface-function

$$\texttt{setData}_r : A_r \times D \rightarrow (\mathbb{S}_r \rightarrow \mathbb{S}_r)$$

defined by

$$\texttt{setData}_r(a_0, d) = \lambda m_{in} : \mathbb{S}_r \cdot \lambda a : A_r \cdot \texttt{update}(m_{in}, a_0, d).$$

For this machine, the set of actions is the range of $\texttt{setData}_r$, $\Re(\texttt{setData}_r)$, which we label $\mathbb{A}_r$.

The set of observables is $\mathbb{O}_r = D$.

In this machine, a viewpoint looks at the value stored at one address. Thus, each viewpoint is characterized by a single address, and is defined to be the application of the state to the address. We define the interface-function

$$\texttt{getData}_r : A_r \rightarrow (\mathbb{S}_r \rightarrow D)$$

by

$$\texttt{getData}_r(a) = \lambda m : \mathbb{S}_r \cdot m(a)$$

so the set of viewpoints is $\mathbb{V}_r = \Re(\texttt{getData}_r)$.

In summary, our RAM machine is the 5-tuple

$$\mathbb{M}_r = \langle \mathbb{S}_r, \texttt{ram}_0, \Re(\texttt{setData}_r), D, \Re(\texttt{getData}_r)\rangle,$$

or

$$\mathbb{M}_r = \langle \mathbb{S}_r, \texttt{ram}_0, \mathbb{A}_r, \mathbb{O}_r, \mathbb{V}_r\rangle.$$

The random access memory machine described in this section allows us to recall the most recent value placed at (*i.e.*, experience involving) each address.

4.2 What is a Pointer?

In order to discuss garbage collection, we need a richer notion of memory. As a step in that direction, we extend our simple memory to one in which pointers (or addresses) make sense. For our purposes, pointers have two critical features: they can be stored in memory, and each pointer indicates a location in memory. Many modern memory architectures are designed so that pointers and "machine-words" are interchangeable, but many of the designers of garbage collecting languages take great pains to hide all concerns about pointers and "machine-words" from the user. In implementations of

GCLs, there are regions of physical memory that hold data and/or pointers: they are commonly called *cells*. In this context, pointers indicate cells.

There are many kinds of cells used in production GCLs. Some cells contain data, some contain pointers, and some have both. The data may be structured, and the number of pointers may be fixed or variable. Efficient implementations are obtained by means of subtle tricks. They take advantage of obscure things like the structure of the cells, interpreting small addresses as integers, and even exploiting the low bits on machines that require data structures to be word-aligned. We use an abstract cell to hide all that: our model contains objects representing the contents of cells, but we do not model the cells themselves.

Each of our cells consists of exactly one data field, and some fixed number of pointer fields. The set of *data* objects is called D, and the data field of a cell holds one element of D. Each of the pointer fields has a name; the set of *pointer field names* is called N_{pf}. We assume that for each cell, there is exactly one *cell-address*, and call the set of cell-addresses A_c. Each of the pointer fields of a cell holds one cell-address. These three sets are fixed and non-empty, but otherwise arbitrary, for the remainder of this thesis.

Each cell contains more than just a data object and some cell-addresses. It also holds a relationship between the names of the pointer fields and the cell-addresses in them. We call this *pointer field map* `pfMap`. Modeling this mapping explicitly makes it easy to quantify over pointer field names, and allows us to avoid mentioning any specific name.

Definition 4.2.1 (Cell-Contents) *A* cell-contents *object is a record with two fields:* `data` *and* `pfMap`*. When c is a cell-contents object,*

- $c.\mathtt{data} \in D$ *is its* data*, and*
- $c.\mathtt{pfMap} : N_{pf} \to A_c$ *tells us which* cell-address *is in each* pointer field.

The set of all such cell-contents records is denoted C.

In Example 4.1.1, we described memory states as functions from addresses to data. Now we extend this to functions from cell-addresses to cell-contents records.

Definition 4.2.2 (Cell-Memory State) *A* cell-memory state *is a function from A_c to C. We denote the set of all cell-memory states by $\vec{C}$.*

Please note that we do not define cell objects. A cell cannot be identified with its contents, because more than one cell can have the same contents. It is not wrong to think of a cell as being its address, but that would lead us to say that cells contain other cells, which is counter-intuitive. Perhaps the most accurate view is: a cell is a function from $\vec{C}$ to C, defined by applying the cell-memory state to a specific cell-address, so the cell at address a is $\lambda m : \vec{C} \cdot m(a)$. This puts cells and cell-addresses in one-to-one correspondence, as required by intuition and common usage. From now on, whenever we refer to addresses, we mean cell-addresses.

We need to be able to set and read the data and pointer fields of cells. Manipulating the data field is particularly easy. Suppose c is a cell-contents record. Then $c\ \texttt{with data} \leftharpoondown d$ is an element of C that differs only in the data field (if at all), and $c.\texttt{data}$ is the contents of the data field. Manipulating pointer fields is more cumbersome. The value of pointer field n is just $c.\texttt{pfMap}(n)$, but updating the pointer field to hold the address a requires

$$c\ \texttt{with pfMap} \leftharpoondown \lambda n' : N_{pf} \cdot \texttt{if}\ (n' = n)\ \texttt{then}\ a\ \texttt{else}\ c.\texttt{pfMap}(n')$$

This last expression is simplified by means of `update`:

$$c\ \texttt{with pfMap} \leftharpoondown \texttt{update}(c.\texttt{pfMap}, n, a).$$

Updating cell-memory states requires longer expressions, but nothing new. To set the data at address a in the cell-memory state m to the value d, we use the expression

$$\texttt{update}(m, a, m(a)\ \texttt{with data} \leftharpoondown d).$$

To set the pointer field named n at the same address to the new value a', we use

$$\texttt{update}(m, a, m(a)\ \texttt{with pfMap} \leftharpoondown \texttt{update}(m(a).\texttt{pfMap}, n, a')).$$

By now it should be clear that manipulating memory at this level is completely unacceptable, which motivates the use of several layers of intermediate definitions.

Definition 4.2.3 (Cell-Contents Interface) *The following functions* [1] *are used to manipulate cell-contents records:*

- $\texttt{setData}_\texttt{C} : D \times C \to C$ *is defined by* $\texttt{setData}_\texttt{C}(d, c) = c\ \texttt{with data} \leftharpoondown d$*, which sets the data field,*

- $\texttt{getData}_\texttt{C} : C \to D$ *is defined by* $\texttt{getData}_\texttt{C}(c) = c.\texttt{data}$*, which gets the contents of the data field,*

- $\texttt{setPF}_\texttt{C} : A_c \times N_{pf} \times C \to C$ *is defined by*

$$\texttt{setPF}_\texttt{C}(a, n, c) = c\ \texttt{with pfMap} \leftharpoondown \texttt{update}(c.\texttt{pfMap}, n, a),$$

  which sets the pointer field named n,

- $\texttt{getPF}_\texttt{C} : N_{pf} \times C \to A_c$ *is defined by* $\texttt{getPF}_\texttt{C}(n, c) = c.\texttt{pfMap}(n)$*, which gets the contents of the pointer field named n, and*

- $\texttt{floodPFs}_\texttt{C}$[2] $: A_c \times C \to C$ *is defined by*

$$\texttt{floodPFs}_\texttt{C}(a, c) = c\ \texttt{with pfMap} \leftharpoondown \lambda n : N_{pf} \cdot a,$$

  which sets all of the pointer fields to the same address.

[1] There are quite a lot of definitions in this thesis. To reduce the load on the reader's memory, the glossary in Appendix B includes the page where each symbol is defined.

[2] We are flood-filling the pointer fields with an address in the same way paint programs flood-fill a region with a color.

Definition 4.2.4 (Cell-Memory Interface) *Here are the functions for working with cell-memory states. First, we have functions that manipulate the contents of entire cells within cell-memory states.*

- $\texttt{setContents}_{\vec{\texttt{C}}} : C \times A_c \times \vec{C} \to \vec{C}$ *is defined by*

  $$\texttt{setContents}_{\vec{\texttt{C}}}(c, a, m) = \texttt{update}(m, a, c),$$

  which replaces the contents of the cell at address a, and

- $\texttt{getContents}_{\vec{\texttt{C}}} : A_c \times \vec{C} \to C$ *is defined by* $\texttt{getContents}_{\vec{\texttt{C}}}(a, m) = m(a)$, *which gets the contents of the cell at address a.*

We also have functions that manipulate the fields within the cells of cell-memory states.

- $\texttt{setData}_{\vec{\texttt{C}}} : D \times A_c \times \vec{C} \to \vec{C}$ *is defined by*

  $$\texttt{setData}_{\vec{\texttt{C}}}(d, a, m) = \texttt{setContents}_{\vec{\texttt{C}}}(\texttt{setData}_{\texttt{C}}(d, \texttt{getContents}_{\vec{\texttt{C}}}(a, m)), a, m),$$

  which sets the data of the cell at a,

- $\texttt{getData}_{\vec{\texttt{C}}} : A_c \times \vec{C} \to D$ *is defined by*

  $$\texttt{getData}_{\vec{\texttt{C}}}(a, m) = \texttt{getData}_{\texttt{C}}(\texttt{getContents}_{\vec{\texttt{C}}}(a, m)),$$

  which gets the data of the cell at a,

- $\texttt{setPF}_{\vec{\texttt{C}}} : A_c \times N_{pf} \times A_c \times \vec{C} \to \vec{C}$ *is defined by*

  $$\texttt{setPF}_{\vec{\texttt{C}}}(a, n, a_0, m) = \texttt{setContents}_{\vec{\texttt{C}}}(\texttt{setPF}_{\texttt{C}}(a, n, \texttt{getContents}_{\vec{\texttt{C}}}(a_0, m)), a_0, m),$$

  which sets the field named n of the cell at a_0,

- $\texttt{getPF}_{\vec{\texttt{C}}} : N_{pf} \times A_c \times \vec{C} \to A_c$ *is defined by*

  $$\texttt{getPF}_{\vec{\texttt{C}}}(n, a, m) = \texttt{getPF}_{\texttt{C}}(n, \texttt{getContents}_{\vec{\texttt{C}}}(a, m)),$$

  which gets the pointer from the field named n of the cell at a, and

- $\texttt{floodPFs}_{\vec{\texttt{C}}} : A_c \times A_c \times \vec{C} \to \vec{C}$ *is defined by*

  $$\texttt{floodPFs}_{\vec{\texttt{C}}}(a, a_0, m) = \texttt{setContents}_{\vec{\texttt{C}}}(\texttt{floodPFs}_{\texttt{C}}(a, \texttt{getContents}_{\vec{\texttt{C}}}(a_0, m)), a_0, m),$$

  which sets all of the pointer fields of the cell at a_0 to the same address.

4.3 Which Parts of Memory are Reachable Via Chains of Pointers?

Reachability is a relative term. When achievability (Definition 3.3.3) goes under the name of reachability, we say that a machine state is reachable from the initial state via a sequence of actions. It is common to say that a memory cell is reachable from another if there is a chain of pointers that connects them. Thus, we need to identify where parts of memory are reachable from. With each memory state we associate a set of starting points called roots.

The literature on garbage collection contains phrases like “the roots are the global variables, registers or stack”, and “we can assume there is just one root”. It is routine practice to make some such assertion in passing, and then to use a simplifying assumption such as “the set of roots is fixed”. In all but one[3] treatment known to the author, it is assumed that all the roots are cells. Such assumptions can make the presentation of a GC algorithm much clearer.

Though it is possible to build a garbage collecting language in which global variables really are root cells, we cannot put a cell on the stack or in a register. Cells hold pointers to cells. If a cell is on the stack and is not garbage when the stack is popped, dangling references will be created. How would you point to a register? The best we can do is put cell-addresses on the stack and in registers, and have the collector follow them to find the root cells. We use the term *root-address* to refer to the addresses stored on the stack and

[3] In [17], instead of the traditional cells and pointers, there are expressions and free variables.

in registers and the addresses of the global-variable cells. The term *root-name* covers the stack locations holding cell-addresses, the names of the registers holding cell-addresses, and the names of the global-variable cells. The *root-cells* are the cells indicated by root-addresses. The author has not seen this distinction between root-cells, root-addresses and root-names in the literature.

The informal meaning of *root-name* is somewhat complicated. We abstract it by saying that N_r is the set of all root-names. This set will be fixed but arbitrary for the remainder of this thesis. Because the stack changes over time, so does the set of *active* or *in use* root-names. To model this, we introduce *rooted-memory states*, which augment cell-memory states with a subset of N_r and a map that indicates the root-addresses.

Definition 4.3.1 (Rooted-Memory State) *A* rooted-memory state *is a record with three fields:* `activeNames`, `rnMap`, *and* `cm`. *When m is a rooted-memory state,*

- $m.\texttt{activeNames} \subseteq N_r$ *is its set of* active *root-names,*
- $m.\texttt{rnMap} : N_r \to A_c$ *indicates the root-addresses, and*
- $m.\texttt{cm} \in \vec{C}$ *is a cell-memory state.*

The set of all such rooted-memory states is denoted R.

When $a = m.\texttt{rnMap}(r)$, we say that a is at r in m, or that r leads to a in m. When we want to say this formally, we use the following definition.

Definition 4.3.2 (The Cell Address at a Root Name) *The function*

$$\texttt{caAtRN}_{\texttt{R}} : N_r \times R \to A_c$$

is defined by

$$\texttt{caAtRN}_{\texttt{R}}(r, m) = m.\texttt{rnMap}(r).$$

We often use the phrase "the cell that r leads to" instead of the more precise and more awkward "the contents of the cell that r leads to", or $\texttt{getContents}_{\vec{\texttt{C}}}(\texttt{caAtRN}_{\texttt{R}}(r, m), m.\texttt{cm})$. When we say "the data that r leads to", we mean $\texttt{getData}_{\vec{\texttt{C}}}(\texttt{caAtRN}_{\texttt{R}}(r, m), m.\texttt{cm})$.

In addition to roots, reachability depends on connectedness. A cell-memory state defines a labeled, directed graph: the nodes are the cell-addresses, the labels are the pointer field names, and we have an edge or connection from address a_{start} to address a_{end} if a_{end} appears in one of the pointer-fields of the cell indicated by a_{start}. In the terminology of graph theory, we say that the nodes reachable from a root are the ones in the same connected component as the root.

Definition 4.3.3 (Cell-Memory Connected) *The predicate*

$$\texttt{connected}_{\vec{\texttt{C}}} : \vec{C} \times A_c \times A_c \to \mathbb{B}$$

is defined by

$$\texttt{connected}_{\vec{\texttt{C}}}(m, a_{start}, a_{end}) = \exists n : N_{pf} \cdot a_{end} = \texttt{getPF}_{\vec{\texttt{C}}}(n, a_{start}, m).$$

The following lemma tells us how connectedness is affected by changing data or pointer fields. Suppose the pointer field n_0 of the cell at a_0 is changed to a. If a_{start} is not a_0, then the edges leaving it are unaffected, and it remains connected to the same addresses as before. This change ensures that a_0 becomes connected to a, while addresses connected to a_0 via other pointer fields remain connected. As expected, changing data has no effect on connectedness.

Lemma 4.3.1 *If a, a_0, a_{start} and a_{end} are addresses, n_0 is a pointer field name, d is a data object, and m is a cell-memory state, then*

$\texttt{connected}_{\vec{\texttt{C}}}(\texttt{setPF}_{\vec{\texttt{C}}}(a, n_0, a_0, m), a_{start}, a_{end})$

$$\begin{aligned} = \quad & \texttt{if} && (a_{start} \neq a_0) \\ & \quad \texttt{then} && \texttt{connected}_{\vec{\texttt{C}}}(m, a_{start}, a_{end}) \\ & \quad \texttt{else} && (a_{end} = a) \vee \exists n : N_{pf} \cdot \{n \neq n_0\} \wedge \{a_{end} = \texttt{getPF}_{\vec{\texttt{C}}}(n, a_0, m)\}, \end{aligned}$$

and

$$\mathtt{connected}_{\vec{\mathtt{C}}}(\mathtt{setData}_{\vec{\mathtt{C}}}(d, a, m), a_{start}, a_{end}) = \mathtt{connected}_{\vec{\mathtt{C}}}(m, a_{start}, a_{end}).$$

We say that a cell-address is reachable in a rooted-memory state if it is a root or if it is reachable from a root by means of $\mathtt{connected}_{\vec{\mathtt{C}}}$.

Definition 4.3.4 (Reachable in a Rooted-Memory State) *The predicate*

$$\mathtt{reachable}_{\mathtt{R}} : R \times A_c \to \mathbb{B}$$

is defined by

$$\begin{aligned} &\mathtt{reachable}_{\mathtt{R}}(m, a) \\ &\quad = \mathtt{reachable}(\{\lambda a_{start}, a_{end} : A_c \cdot \mathtt{connected}_{\vec{\mathtt{C}}}(m.\mathtt{cm}, a_{start}, a_{end})\}, \\ &\qquad\qquad \Im(m.\mathtt{rnMap}, m.\mathtt{activeNames}), a). \end{aligned}$$

In this definition, the set of root addresses is the set of addresses that active root names lead to: the image of the set of active names under $m.\mathtt{rnMap}$, $\Im(m.\mathtt{rnMap}, m.\mathtt{activeNames})$. Recall from Chapter 2 that `reachable` takes a binary relation, which we obtain by currying $\mathtt{connected}_{\vec{\mathtt{C}}}$.

Later, we use rooted-memory states as components of the states for an AVM. The actions of the AVM change the rooted-memory states by adding and removing root names, changing how they are associated with cells, and altering the contents of cells. The following functions support defining those actions.

Definition 4.3.5 (Rooted-Memory Manipulations) *Functions for manipulating rooted-memory states are as follows:*

- $\mathtt{addRN}_{\mathtt{R}} : A_c \times N_r \times R \to R$ *is defined by*

  $$\begin{aligned} \mathtt{addRN}_{\mathtt{R}}(a, r, m) = m \ \mathtt{with} \quad & \mathtt{activeNames} \leftharpoonup r \ \mathtt{insert} \ m.\mathtt{activeNames}, \\ & \mathtt{rnMap} \leftharpoonup \mathtt{update}(m.\mathtt{rnMap}, r, a), \end{aligned}$$

  which adds the root-name r, making it refer to a,

- $\texttt{removeRN}_{\texttt{R}} : N_r \times R \to R$ *is defined by*

$$\texttt{removeRN}_{\texttt{R}}(r, m) = m \;\texttt{with activeNames} \leftharpoonup m.\texttt{activeNames delete}\; r,$$

  which removes the root-name r,

- $\texttt{assignRN}_{\texttt{R}} : N_r \times A_c \times R \to R$ *is defined by*

$$\texttt{assignRN}_{\texttt{R}}(r, a, m) = m \;\texttt{with rnMap} \leftharpoonup \texttt{update}(m.\texttt{rnMap}, r, a),$$

  which causes the root-name r to refer to a,

- $\texttt{floodPFs}_{\texttt{R}} : A_c \times A_c \times R \to R$ *is defined by*

$$\texttt{floodPFs}_{\texttt{R}}(a, a_0, m) = m \;\texttt{with cm} \leftharpoonup \texttt{floodPFs}_{\vec{\texttt{C}}}(a, a_0, m.\texttt{cm}),$$

  which sets all of the pointer fields of the cell at a_0,

- $\texttt{setPF}_{\texttt{R}} : N_r \times N_{pf} \times N_r \times R \to R$ *is defined by*

$$\begin{aligned}&\texttt{setPF}_{\texttt{R}}(r, n, r_{affected}, m)\\ &\quad = m \;\texttt{with cm} \leftharpoonup \texttt{setPF}_{\vec{\texttt{C}}}(\texttt{caAtRN}_{\texttt{R}}(r, m), n, \texttt{caAtRN}_{\texttt{R}}(r_{affected}, m), m.\texttt{cm}),\end{aligned}$$

  which puts the address named by r in the pointer field named n of the cell indicated by $r_{affected}$ (see Figure 4.1),

- $\texttt{followPFN}_{\texttt{R}} : N_r \times N_{pf} \times N_r \times R \to R$ *is defined by*

$$\begin{aligned}&\texttt{followPFN}_{\texttt{R}}(r_{affected}, n, r_{start}, m)\\ &\quad = m \;\texttt{with}\\ &\qquad \texttt{rnMap} \leftharpoonup \texttt{update}(m.\texttt{rnMap}, r_{affected}, \texttt{getPF}_{\vec{\texttt{C}}}(n, \texttt{caAtRN}_{\texttt{R}}(r_{start}, m), m.\texttt{cm})),\end{aligned}$$

  which follows r_{start} in order to extract an address from a cell, and causes $r_{affected}$ to refer to that address (see Figure 4.2),

- $\texttt{setData}_{\texttt{R}} : D \times N_r \times R \to R$ *is defined by*

$$\texttt{setData}_{\texttt{R}}(d, r, m) = m \;\texttt{with cm} \leftharpoonup \texttt{setData}_{\vec{\texttt{C}}}(d, \texttt{caAtRN}_{\texttt{R}}(r, m), m.\texttt{cm}),$$

  which sets the data field of the cell indicated by r.

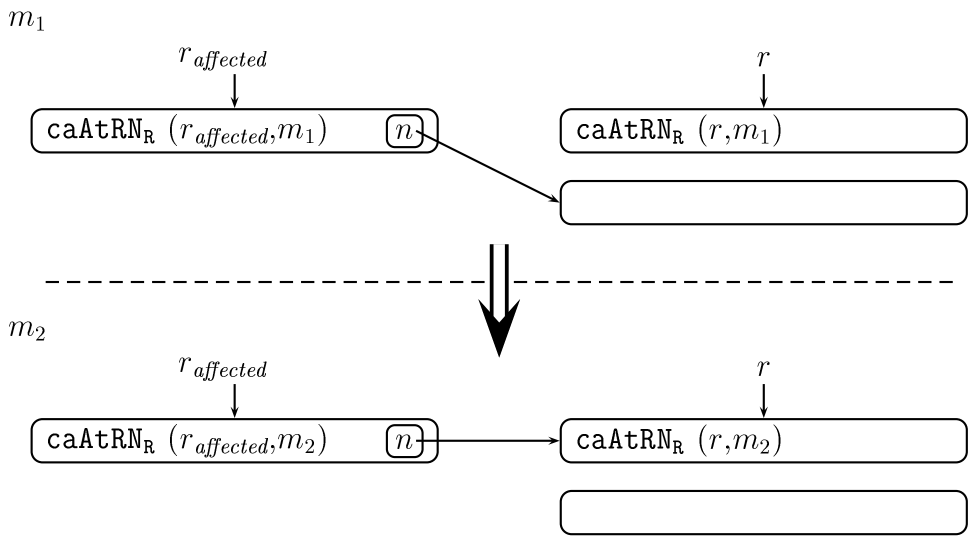

Figure 4.1: $\texttt{setPF}_\texttt{R}$ transforms the top into the bottom: $m_2 = \texttt{setPF}_\texttt{R}(r, n, r_{affected}, m_1)$. The cells, root-names, and root-cells are fixed, but the content of one pointer field changes.

4.4 What is a Machine with Memory and Pointers?

So far, we have a state in which pointers and reachability make sense. We also have a collection of functions for working with rooted-memory states. In this section, we encapsulate these in an action-view machine. We call this new machine the *managed-memory machine* or $\mathbb{M}_m$.

Our $\mathbb{M}_m$ must meet two goals. First and foremost, it must be a convincing example of the memory models presented by garbage collecting languages. We hope that, by the end of this section, readers familiar with GCLs will agree that $\mathbb{M}_m$ meets this goal. Second, it must facilitate the formalization of Intuition 1. For this, it must restrict the ways in which computations manipulate memory so that they are independent of the unreachable parts of memory.

Consider how the interface functions of the ram-machine, $\mathbb{M}_r$, take address parameters. As a result, there is an action and a view for every address. Thus, when we pause a computation of $\mathbb{M}_r$, there is nothing in its state to tell us which parts of memory the rest

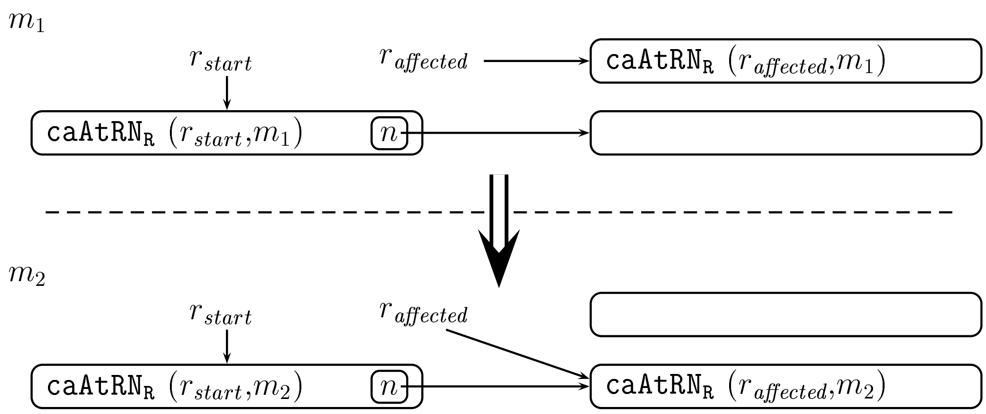

Figure 4.2: $\texttt{followPFN}_{\texttt{R}}$ takes m_1 to m_2: $m_2 = \texttt{followPFN}_{\texttt{R}}(r_{affected}, n, r_{start}, m_1)$. The cells, root-names, and pointer fields are fixed, but one root-name refers to a different cell.

of the computation depends on. If we do not examine the remainder of the computation, then we cannot safely change the state.

The interface functions of our $\mathbb{M}_m$ must not take cell-address parameters. Otherwise, a computation could depend on an unreachable address, changing the contents of unreachable memory would no longer be safe, and Intuition 1 would be false. Each state of $\mathbb{M}_m$ is defined to be a rooted-memory state and a set of unused cell-addresses. This permits us to restrict all the parameters of its interface functions to the sets N_r, N_{pf}, and D. In particular, the set of unused addresses permits us to define actions that allocate cells without exposing their addresses to the user. Thus, our $\mathbb{M}_m$ makes addresses invisible to the user.

Definition 4.4.1 (Managed-Memory State) *A* managed-memory state *is a record with two fields:* `rm` *and* `unused`*. When m is a managed-memory state,*

- *m.`rm` $\in R$ is its rooted-memory state, and*
- *m.`unused` $\subseteq A_c$ is the set of cell-addresses available for allocation.*

The set of all such managed-memory states is denoted $\mathbb{S}_m$.

What cell-address should a new root-name indicate? What should be in the pointer fields of a newly allocated cell? We are not concerned with the concept of uninitialized variables in this thesis, so we choose some arbitrary default address, $a_{default} \in A_c$ (recall that we assumed $A_c \neq \emptyset$). This roughly corresponds to the nil object of Lisp or the NULL pointer of C. In C, any attempt to get data or another pointer via the NULL pointer crashes the program. In some versions of Lisp, any attempt to access a field of the nil object returns nil; in other versions, it causes an exception to be thrown or the program to terminate with an error message. Versions of Lisp that return nil as the value of a field of nil, do not allow the field to be updated. These examples show that nil is not like any other object, and NULL is not like any other pointer. We choose to abstract away these special cases and use $a_{default}$, which is just like every other cell-address. We call it a default because

- when the machine starts, all of the pointer fields of the cell indicated by $a_{default}$, contain $a_{default}$,
- when a new root-name is introduced, it initially refers to $a_{default}$, and
- when a new cell is allocated, all of its pointer fields initially contain $a_{default}$.

For the remainder of this thesis, $a_{default}$ denotes an arbitrary but fixed member of A_c.

Unlike nil and NULL, our default address refers to a real cell. Its data and pointer fields can be accessed and updated freely. We choose this approach because it eliminates many special cases in our definitions; in exchange, we accept that some proofs based on this formalization are more difficult.

Now we choose an initial state for our $\mathbb{M}_m$. It must satisfy the following constraints:

- it has no active root-names,
- all of the pointer-fields of the cell indicated by $a_{default}$ hold $a_{default}$, and
- $a_{default}$ is the only address not available for allocation.

Definition 4.4.2 (Construction of the Initial Managed-Memory State) *We now construct a managed-memory state, which we call m_0, and use for the* initial *state of $\mathbb{M}_m$. Choose some fixed but arbitrary $m' \in \mathbb{S}_m$. Put $a_{default}$ in all of the pointer-fields of the cell at $a_{default}$, obtaining*

$$m'' = m' \texttt{ with rm} \leftharpoonup \texttt{floodPFs}_{\texttt{R}}(a_{default}, a_{default}, m'.\texttt{rm}).$$

Ensure there are no active root-names thus:

$$m''' = m'' \texttt{ with rm} \leftharpoonup (m''.\texttt{rm with activeNames} \leftharpoonup \emptyset).$$

Now arrange that all addresses except $a_{default}$ are unused. We define

$$m_0 = m''' \texttt{ with unused} \leftharpoonup A_c \texttt{ delete } a_{default}.$$

Now we define the actions and viewpoints for our $\mathbb{M}_m$. The actions fall into six categories, which correspond to six interface functions: `addRN`, `removeRN`, `new`, `setPF`, `followPFN`, and `setData`. We define the viewpoints using the interface function `getData`. Notice the similarity to the interface described in Section 1.4.

Adding a root-name inserts it in `activeNames`. At this point `rnMap` must associate the name with the default cell-address. Adding a root-name should have no other effects. Removing a root-name takes it out of `activeNames`; since we only care about the values of `rnMap` on the set `activeNames`, it has no other effect.

Definition 4.4.3 (`addRN` and `removeRN` Actions) *The interface-function*

$$\texttt{addRN} : N_r \to (\mathbb{S}_m \to \mathbb{S}_m)$$

is defined by

$$\texttt{addRN}(r) = \lambda m \cdot m \texttt{ with rm} \leftharpoonup \texttt{addRN}_{\texttt{R}}(a_{default}, r, m.\texttt{rm}).$$

The set of add root-name actions is the range of `addRN`, $\Re(\texttt{addRN})$. *The interface-function*

$$\texttt{removeRN} : N_r \to (\mathbb{S}_m \to \mathbb{S}_m)$$

is defined by

$$\texttt{removeRN}(r) = \lambda m \cdot m \texttt{ with } \texttt{rm} \leftharpoonup \texttt{removeRN}_{\texttt{R}}(r, m.\texttt{rm}).$$

The set of remove root-name actions is $\Re(\texttt{removeRN})$.

To allocate a new cell in managed-memory state m, we remove an address from $m.\texttt{unused}$ and associate it with a root-name. We don't care which address, but we need a formal way to choose one. To this end, we define a `choice` function taking the power set of A_c to A_c. `choice` associates each nonempty subset x of A_c with an element of x. For the empty set, `choice` returns an arbitrary element of A_c.

Definition 4.4.4 (`choice`) *The function*

$$\texttt{choice} : \wp(A_c) \to A_c$$

is some fixed but arbitrary function that satisfies

$$x \subseteq A_c \land x \neq \emptyset \Rightarrow \texttt{choice}(x) \in x.$$

We show that such a `choice` function exists by means of the Well-Ordering Principle, which states that there is a well-ordering for every set. (This principle is equivalent to the Axiom of Choice. It is not needed if A_c is countable, as an injection to the natural numbers induces a well-ordering.) If ω is a well-ordering on A_c, then every non-empty subset of A_c has a least element with respect to ω. An example of a valid `choice` function is one that takes the empty set to $a_{default}$, and each non-empty set to its least element with respect to ω.

After choosing a cell to allocate, and associating its address with an active root-name, all of the addresses in its pointer fields become reachable. Since the contents of these fields is initially arbitrary, and we don't want to risk making an unused cell reachable, we set all of them to $a_{default}$.

In order to avoid the issue of use-before-initialization, we choose some arbitrary default data value, $d_{default} \in D$. When `new` allocates a cell, the cell's data field is set to $d_{default}$.

Definition 4.4.5 (`new` Actions) *The interface-function*

$$\mathtt{new} : N_r \rightarrow (\mathbb{S}_m \rightarrow \mathbb{S}_m)$$

is defined by

$$\begin{aligned}\mathtt{new}(r) = \lambda m : \mathbb{S}_m \cdot\ & \mathtt{let}\ a = \mathtt{choice}(m.\mathtt{unused})\ \mathtt{in} \\ & (m\ \mathtt{with} \\ & \quad \mathtt{rm} \leftharpoondown \mathtt{setData}_\mathtt{R}(d_{default}, r, \mathtt{assignRN}_\mathtt{R}(r, a, \mathtt{floodPFs}_\mathtt{R}(a_{default}, a, m.\mathtt{rm}))), \\ & \quad \mathtt{unused} \leftharpoondown m.\mathtt{unused}\ \mathtt{delete}\ a).\end{aligned}$$

The set of new-cell actions is $\Re(\mathtt{new})$.

Our managed-memory machine provides two ways to manipulate pointers. The `setPF` interface-function defines actions that copy the cell-address indicated by a root-name into a pointer-field of a cell reached via a root-name. The `followPFN` interface defines actions that do the opposite, making the contents of a pointer-field become a root-address. These are their only effects.

Definition 4.4.6 (`setPF` and `followPFN` Actions) *The interface-function*

$$\mathtt{setPF} : N_r \rightarrow N_{pf} \rightarrow N_r \rightarrow (\mathbb{S}_m \rightarrow \mathbb{S}_m)$$

is defined by

$$\mathtt{setPF}(r, n, r_{affected}) = \lambda m : \mathbb{S}_m \cdot m\ \mathtt{with}\ \mathtt{rm} \leftharpoondown \mathtt{setPF}_\mathtt{R}(r, n, r_{affected}, m.\mathtt{rm}).$$

The set of set-pointer-field actions is $\Re(\mathtt{setPF})$.

The interface-function

$$\mathtt{followPFN} : N_r \rightarrow N_{pf} \rightarrow N_r \rightarrow (\mathbb{S}_m \rightarrow \mathbb{S}_m)$$

is defined by

$$\mathtt{followPFN}(r_{affected}, n, r_{start}) = \lambda m : \mathbb{S}_m \cdot m\ \mathtt{with}\ \mathtt{rm} \leftharpoondown \mathtt{followPFN}_\mathtt{R}(r_{affected}, n, r_{start}, m.\mathtt{rm}).$$

The set of follow-pointer-field actions is $\Re(\mathtt{followPFN})$.

What good is memory without data? We write to the data field of a root-cell using the actions defined by the `setData` interface-function. These actions do nothing else.

Definition 4.4.7 (`setData` Actions) *The interface-function*

$$\mathtt{setData} : D \to N_r \to (\mathbb{S}_m \to \mathbb{S}_m)$$

is defined by

$$\mathtt{setData}(d, r) = \lambda m : \mathbb{S}_m \cdot m \ \mathtt{with}\ \mathtt{rm} \leftharpoondown \mathtt{setData_R}(d, r, m.\mathtt{rm}).$$

The set of set-data actions is $\Re(\mathtt{setData})$*.*

We also need to read the data, so we use the `getData` interface-function to define get-data viewpoints. A get-data viewpoint returns the contents of the data field of a root-cell.

Definition 4.4.8 (`getData` Viewpoints) *The interface-function*

$$\mathtt{getData} : N_r \to (\mathbb{S}_m \to D)$$

is defined by

$$\mathtt{getData}(r) = \lambda m : \mathbb{S}_m \cdot \mathtt{getData}_{\vec{\mathtt{C}}}(\mathtt{caAtRN_R}(r, m.\mathtt{rm}), m.\mathtt{rm}.\mathtt{cm}).$$

The set of get-data viewpoints is $\Re(\mathtt{getData})$*.*

In summary, we have the following definition of our managed-memory machine. Its set of states is $\mathbb{S}_m$ and its initial state is m_0. Its observables are the data values and it has the actions and viewpoints described above.

Definition 4.4.9 (Managed-Memory Machine, $\mathbb{M}_m$) *We define our managed-memory machine to be*

$$\mathbb{M}_m = \langle \mathbb{S}_m, m_0, \mathbb{A}_m, \mathbb{O}_m, \mathbb{V}_m \rangle,$$

where

$$\mathbb{A}_m = \Re(\texttt{addRN}) \cup \Re(\texttt{removeRN}) \cup \Re(\texttt{new}) \cup \Re(\texttt{setPF})$$
$$\cup\ \Re(\texttt{followPFN}) \cup \Re(\texttt{setData}),$$
$$\mathbb{O}_m = D, and$$
$$\mathbb{V}_m = \Re(\texttt{getData}).$$

Our use of $a_{default}$ forces us to change our notion of reachability slightly. We can obtain the contents of the cell at $a_{default}$ at any time by adding and immediately using a new root-name. Thus, there is a sense in which the default cell is always a root cell.

Definition 4.4.10 (Reachability in a Managed-Memory State) *The predicate*

$$\texttt{reachable}_{\texttt{m}} : \mathbb{S}_m \times A_c \to \mathbb{B}$$

is defined by

$$\begin{aligned}&\texttt{reachable}_{\texttt{m}}(m, a)\\ &\quad = \texttt{reachable}(\{\lambda a_{start}, a_{end} : A_c \cdot \texttt{connected}_{\vec{\texttt{C}}}(m.\texttt{rm.cm}, a_{start}, a_{end})\},\\ &\qquad\qquad a_{default}\ \texttt{insert}\ \Im(m.\texttt{rnMap}, m.\texttt{activeNames}), a).\end{aligned}$$

Because $a_{default}$ is always a root, everything reachable from $a_{default}$ is reachable. This is why our use of $a_{default}$ makes some proofs more complicated. We feel this is more than justified by the elimination of special cases in the formalism.

Observe that $a_{default}$ is the only reachable address in m_0. This is because m_0 has no active root names and all the pointer fields of the cell at $a_{default}$ contain $a_{default}$.

Chapter 5

Characterizing Garbage Collection

5.1 What Does it Mean to Alter Unreachable Memory While the Machine is Paused?

Intuition 1 states that we may freely alter the unreachable parts of memory. Contrapositively, we must not change the reachable bits. We rephrase Intuition 1 accordingly.

Intuition 2 *The result of the computation will be unchanged if we pause the machine, replace the memory state with one that matches the parts of the original memory that were reachable via some chain of pointers, and allow the machine to resume.*

Intuition 2 is harder to read, but it lends itself to further formalization. Here is a direct translation into the terms we have defined so far.

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \rightarrow \mathbb{S}_m \cdot \\
&\qquad (\forall m : \mathbb{S}_m, a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \{m.\texttt{rm}.\texttt{cm}(a) = \delta(m).\texttt{rm}.\texttt{cm}(a)\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta)
\end{aligned}$$

Because we find this just a little annoying to read, we introduce the following abbreviation.

Definition 5.1.1 (sameContents) *The predicate*

$$\texttt{sameContents} : A_c \times \mathbb{S}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\texttt{sameContents}(a, m_1, m_2) = \{m_1.\texttt{rm}.\texttt{cm}(a) = m_2.\texttt{rm}.\texttt{cm}(a)\}.$$

Now for our first attempt at formalization.

Formalization 5.1.1

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\qquad \{\forall m : \mathbb{S}_m, a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\} \\
&\quad \Rightarrow \texttt{pcSafe}(\delta)
\end{aligned}$$

We have a problem. δ, our potential garbage collector, may alter the map from root names to cell addresses. In the extreme, the antecedent of Formalization 5.1.1 allows $\delta(m)$ to have all root names lead to $a_{default}$, even if none of the root names in m do. A δ function that does this cannot satisfy the consequent, so Formalization 5.1.1 does not hold. We strengthen the antecedent by explicitly forbidding any change in the map from active root names to root addresses.

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\qquad (\forall m : \mathbb{S}_m, a : A_c, r : N_r \cdot \\
&\qquad\qquad (r \in m.\texttt{rm}.\texttt{activeNames} \Rightarrow \{\texttt{caAtRN}_{\texttt{R}}(r, m.\texttt{rm}) = \texttt{caAtRN}_{\texttt{R}}(r, \delta(m).\texttt{rm})\}) \\
&\qquad\quad \wedge \{\texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta)
\end{aligned}$$

We introduce another abbreviation to make this a bit more readable.

Definition 5.1.2 (rootNamesAgree) *The predicate*

$$\texttt{rootNamesAgree} : \mathbb{S}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\texttt{rootNamesAgree}(m_1, m_2)$$
$$= \forall r : N_r \cdot r \in m_1.\texttt{rm.activeNames} \Rightarrow \{\texttt{caAtRN}_\texttt{R}(r, m_1.\texttt{rm}) = \texttt{caAtRN}_\texttt{R}(r, m_2.\texttt{rm})\}.$$

This gives us

Formalization 5.1.2

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\quad (\forall m : \mathbb{S}_m \cdot \\
&\qquad\quad \texttt{rootNamesAgree}(m, \delta(m)) \\
&\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_\texttt{m}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta).
\end{aligned}$$

Suppose we pause-and-change the machine twice in a row, without any intervening actions. Since the pauses have no effect of their own, this is the same as pausing and making two changes. Since our intuition only allows a change to affect unreachable memory, it is natural that it makes the same restriction on a pair of changes. On the other hand, Formalization 5.1.2 permits the first change to reduce the set of active root names, thereby enlarging the set of unreachable memory. After that, the second change can alter parts of memory that were originally unreachable. The next version ensures that the active root names remain active.

Formalization 5.1.3

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\quad (\forall m : \mathbb{S}_m \cdot \\
&\qquad\quad m.\texttt{rm.activeNames} \subseteq \delta(m).\texttt{rm.activeNames} \\
&\qquad \wedge \texttt{rootNamesAgree}(m, \delta(m)) \\
&\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_\texttt{m}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta)
\end{aligned}$$

There is another problem. $\delta(m)$ may have too few cells left to allocate. As a counterexample, $\delta(m_0)$ could have no unused cells and still satisfy Formalization 5.1.3. We must be able to make as many allocations in $\delta(m)$ as we could in m. We strengthen the antecedent of Formalization 5.1.3, obtaining

Formalization 5.1.4

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\quad (\forall m : \mathbb{S}_m \cdot \\
&\qquad\quad m.\texttt{rm.activeNames} \subseteq \delta(m).\texttt{rm.activeNames} \\
&\qquad \wedge |m.\texttt{unused}| \leq |\delta(m).\texttt{unused}| \\
&\qquad \wedge \texttt{rootNamesAgree}(m, \delta(m)) \\
&\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta).
\end{aligned}$$

We still have problems. Our machine is set up to ensure that, if m is achievable, and a is an address, then

$$a \in m.\texttt{unused} \Rightarrow \neg\texttt{reachable}_{\texttt{m}}(m, a),$$

but Formalization 5.1.4 allows us to have

$$a \in \delta(m).\texttt{unused} \wedge \texttt{reachable}_{\texttt{m}}(\delta(m), a).$$

Consider a counterexample in which $\delta(m_0).\texttt{unused} = A_c$ and $\texttt{choice}(A_c) = a_{default}$. Here is the computation for this counterexample:

$\texttt{addRN}(r_1)$	r_1 points to $a_{default}$.
$\texttt{setData}(3, r_1)$	Put 3 in $a_{default}$.
$\texttt{addRN}(r_2)$	Prepare another root-name.
$\texttt{new}(r_2)$	Get a cell to hold the value 4.
$\texttt{setData}(4, r_2)$	Put 4 in the cell.
$\texttt{getData}(r_1)$	Get the data from $a_{default}$.

Assuming $|A_c| \geq 2$, and there is no pause-and-change during this computation, the final value is 3. With the problematic δ and `choice` functions, if we pause-and-change just before calling `new`, the final value is 4. Since this is consistent with its antecedent, Formalization 5.1.4 must be false. We weaken it even further, obtaining

Formalization 5.1.5

$$
\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\qquad (\forall m : \mathbb{S}_m \cdot \\
&\qquad\qquad m.\texttt{rm}.\texttt{activeNames} \subseteq \delta(m).\texttt{rm}.\texttt{activeNames} \\
&\qquad\qquad \wedge |m.\texttt{unused}| \leq |\delta(m).\texttt{unused}| \\
&\qquad\qquad \wedge \{\forall a : A_c \cdot a \in \delta(m).\texttt{unused} \Rightarrow \neg\texttt{reachable}_{\texttt{m}}(\delta(m), a)\} \\
&\qquad\qquad \wedge \texttt{rootNamesAgree}(m, \delta(m)) \\
&\qquad\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, \delta(m))\}) \\
&\quad \Rightarrow \texttt{pcSafe}(\delta).
\end{aligned}
$$

If m and $\delta(m)$ satisfy the antecedent of Formalization 5.1.5, we say that $\delta(m)$ contains the reachable part of m. It is convenient to define a corresponding predicate.

Definition 5.1.3 (`reachablyContained`) *The predicate*

$$\texttt{reachablyContained} : \mathbb{S}_m \to \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$
\begin{aligned}
&\texttt{reachablyContained}(m, m') \\
&\qquad = \quad m.\texttt{rm}.\texttt{activeNames} \subseteq m'.\texttt{rm}.\texttt{activeNames} \\
&\qquad\qquad \wedge |m.\texttt{unused}| \leq |m'.\texttt{unused}| \\
&\qquad\qquad \wedge \{\forall a : A_c \cdot a \in m'.\texttt{unused} \Rightarrow \neg\texttt{reachable}_{\texttt{m}}(m', a)\} \\
&\qquad\qquad \wedge \texttt{rootNamesAgree}(m, m') \\
&\qquad\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, m')\}.
\end{aligned}
$$

Thus, our formalization now reads

Formalization 5.1.6

$$\forall \delta : \mathbb{S}_m \rightarrow \mathbb{S}_m \cdot \{\forall m : \mathbb{S}_m \cdot \texttt{reachablyContained}(m, \delta(m))\} \Rightarrow \texttt{pcSafe}(\delta).$$

We hope the reader agrees that `reachablyContained` captures the correct notion of *"the replacement memory state matches the parts of the original memory that were reachable via some chain of pointers."* As we shall see in the next section, the consequent in Formalization 5.1.6 is still too strong, so that Formalization 5.1.6 does not hold. In Section 5.3 we discuss a generalization of `reachablyContained`, and in Chapter 6 we employ a sub-relation of it.

5.2 Which Computations Count?

Intuition 1 starts with the phrase *"The result of the computation ..."*, which, at first glance, suggests that there is one specific computation under discussion. On a more careful reading, we see that, since the computation is not specified, this is an informal Skolemization of a universal quantification. So far, we have given an example of a machine ($\mathbb{M}_m$) to which Intuition 1 applies, and we have defined the set of all computations in that machine (${\mathbb{A}_m}^* \times \mathbb{V}_m$). But, if we interpret Intuition 1 as a quantification over ${\mathbb{A}_m}^* \times \mathbb{V}_m$, we discover that it does not hold.

Consider the computation with no actions, just the viewpoint $\texttt{getData}(r)$. The result of this computation is

$\texttt{compute}(m_0, [\,], \texttt{getData}(r))$	Definition of `compute`.
$= \{\texttt{getData}(r)\}\texttt{do}(m_0, [\,])$	Definition of `do`.
$= \{\texttt{getData}(r)\}(m_0)$	Definition of `getData`.
$= \texttt{getData}_{\vec{\texttt{C}}}(\texttt{caAtRN}_{\texttt{R}}(r, m_0.\texttt{rm}), m_0.\texttt{rm.cm}),$	

which is just the data stored in the cell that r leads to in the initial state. The construction of $\mathbb{M}_m$ does not tell us which cell is associated with r, so we must assume the worst. If, in m_0, r does not lead to $a_{default}$, then $\texttt{caAtRN}_{\texttt{R}}(r, m_0.\texttt{rm})$ is unreachable, and Intuition 1 asserts that we can change the data stored there without affecting the result of this computation. To avoid this contradiction, we say that this computation does not count. (Notice that this contradiction only occurs if there are at least two cells and two data values.)

Because our definition of the initial state of $\mathbb{M}_m$ provides no information about the contents of any of the data fields, a computation with no actions is like using a variable before it is initialized. Before we can know anything about the value returned by a viewpoint, we must ensure that the associated handle refers to a cell that has been given data by means of a `setData` action. Use before initialization is such a common error that we expect a language implementation to warn us whenever there is a chance for it to occur. Thus, we don't care if δ alters the results of action-less computations, and we see no loss in barring them from the domain of Intuition 1.

In this section, we show that there are many more computations that do not count. By the end of the section, we produce a predicate,

$$\texttt{computationCounts} : {\mathbb{A}_m}^* \times \mathbb{V}_m \to \mathbb{B},$$

which tells us which computations of $\mathbb{M}_m$ count. Our definition of `computationCounts` is the last in a sequence of approximations, the first of which is

$$\texttt{computationCounts}_0(l, v) = \top.$$

For each step in this progression, we

1. describe a problematic computation allowed by the best approximation so far,
2. show how to alter its result via a pause and a change to reachable memory, and
3. describe a stronger approximation that disallows the problematic computation.

Suppose we have a computation ending in the viewpoint $\texttt{getData}(r)$, with r leading to an unreachable cell in the final state. Then, we can pause the machine after all of the actions have been performed, and change the data in that cell, thus changing the result. To avoid this, we say that a viewpoint only counts if the corresponding root name is active. Before we produce the matching approximation to `computationCounts`, we introduce a couple of abbreviations.

Definition 5.2.1 (Active Root Names in $\mathbb{M}_m$) *The predicate*

$$\mathtt{active} : N_r \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\mathtt{active}(r, m) = r \in m.\mathtt{rm}.\mathtt{activeNames}.$$

Definition 5.2.2 (The Viewpoints that Count in $\mathbb{M}_m$) *The predicate*

$$\mathtt{vpCounts} : \mathbb{V}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\mathtt{vpCounts}(v, m) = \exists r : N_r \cdot \{v = \mathtt{getData}(r)\} \wedge \mathtt{active}(r, m).$$

Here is an approximation to `computationCounts` that eliminates our first set of problematic computations:

$$\mathtt{computationCounts}_1(l, v) = \mathtt{vpCounts}(v, \mathtt{do}(m_0, l))$$

When r is active, then, by definition, the cell r leads to is reachable. Thus, this disallows computations that end by following root names leading to unreachable cells. (The observant reader may object that this restriction is too strong, as there can be inactive root names that lead to reachable cells. In what follows we discuss problems caused by using `setData` with root names that are not active. A weaker restriction would allow us to use the mutability of $m.\mathtt{rm}.\mathtt{rnMap}$ to construct similar problems using `getData`.)

Consider this computation:

m_0	$\texttt{addRN}(r_{\mathit{affected}})$	A new root name.
m_1	$\texttt{followPFN}(r_{\mathit{affected}}, n, r)$	Update the new root name.
m_2	$\texttt{getData}(r_{\mathit{affected}})$	Get the data from the new root name.

(The left column gives a name to the state, while the middle column specifies the action to perform in that state. Thus, $m_1 = \texttt{do}(m_0, [\texttt{addRN}(r_{\mathit{affected}})])$.)

In m_2, r_{affected} leads to the same cell that was found in m_1 by following r to a cell, and then following n from that cell to another (*i.e.*, the lower right cell of Figure 4.2, $\texttt{caAtRN}_{\texttt{R}}(r_{\mathit{affected}}, m_1.\texttt{rm})$). The result of this computation is the data in that cell. Now suppose that cell is not reachable in m_1. Then the result can be altered by pausing the machine in state m_1 and changing the data in the cell.

Before eliminating this type of computation, we introduce two more abbreviations. At this point, we can only give approximations; proper definitions appear with the final definition of `computationCounts`. The predicate

$$\texttt{actionCounts} : \mathbb{A}_m \times \mathbb{S}_m \to \mathbb{B},$$

describes the actions that count in each state. Our first approximation is

$$\begin{aligned}&\texttt{actionCounts}_0(\alpha, m)\\ &\quad = \forall r_{\mathit{affected}}, r : N_r, n : N_{pf} \cdot \{\alpha = \texttt{followPFN}(r_{\mathit{affected}}, n, r)\} \Rightarrow \texttt{active}(r, m).\end{aligned}$$

This states that if the action α is a `followPFN` action, then its second root name argument must be active in m. The predicate

$$\texttt{alCounts} : {\mathbb{A}_m}^* \times \mathbb{S}_m \to \mathbb{B},$$

lets us say when an entire list of actions counts. Our first approximation is

$$\begin{aligned}&\texttt{alCounts}_0(l, m)\\ &\quad = \forall l_1, l_2 : {\mathbb{A}_m}^*, \alpha : \mathbb{A}_m \cdot \{l = \texttt{append}(l_1, \alpha :: l_2)\} \Rightarrow \texttt{actionCounts}_0(\alpha, \texttt{do}(m, l_1)).\end{aligned}$$

A non-empty action list can be written as $\texttt{append}(l_1, \alpha :: l_2)$, where l_1 and l_2 are action lists, and α some action in the list. Thus, we require that each action count in the state where it is performed (when l is performed in m, α is performed in the state $\texttt{do}(m, l_1)$).

Now, we eliminate the latest set of problematic computations by moving to

$$\texttt{computationCounts}_2(l, v) = \texttt{vpCounts}(v, \texttt{do}(m_0, l)) \wedge \texttt{alCounts}_0(l, m_0).$$

Here is another problematic computation:

m_0	$\texttt{addRN}(r_{\mathit{affected}})$	A new root name.
	$\texttt{new}(r_{\mathit{affected}})$	Allocate a cell.
m_1	$\texttt{setPF}(r, n, r_{\mathit{affected}})$	Set a pointer field in the new cell.
	$\texttt{followPFN}(r_{\mathit{affected}}, n, r_{\mathit{affected}})$	Now r_{affected} and r point to the same cell.
	$\texttt{getData}(r_{\mathit{affected}})$	Get the data the two root names lead to.

At the end of this computation, we get the data r leads to in m_1. Suppose the cell r leads to is not reachable in m_1. Then, we can alter the result of the computation by pausing $\mathbb{M}_m$ in m_1, and altering the data in that cell. Henceforth, we implicitly give approximations to `computationCounts` by explicitly giving approximations to `actionCounts`. Thus, we prohibit computations of this kind by means of the following:

$$\begin{aligned}
&\texttt{actionCounts}_1(\alpha, m) \\
&\quad = \forall r_{\mathit{affected}}, r : N_r, n : N_{pf}\cdot \\
&\qquad\quad (\{\alpha = \texttt{followPFN}(r_{\mathit{affected}}, n, r)\} \Rightarrow \texttt{active}(r, m)) \\
&\qquad \wedge(\{\alpha = \texttt{setPF}(r, n, r_{\mathit{affected}})\} \Rightarrow \texttt{active}(r, m)).
\end{aligned}$$

A `setPF` action only counts in states where its first root name argument is active.

The next computation leads to problems in a less direct manner.

m_0	$\texttt{addRN}(r_1)$	A new root name.
	$\texttt{new}(r_1)$	Allocate a cell.
	$\texttt{setData}(d_1, r_1)$	Set the data of the new cell.
m_1	$\texttt{setData}(d_2, r)$	Set some data via some other root name.
	$\texttt{getData}(r_1)$	Get the data the first root name leads to.

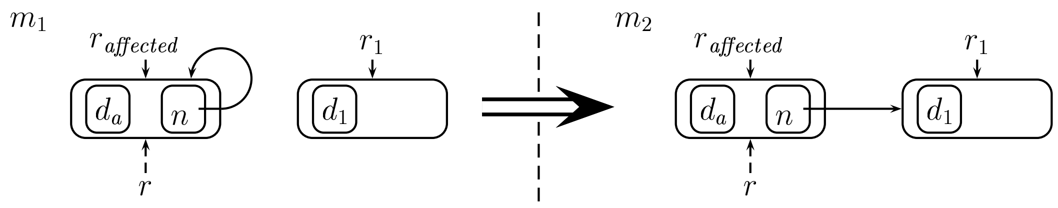

Figure 5.1: `setPF` transforms the left into the right: $m_2 = \{\texttt{setPF}(r_1, n, r)\}(m_1)$.

The problem arises when r is different from r_1, but, in m_1, it leads to the same cell that $\texttt{new}(r_1)$ attached to r_1. This can only happen if r is not active in m_1. If it does happen, the result of the computation is d_2. If we pause the machine in m_1, and change $m_1.\texttt{rm}.\texttt{rnMap}$ so that r leads somewhere else, the result of the computation becomes d_1.

It is reasonable to object that, according to our intuition, $m.\texttt{rm}.\texttt{rnMap}$ is not part of memory, so its value at r is not within the unreachable part of memory. On the other hand, practical implementations of garbage collection do manipulate this association, and our definitions do make $m.\texttt{rm}.\texttt{rnMap}$ part of the memory state that δ is allowed to alter. If we do not forbid this computation (and others like it) then Intuition 1 does not hold in our formalization.

We forbid the problematic computations involving `setData` with

$$
\begin{aligned}
&\texttt{actionCounts}_2(\alpha, m)\\
&\quad = \forall r_{affected}, r : N_r, n : N_{pf}, d : D \cdot\\
&\qquad\qquad (\{\alpha = \texttt{followPFN}(r_{affected}, n, r)\} \Rightarrow \texttt{active}(r, m))\\
&\qquad\qquad \wedge(\{\alpha = \texttt{setPF}(r, n, r_{affected})\} \Rightarrow \texttt{active}(r, m))\\
&\qquad\qquad \wedge(\{\alpha = \texttt{setData}(d, r)\} \Rightarrow \texttt{active}(r, m)).
\end{aligned}
$$

This time the new clause states that `setData` can only be applied to root names that are active.

If the last argument of `setPF` is not in use, we can have the same kind of problems that `setData` produced. Consider

m_0	$\texttt{addRN}(r_{\mathit{affected}})$	A new root name.
	$\texttt{new}(r_{\mathit{affected}})$	Allocate a cell.
	$\texttt{setData}(d_a, r_{\mathit{affected}})$	Set the data of the new cell.
	$\texttt{setPF}(r_{\mathit{affected}}, n, r_{\mathit{affected}})$	Make it point to itself.
	$\texttt{addRN}(r_1)$	Another new root name.
	$\texttt{new}(r_1)$	Another new cell.
	$\texttt{setData}(d_1, r_1)$	This cell has different data.
m_1	$\texttt{setPF}(r_1, n, r)$	Point to the same cell as r_1.
m_2	$\texttt{followPFN}(r_{\mathit{affected}}, n, r_{\mathit{affected}})$	Where does r_{affected}'s pointer lead?
	$\texttt{getData}(r_{\mathit{affected}})$	Either d_a or d_1.

In this computation, r has not been added, so it is inactive, or not in use. Because our construction of m_0 does not say anything about $m_0.\texttt{rm}.\texttt{rnMap}$, is is possible that, in m_1, r and r_{affected} lead to the same cell. The left side of Figure 5.1 shows this possible m_1. Under these conditions, performing $\texttt{setPF}(r_1, n, r)$ changes m_1 into m_2, the right side of Figure 5.1. Then the result of this computation is d_1. We can pause this computation in m_1, change the machine so that r leads to anything else, and the result becomes d_a. Thus, we introduce

$$
\begin{aligned}
&\texttt{actionCounts}_3(\alpha, m)\\
&\quad = \forall r_{\mathit{affected}}, r : N_r, n : N_{pf}, d : D \cdot\\
&\qquad\qquad (\{\alpha = \texttt{followPFN}(r_{\mathit{affected}}, n, r)\} \Rightarrow \texttt{active}(r, m))\\
&\qquad\quad \wedge(\{\alpha = \texttt{setPF}(r, n, r_{\mathit{affected}})\} \Rightarrow \texttt{active}(r, m) \wedge \texttt{active}(r_{\mathit{affected}}, m))\\
&\qquad\quad \wedge(\{\alpha = \texttt{setData}(d, r)\} \Rightarrow \texttt{active}(r, m)).
\end{aligned}
$$

The new consequence of the `setPF` clause ensures that we cannot use an inactive root name to indicate which cell to update.

There is one last group of computations to ban. What happens if all of the cells have been allocated? Dealing with that, of course, is the whole point of garbage collection.

However, we have not included any GC in $\mathbb{M}_m$ because we are formalizing the intuition that justifies adding GC. Furthermore, we *want* GC to alter the result in this case, so that, for some computations, the given machine can simulate another machine with more memory. The definition of $\mathbb{M}_m$ only states that, when $m.\texttt{unused}$ is not empty, $\texttt{new}(r)$ removes an address from $m.\texttt{unused}$ and associates it with r. If $m.\texttt{unused}$ is empty, then it is possible for $\texttt{new}(r)$ to associate r with a cell that is already reachable. This causes the same kind of problems that $\texttt{followPFN}(r_{\mathit{affected}}, n, r)$ causes when r is not active. Now we give our final definitions for this section.

Definition 5.2.3 (`actionCounts`, `alCounts`, and `computationCounts`) *The predicate*

$$\texttt{actionCounts} : {\mathbb{A}_m}^* \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\begin{aligned}
&\texttt{actionCounts}(\alpha, m)\\
&\quad = \forall r_{\mathit{affected}}, r : N_r, n : N_{pf}, d : D\cdot\\
&\qquad\qquad (\{\alpha = \texttt{followPFN}(r_{\mathit{affected}}, n, r)\} \Rightarrow \texttt{active}(r, m))\\
&\qquad\qquad \wedge(\{\alpha = \texttt{setPF}(r, n, r_{\mathit{affected}})\} \Rightarrow \texttt{active}(r, m) \wedge \texttt{active}(r_{\mathit{affected}}, m))\\
&\qquad\qquad \wedge(\{\alpha = \texttt{setData}(d, r)\} \Rightarrow \texttt{active}(r, m))\\
&\qquad\qquad \wedge(\{\alpha = \texttt{new}(r)\} \Rightarrow \{m.\texttt{unused} \neq \emptyset\}).
\end{aligned}$$

The predicate

$$\texttt{alCounts} : {\mathbb{A}_m}^* \times \mathbb{S}_m \to \mathbb{B},$$

is defined by

$$\begin{aligned}
&\texttt{alCounts}(l, m)\\
&\quad = \forall l_1, l_2 : {\mathbb{A}_m}^*, \alpha : \mathbb{A}_m \cdot \{l = \texttt{append}(l_1, \alpha :: l_2)\} \Rightarrow \texttt{actionCounts}(\alpha, \texttt{do}(m, l_1)).
\end{aligned}$$

The predicate

$$\texttt{computationCounts} : {\mathbb{A}_m}^* \times \mathbb{V}_m \to \mathbb{B}$$

is defined by

$$\texttt{computationCounts}(l, v) = \texttt{vpCounts}(v, \texttt{do}(m_0, l)) \wedge \texttt{alCounts}(l, m_0).$$

Here is an English approximation to this last definition: *A computation counts if it only uses reasonable root names and does not use too much memory.* By *reasonable root names*, we mean that the root names must be active in the state where they are used - this is more than we require formally, but it is only an English approximation. By *does not use too much memory*, we mean that whenever `new` is called, there is a cell available for allocation.

5.3 Toward a Formal Characterization of Garbage Collection

Now that we have formalized all the parts of Intuition 1, we can state it formally as

Formalization 5.3.1

$$\begin{aligned} &\forall l_1, l_2 : \mathbb{A}_m{}^*, v : \mathbb{V}_m, m : \mathbb{S}_m. \\ &\quad \texttt{computationCounts}(\texttt{append}(l_1, l_2), v) \wedge \texttt{reachablyContained}(\texttt{do}(m_0, l_1), m) \\ &\qquad \Rightarrow \texttt{preservesResult}(m_0, l_1, m, l_2, v). \end{aligned}$$

If $\langle \texttt{append}(l_1, l_2), v \rangle$ is a computation that counts, we can pause it after performing the actions in l_1. This puts the machine in state $\texttt{do}(m_0, l_1)$. We can then replace that state with any that reachably contains it (*e.g.*, m), and the result will be preserved.

We defined `pcSafe` so that the δ function had to preserve the results of all computations. As we have seen in the previous section, this is too much to expect for a garbage collector. Let us make a small generalization of `pcSafe`, so that we can have a cleaner statement of our intuition. We parameterize it by a predicate that classifies computations, so we can limit the domain on which the δ function must be safe.

Definition 5.3.1 (`limitedPcSafe`) *The predicate*

$$\texttt{limitedPcSafe} : (\mathbb{A}^* \times \mathbb{V} \to \mathbb{B}) \times (\mathbb{S} \to \mathbb{S}) \to \mathbb{B}$$

is defined by

$$\begin{aligned}&\texttt{limitedPcSafe}(\rho, \delta)\\ &\quad = \forall l_1, l_2 : \mathbb{A}^*, v : \mathbb{V}\cdot\\ &\qquad \rho(\texttt{append}(l_1, l_2), v) \Rightarrow \texttt{preservesResult}(s_0, l_1, \delta(\texttt{do}(s_0, l_1)), l_2, v).\end{aligned}$$

For $\mathbb{M}_m$, we can now say that our intuition corresponds to

$$\begin{aligned}&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m\cdot\\ &\quad \{\forall m : \mathbb{S}_m \cdot \texttt{reachablyContained}(m, \delta(m))\}\\ &\qquad \Rightarrow \texttt{limitedPcSafe}(\texttt{computationCounts}, \delta).\end{aligned}$$

Our definitions allow us to formalize a more interesting intuition.

> *If we pause the machine, and replace the state with one in which the old state can be isomorphically embedded, the result will not be affected.*

Intuitively, if m is isomorphically embedded in m', then

1. the active root names of m are active in m',
2. the new unused set is not smaller,
3. none of the unused addresses are reachable,
4. distinct reachable addresses in m correspond to distinct addresses in m',
5. the structure of the root name map is preserved for active root names,
6. the structure of the connections between reachable cell addresses is preserved, and
7. the structure of the data in the reachable cells is preserved.

Thus, this form of embeddability is a weakening of `reachablyContained` that involves a directed graph embedding of the reachable part of m into m'. Items 1, 2 and 3 already appear in the definition of `reachablyContained`. Item 4 disallows degeneracy of the embedding. Item 5 corresponds to the "rootNamesAgree" clause of `reachablyContained`, while items 6 and 7 correspond to the last clause. Before we give the definition for isomorphic embeddability, we introduce abbreviations to handle items 4, 5, 6 and 7.

Definition 5.3.2 (Isomorphic Embeddability and Related Abbreviations) *The predicate*

$$\texttt{nonDegenerate} : (A_c \to A_c) \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\begin{aligned}&\texttt{nonDegenerate}(\Phi, m)\\ &\quad = \forall a, a' : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \wedge \texttt{reachable}_{\texttt{m}}(m, a') \wedge (a \neq a') \Rightarrow \{\Phi(a) \neq \Phi(a')\}.\end{aligned}$$

The predicate

$$\texttt{preservesRootMap} : (A_c \to A_c) \times \mathbb{S}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\begin{aligned}&\texttt{preservesRootMap}(\Phi, m, m')\\ &\quad = \forall r : N_r \cdot \texttt{active}(m, r) \Rightarrow \{\Phi(\texttt{caAtRN}_{\texttt{R}}(r, m.\texttt{rm})) = \texttt{caAtRN}_{\texttt{R}}(r, m'.\texttt{rm})\}.\end{aligned}$$

The predicate

$$\texttt{preservesContents} : (A_c \to A_c) \times \mathbb{S}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\begin{aligned}&\texttt{preservesContents}(\Phi, m, m')\\ &\quad = \forall a : A_c, n : N_{pf} \cdot\\ &\qquad \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \quad \{\texttt{getPF}_{\vec{\texttt{C}}}(n, \Phi(a), m'.\texttt{rm.cm}) = \Phi(\texttt{getPF}_{\vec{\texttt{C}}}(n, a, m.\texttt{rm.cm}))\}\\ &\qquad\qquad \wedge\{\texttt{getData}_{\vec{\texttt{C}}}(\Phi(a), m'.\texttt{rm.cm}) = \texttt{getData}_{\vec{\texttt{C}}}(a, m.\texttt{rm.cm})\}.\end{aligned}$$

The predicate

$$\mathtt{embeddable} : \mathbb{S}_m \times \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$\begin{aligned}
&\mathtt{embeddable}(m, m') \\
&\quad = \quad m.\mathtt{rm}.\mathtt{activeNames} \subseteq m'.\mathtt{rm}.\mathtt{activeNames} \\
&\qquad \wedge |m.\mathtt{unused}| \leq |m'.\mathtt{unused}| \\
&\qquad \wedge \{\forall a : A_c \cdot a \in m'.\mathtt{unused} \Rightarrow \neg \mathtt{reachable}_{\mathtt{m}}(m', a)\} \\
&\qquad \wedge \exists \Phi : A_c \to A_c \cdot \quad \mathtt{nonDegenerate}(\Phi, m) \\
&\qquad\qquad\qquad\qquad \wedge \mathtt{preservesRootMap}(\Phi, m, m') \\
&\qquad\qquad\qquad\qquad \wedge \mathtt{preservesContents}(\Phi, m, m').
\end{aligned}$$

It may help to compare the latest definition with Definition 5.1.3. If we require that Φ be the identity function, then $\mathtt{nonDegenerate}(\Phi, m)$ is trivially true, and embeddability degenerates into containment. Thus, we see that reachable containment implies reachable embeddability.

With this in hand, we can express our intuition about embeddability using the following formal expression:

Formalization 5.3.2

$$\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\qquad \{\forall m : \mathbb{S}_m \cdot \mathtt{embeddable}(m, \delta(m))\} \Rightarrow \mathtt{limitedPcSafe}(\mathtt{computationCounts}, \delta).
\end{aligned}$$

This states that it is safe to use a garbage collector, δ, which yields an output state into which the reachable parts of the input state can be isomorphically embedded.

Chapter 6

Encoding Garbage Collection in HOL

We have encoded our formalism in the language of the HOL proof assistant [10], and used it to prove the lemmas and theorems presented in chapters 3, 4, and 5. Most of the assertions that appear embedded in the text were also encoded and proven using HOL. A simple manual editing process was used to convert them into the form presented here.

HOL is an open-source theorem proving tool written in ML. It can be obtained from SourceForge [10]. Definitions and theorems in HOL are expressed in a variant of the typed lambda calculas.

The HOL encoding for Lemma 3.3.1 looks like this:

```
!avm:^AVM chg.
    pcSafe avm chg
            = (!s. achievable avm s ==> computationEq avm s (chg s))
```

Notice that HOL uses `!` for $\forall$, and `==>` for $\Rightarrow$. The left hand of the equality represents the application of `pcSafe` to two arguments, `avm` and `chg`. In this encoding, `^AVM` represents the type of all action-view machines. All of our definitions in HOL explicitly refer to this type.

Here is the proof of Lemma 3.3.1 in HOL:

```
RW_TAC std_ss [pcSafe,                          Definition 3.3.2
               achievable,                      Definition 3.3.3
               computationEq,                   Definition 3.3.4
               preservesResult,                 Definition 3.3.1
               computeOverAppend,               Lemma 3.2.1
               specifiesComputation,
               isActionListOverAppend_lemma]
THEN PROVE_TAC []
```

The right hand column of this proof refers to definitions and a lemma as they appear in this thesis. `RW_TAC` uses `std_ss` and the definitions and lemmas in the list to simplify Lemma 3.3.1. In HOL, the definition of `pcSafe` uses `specifiesComputation` to explicitly disallow actions and viewpoints that are not part of the AVM; in our less formal presentation, these were implicitly disallowed. `isActionListOverAppend_lemma` is a simple lemma in HOL that states that if we append two lists of actions from $\mathbb{A}^*$, the new list is also in $\mathbb{A}^*$. After `RW_TAC` simplifies Lemma 3.3.1, `PROVE_TAC` searches for, and finds, a proof. (HOL does not tell us the simplification steps used, or the details of the proof found.)

We have not proven either of the formalizations of Section 5.3, but we have proven that a weaker formalization of Intuition 1 holds. Instead of the requirement that the old state be reachably embeddable in the replacement state, or the stronger requirement that it be reachably contained, we proved a statement using the overly strong requirement that the old state be reachably equal to the new state. The following definition of reachable equivalence corresponds directly to the definition used in HOL.

Definition 6.1 (`reachablyEq`) *The predicate*

$$\texttt{reachablyEq} : \mathbb{S}_m \to \mathbb{S}_m \to \mathbb{B}$$

is defined by

$$
\begin{aligned}
&\texttt{reachablyEq}(m, m') \\
&\quad = \{m.\texttt{rm.activeNames} = m'.\texttt{rm.activeNames}\} \\
&\qquad \wedge \texttt{rootNamesAgree}(m, m') \\
&\qquad \wedge \{m.\texttt{unused} = m'.\texttt{unused}\} \\
&\qquad \wedge \{\forall a : A_c \cdot \texttt{reachable}_{\texttt{m}}(m, a) \Rightarrow \texttt{sameContents}(a, m, m')\}.
\end{aligned}
$$

`reachablyEq` is a stronger property than `reachablyContained`. First, it requires that the set of root names be identical (a small and not unreasonable restriction), while `reachablyContained` allows the set to grow. Second, the set of unused cells must be exactly the same in m and m'. The following lemma gives two ways `reachablyEq` could be defined in terms of `reachablyContained`.

Lemma 6.0.1 *If m and m' are states of $\mathbb{M}_m$, then*

$$
\begin{aligned}
&\texttt{reachablyEq}(m, m') \\
\Leftrightarrow\ &\texttt{reachablyContained}(m, m') \\
&\quad \wedge \{m.\texttt{rm.activeNames} = m'.\texttt{rm.activeNames}\} \wedge \{m.\texttt{unused} = m'.\texttt{unused}\} \\
\Leftrightarrow\ &\texttt{reachablyContained}(m, m') \wedge \texttt{reachablyContained}(m', m).
\end{aligned}
$$

Insisting that the set of unused cells stays the same is far too strong; it disallows garbage collection. On the other hand, it makes the proof in HOL much simpler. Indeed, it was essential to obtaining a fully checked proof of the following weak formalization of Intuition 1:

Formalization 6.1

$$
\begin{aligned}
&\forall \delta : \mathbb{S}_m \to \mathbb{S}_m \cdot \\
&\quad \{\forall m : \mathbb{S}_m \cdot \texttt{reachablyEq}(m, \delta(m))\} \Rightarrow \texttt{limitedPcSafe}(\texttt{computationCounts}, \delta).
\end{aligned}
$$

Chapter 7

Conclusion

7.1 Conclusions

We presented a full formalization of Intuition 1, which is fundamental to the notion of garbage collection. To do so, we built up a large collection of formal definitions for basic terms and phrases that occur frequently in discussions of GC. As shown in Section 5.3, these definitions provide some of the tools needed to analyze much more difficult aspects of GC. They also provide the foundation for a suite of tools for analyzing GC with the HOL proof assistant.

Garbage collection is a service provided by many programming languages. As such, the requirements that GC must meet need to be expressed in terms that make sense from the language implementor's point of view as a user of managed memory. We have taken the first step in that direction by basing our formalization on computations - *"reachability of cells"* is not a concept that appears in the language implementor's problem space, *"the value obtained at the end of a computation"* is.

Traditional formal methods have difficulty with side effects that change the referents of names (*e.g.*, aliasing, call-by-reference, and pointers) [16]. GC is, of course, replete with behaviors of this type. These problems can make it hard to treat even the simplest of pointer-related tasks.

For example, consider the chore of writing a routine that removes an entry from a linked list in the programming language C. (In spite of the basic nature of this, it is one of *the* tests used when hiring C programmers - apparently, many can't get it right.) In addition to the standard requirements concerning the contents of the list, it is usually necessary to make the memory that represented the entry available for future use (to "free" it). It is currently quite difficult to give a formal statement of either this requirement or of the conditions under which such a routine may be safely used.

This thesis is part of the author's long term efforts to make it easier to reason about algorithms that rely on techniques like call-by-reference. We feel that it is essential to provide a formalism that makes it straightforward to talk about states. In the case of garbage collection, our intuition is state-oriented. Because our formalism is equally state-oriented, we are more confident that it correctly reflects our intuition.

7.2 Limitations

A full formal characterization of garbage collection is the same as a specification for it. What we presented is, at best, a specification by implementation of one managed memory machine, and some requirements that a garbage collector for this machine must meet. Our approach is heavily dependent on our model of memory. This obscures the essential aspects of the problem. An axiomatic or algebraic approach would be better, so long as it is easy to see that the formalization corresponds to the problem.

It is essential that a garbage collector be safe no matter how many times it is run. Our formalization only states the conditions under which a single pause-and-change can be performed. Our intuition led us to predicates involving entire computations, so that our results only apply to achievable states, and we don't know that the state following the pause-and-change is achievable. To rectify this, we must either change the definition of `achievable`, or the definition of `pcSafe`, or come up with new predicates.

Of the three formalizations in Section 5.3, we were only able to prove the last and most restrictive. As noted previously, if the new state is `reachablyEq` to the old one, then none of the garbage has been collected.

7.3 Future Work

Our experiences while preparing this thesis have suggested more areas for future work than we have room to describe here. We will not repeat the limitations from the previous section. What follows are just some of the more interesting possibilities.

7.3.1 Extensions to Our Work

We would like to be able to state the requirement that garbage collection meets. A first, informal statement might be *"to make a finite machine simulate an infinite machine"*. We can make this more precise by saying that, if N is the maximum number of cells reachable at any point when the computation is run on an infinite machine, and N is not more than the number of cells in the finite machine, then both machines should give the same result. What we have presented so far brings us very close to this goal. To do this, we must formalize observational equivalence in our setting. We have started work on it, but feel that it is outside the scope of this thesis.

In our work with HOL, we used an alternate way of specifying which computations count. It described actions using a state-domain predicate in addition to the state-transformation function presented here. Our `actionCounts` is effectively the conjunction of these predicates. It seems that a specifier would find it more natural, as it associates preconditions directly with actions. It does not appear here because it requires a great deal more infrastructure. Presenting it, and proving that it yields the same set of computations, would be good future work.

Real memory management systems have cells of varying sizes, and often keep regions of unused memory instead of a set of unused cells. Furthermore, fragmentation of available memory is a serious concern. We would like to extend our formalization to include models or representations with size functions. Some of the operations (*e.g.*, `new`, or `setPF` when it is a new link from the cell) would cause the size of the representation to increase. This should allow us to specify the requirement that the collector decrease the size of the representation. It should also support requiring that the collector ensure large contiguous regions can be allocated.

It may prove profitable to view a program as a function from machine states to sequences of actions. Formalizing this would link programs to computations. We would then use this to define the standard program fragments: while, if, recursion, etc. In particular, this would support assignment and other side-effects. Future work on this requires some careful but "intuitively obvious" (*i.e.*, should be easy, but might be impossible) work to define something that is like our version of computation, but allows a sequence of interleaved observations, rather than just one at the end.

7.3.2 Applications of Our Work

The most obvious application is to use this framework for proving specific GC algorithms correct. In doing so, we would have to provide a more realistic treatment of null and cells with various kinds of structure.

Can this framework be used in planning? It has a great deal in common with the situation calculus. Is it sufficient to allow planning techniques to be applied to finding programs? This is another of the author's long term goals, and it greatly influenced the formulation of the approach presented here.

Because we explicitly reason about sequences of states and actions, our definitions look suitable for formalizing the type of use-def reasoning central to many compiler optimizations. Although we avoided the issue of use-before-initialization, our framework should give us most of the tools needed to formalize the concept. We would like to provide terms that make it as easy as English to rigorously say things like “you must allocate memory before you use it” and “you must not use memory after you free it”.

We gave a general definition of when replacing one state with another preserves the result of the computation. Reasoning about algorithms in this way looks promising. We would like to look for other algorithms to which this approach can be applied.

Appendix A

HOL

A.1 Advice for Future Users of HOL

After the initial work was done (in HOL), and we knew what we were going to put in this thesis, there was still a great deal to do before the material was presentable. We found it most helpful to work with HOL in parallel with preparing this document. When a definition needed to be decomposed for the reader, we did so in HOL first. Often, when we needed intermediate results in the text, we first proved little theorems in HOL. Thus, HOL served as a proof-reader (*i.e.*, syntax and consistency checker) for the thesis as well as an assistant in finding and expressing proofs. Since we find it an eye-glazing, mind-numbing task to ensure that all the formulas are just right, we love to get any mechanical assistance of this sort, and we highly recommend it to others.

The HOL theorem proving tool has many good points. The author is particularly happy with the expressiveness of the specific higher-order logic used in HOL. The type inferencing is a big help in reducing clutter. Nonetheless, HOL is not easy to use, and it introduces a number of artificial barriers. In this section, we pass on some of the techniques we found helpful when working with HOL.

HOL provides a set of tools, called `bosslib`, to help search for proofs. The theorem you are trying to prove is placed on the *goalstack*, which you manipulate using a wide range of more basic proof tools. Watch out for the seductiveness of `bosslib`. Don't let the tool direct your search for a proof. It is better to have a good idea about how you will prove it first, then use HOL to fill in the details and check your work. Use LOTS of small definitions and lemmas.

HOL comes with a huge set of theories (67 of them), each of which provides some definitions, and a bunch of theorems about the definitions. To use a specific definition or theorem, you must know its name, but there are far too many names to remember, and the naming conventions change from theory to theory. You can reduce this load by using the simplifiers and provers provided by HOL. Our favorites are `RW_TAC`, `FULL_SIMP_TAC`, `PROVE_TAC` and `ARITH_PROVE`.

`RW_TAC` and `FULL_SIMP_TAC` take two arguments: a *simplification set*, and a list of theorems. The simplification set is mainly a set of rewrite rules, but may contain a lot more (*e.g.*, *congruences* and *conversions*, which we will not explain here.) In HOL, a rewrite rule corresponds to a theorem with an equational form (*e.g.*, $\texttt{append}(x, l) = x :: l$). It will make life much easier if you learn about simplification sets early, and augment the defaults. The theories provided with HOL come with simplification sets. We combined all the simplification sets for the HOL theories that we used, and threw in some other rewrite rules as well. Sometimes we had to fall back to the default simplification set because our set was rewriting too far, while the default stopped at a more useful point. The other argument to these two simplifiers is a list of theorems to be temporarily added to the simplification set.

`RW_TAC` is a much more powerful rewriter than `FULL_SIMP_TAC`, and sometimes proves simple theorems all by itself. Typically, it strips the theorem down into a set of assumptions and a goal. For example, if you place $\forall x \cdot P(x) \Rightarrow Q(x)$ on the goalstack, and then apply `RW_TAC`, `RW_TAC` does skolemization and makes $\{P(x)\}$ the set of assumptions and $Q(x)$ the goal. However, `RW_TAC` will not make further simplifications to the set of assumptions, for that you need FULL_SIMP_TAC.

PROVE_TAC does much more than simplification. For example, it can instantiate existentials in preparation for applying modus ponens. However, it does not use simplification sets, and it can only be applied as the last step in a proof.

ARITH_PROVE can let you avoid looking up simple theorems. You give ARITH_PROVE a HOL formula stating the theorem you want to use when applying a tool like RW_TAC. ARITH_PROVE proves it on the spot, and you have a nameless theorem for the other tool to use. For example, you may wish to use the fact that

$$\forall x, y, z \in \mathbb{N} \cdot x < y \wedge y < z \Rightarrow x < z.$$

Without ARITH_PROVE, you would have to find it in the huge set of theorems HOL provides (it is called arithmetic.LESS_TRANS). In some cases, the theorem is not provided by HOL, but ARITH_PROVE proves it easily (*e.g.*, transitivity of $>$).

HOL provides a mechanism for defining sophisticated datatypes (HOL_datatype), which we used extensively. Occasionally, reasoning with these can be quite painful. RW_TAC handles these very well; it is aware of all the datatypes currently defined, and most of the associated simplification rules. FULL_SIMP_TAC is completely unaware of them. When simplifying expressions involving datatype x in the set of assumptions, we had to call FULL_SIMP_TAC with a list of theorems obtained from the expression type_rws x.

Don't let HOL guess your types - type inferencing is good, type guessing is bad. If HOL says that it is introducing a type, change your definition. If you ever need to use INST_TYPE to instantiate a type, it will be easier if you can find the type in your definition - otherwise, you will have to decompose the HOL expression, which is annoying. Furthermore, your use of INST_TYPE will be easier to understand if it involves types will well-chosen names, rather than name generated by HOL. If HOL says there are multiple, overloaded choices, then most likely it guessed a type somewhere, which is why it is unable to find a unique match. We were able to eliminate all such problems by explicitly stating a small number of types - type inferencing took care of the rest. ty_antiq is very useful for this. It was the only way we could find in the `hol98` release of HOL to declare an abbreviation for a complex type expression.

There are multiple ways to tell HOL that a string such as xxx is an expression in the logic. Sometimes you use 'xxx', sometimes "xxx", and sometimes $--$ 'xxx' $--$. The last one is the best if you are using `ty_antiq`.

Read lots of code from the theories that come with HOL. The documentation isn't bad, but it is nowhere near good enough. You just won't learn some of the important techniques any other way. For example, that's how we found out about `ty_antiq` and `PRENEX_CONV`.

`PRENEX_CONV` can be used to rewrite the goalstack so that the goal is in prenex normal form (all quantifiers moved to the outside of the expression). Moving quantifiers out any other way is ridiculously painful. Fairly often, putting the goal in prenex normal form is sufficient to let `PROVE_TAC` finish the proof.

When you are experimenting with different definitions, sometimes HOL starts to act funny. This is because HOL remembers definitions and theorems in a global database, and changes to definitions may invalidate existing theorems. When this happens, you have to terminate and restart your HOL session.

One of the nicest ways to define a new symbol in HOL is the `Define` command. Unfortunately, it quietly quantifies over any undefined symbols in your definition. This is almost never what you wanted it to do. The first sign that this happened is usually much later, when HOL reports a unification error.

HOL is not good with basic set-theory: proving that

$$x \in \{y, z\} = (x = y \vee x = z)$$

is surprisingly difficult. Using this in a proof is even more difficult. If you have the left hand side of this in a definition, you should seriously consider changing your definition to use the right hand side instead.

HOL does not provide some basic manipulations: the philosophy of the authors of HOL is that these manipulations lead to fragile proofs. They don't support reffering to specific assumptions, in case the order changes due to changes in your definitions. Sometimes this makes finding a proof very hard. We had to implement some of these manipulations ourselves. The authors of HOL are right about the fragile proofs. Once you have the final proof, it is worth the effort to clean it up and get rid of anything that might depend on specific assumptions or the order of assumptions.

The tutorial on HOL [10] describes some aspects of proof clean-up. Unfortunately, they make it sound like a waste of time. Their description of people that really clean up their proofs a lot makes them seem compulsive. We found clean proofs essential to getting HOL to support the writing of this thesis.

By a clean proof, we mean one in which almost all the work is done by two or three invocations of `RW_TAC`, `FULL_SIMP_TAC` and/or `PROVE_TAC`. These commands can't handle induction, so many clean proofs start with `Induct_on` x. If you need much more than these in a proof, you should seriously consider introducing some lemmas that can be named when calling `RW_TAC`, `FULL_SIMP_TAC` or `PROVE_TAC`.

We worked with `hol98` until very recently. Half way through, a new version came out - *kananaskis-hol*, but we were worried about the effort needed to migrate to it. Finally we did migrate to it, and we found it quite easy. A few symbols changed, and *type-instantiation* (we won't define that here) was notationally different, but, because we had thoroughly polished our proofs by then, the entire migration took only a few hours. Kananaskis-hol seems much better (especially the documentation), so we recommend future users not use any earlier version.

We started with the `set` theory provided by HOL, and eventually moved to the `pred_set` theory. We recommend others go straight to `pred_set` theory (if you use kananaskis-hol, you may not even realize you have a choice).

A.2 Future Work Involving HOL

In this section we describe some areas for future work suggested by our experience with HOL.

Consider the list $[a, b]$. In HOL, it is tedious, but not hard, to prove that it has exactly three prefixes: $[\,]$, $[a]$, and $[a, b]$. The definitions and theorems used in such a proof are quite similar to what we would see in Prolog. Prolog can tell us what the prefixes are, while HOL can only help us prove that we know what they are. It would be interesting to add a Prolog-style query facility to HOL.

HOL has a number of backtracking proof searchers (`PROVE_TAC` is one of them). Currently, HOL is single threaded, and you run one of these and wait for it to succeed or run into a limit on search depth. If you get fed up, you abort the search and proceed by hand. It would be interesting to build a multi-threaded interface that allowed you to initiate a search in the background while you continued to work by hand in the foreground.

Once a proof has been found, cleaning it up is often fairly routine. An automated proof-cleaner would be interesting.

Judging by the literature routinely published in the HOL community, there is an endless supply of things that should be formalized in HOL.

The provers provided by HOL seem rather ad-hoc. The documentation does not give the reader a feel for what they can do or when they should be used. As a result, it takes a great deal of trial-and-error to get to know how to use them effectively. HOL would benefit a great deal from work in this area.

The HOL `bosslib` provides only the most rudimentary support for proof search. It does not even record the steps taken so far in the current proof attempt. Other systems do much better.

The dogmatic approach in the proof support is a bad idea. We should always make simple things simple. Often, the user needs to get the proof done just to make sure the assertion is really true. This dogmatic interface can leave the user wondering if the problem is just their inability to work around the artificial constraints imposed by HOL.

In terms of our thesis, there are three things we would like to do with HOL in the future. First, we would like to build a better set of theorems for working with reachability in the presence of graph mutations. Second, we would like to go back to our early, failed proofs, and see if our improved skills would allow them to go through. Third, we would like to see if we can make our proofs even cleaner by the proper application of simplification sets.

Appendix B

Symbols Used in this Thesis

$[\,]$	$p.14$	The empty list.
$::$	$p.14$	The list constructor.
$\|S\|$	$p.11$	The cardinality of S.
S^*	$p.14$	The Kleene closure of S.
$x.y$	$p.14$	Field y of record x.
$\mathbb{A}$	$p.17$	A set of actions.
A_c	$p.27$	The set of cell addresses.
`achievable`	$p.22$	Are there are actions that lead to this state?
`actionCounts`	$p.56$	Does this action count in this managed memory state?
`active`	$p.51$	Is the root name in use in a managed memory state?
`activeNames`	$p.32$	The active root names field in a rooted memory state.
$\mathtt{add}_\mathtt{a}$	$p.17$	Add to a rod of the abacus-machine.
`addRN`	$p.39$	Defines the add-root-name actions for $\mathbb{M}_m$.
$\mathtt{addRN}_\mathtt{R}$	$p.34$	Add a root name to a rooted memory.
$a_{default}$	$p.38$	The default cell address for the managed memory machine.
`alCounts`	$p.56$	Does this list of actions count in this managed memory state?
$\mathbb{A}_m$	$p.42$	The managed memory actions.

`append`	$p.14$	Append two lists.
A_r	$p.24$	The set of RAM addresses.
$\mathbb{A}_r$	$p.25$	The set of actions for the RAM machine.
$\texttt{assignRN}_\texttt{R}$	$p.35$	Assigns an address to a root name in a rooted memory.
$\mathbb{B}$	$p.11$	The set of boolean values.
C	$p.27$	The set of cell-contents records.
$\vec{C}$	$p.28$	The set of cell memory states.
$\texttt{caAtRN}_\texttt{R}$	$p.32$	Where does this root name lead?
`choice`	$p.40$	Choose some element from a nonempty set.
`cm`	$p.32$	The cell memory component of a rooted memory state.
`computationCounts`	$p.56$	Does this computation count?
`computationEq`	$p.22$	Are these states equivalent under computations?
`compute`	$p.19$	Perform a computation.
$\texttt{connected}_{\vec{\texttt{C}}}$	$p.33$	Are two addresses connected in a cell memory?
D	$p.24, 27$	The set of data values.
`data`	$p.27$	The data field of a cell-contents record.
$d_{default}$	$p.40$	The default data value for newly allocated cells.
`delete`	$p.11$	Remove an element from a set.
`do`	$p.19$	Perform some actions.
`embeddable`	$p.60$	Is one managed memory state embeddable in another?
$\texttt{floodPFs}_\texttt{C}$	$p.29$	Set all the pointer fields for a cell-contents record.
$\texttt{floodPFs}_{\vec{\texttt{C}}}$	$p.31$	Set all pointer fields of a cell in a cell memory.
$\texttt{floodPFs}_\texttt{R}$	$p.35$	Set all pointer fields of a cell in a rooted memory.
`followPFN`	$p.41$	Defines the follow-pointer-field actions for $\mathbb{M}_m$.
$\texttt{followPFN}_\texttt{R}$	$p.35$	Follow a pointer field in a rooted memory.
$\texttt{getContents}_{\vec{\texttt{C}}}$	$p.30$	Get cell contents from a cell memory.
`getData`	$p.42$	Defines the get-data viewpoints for $\mathbb{M}_m$.

$\mathtt{getData}_{\mathtt{C}}$	$p.29$	Retrieve data from a cell-contents record.
$\mathtt{getData}_{\vec{\mathtt{C}}}$	$p.30$	Get data from a cell memory.
$\mathtt{getData}_{r}$	$p.26$	Defines the get-data viewpoints for the RAM machine.
$\mathtt{getPF}_{\mathtt{C}}$	$p.29$	Get a pointer from a cell-contents record.
$\mathtt{getPF}_{\vec{\mathtt{C}}}$	$p.30$	Get a pointer from a cell memory.
`if - then - else`	$p.12$	The conditional expression.
$\Im$	$p.11$	The image of a set under a function.
`insert`	$p.11$	Insert an element into a set.
`let - in`	$p.12$	A way of introducing local variables.
`limitedPcSafe`	$p.58$	Is pause-and-change safe for computations that count?
$\mathbb{M}$	$p.17$	An action-view machine.
m_0	$p.39$	The initial state of the managed memory machine.
$\mathbb{M}_m$	$p.36, 42$	The managed memory machine.
$\mathbb{M}_r$	$p.26$	The RAM machine.
$\mathbb{N}$	$p.11$	The set of natural numbers.
`new`	$p.41$	Defines the cell-allocation actions for $\mathbb{M}_m$.
`nonDegenerate`	$p.59$	Is this map one-to-one on the reachable addresses?
N_{pf}	$p.27$	The set of pointer field names.
N_r	$p.32$	The set of root names.
$\emptyset$	$p.11$	The empty set.
$\mathbb{O}$	$p.17$	A set of observables.
$\mathbb{O}_m$	$p.42$	The managed memory observables.
$\mathbb{O}_r$	$p.26$	The set of observables for the RAM machine.
$\wp$	$p.11$	The power set.
`pcSafe`	$p.22$	Is it safe to pause and use this map to change the state?
`pfMap`	$p.27$	The pointer-field-map of a cell-contents record.
`preservesContents`	$p.59$	Does this map preserve the contents of reachable cells?

`preservesResult`	*p.*21	Does the new state preserve the computation's result?
`preservesRootMap`	*p.*59	Does this map preserve the structure of the root map?
R	*p.*32	The set of rooted memory states.
$\Re$	*p.*11	The range of a function.
$\mathtt{ram}_0$	*p.*24	The initial state for the RAM machine.
`reachable`	*p.*13	Is a node reachable from a set?
`reachableInN`	*p.*13	Is a node reachable from a set in n steps?
$\mathtt{reachable}_{\mathtt{m}}$	*p.*43	Is an address reachable in a managed memory machine?
$\mathtt{reachable}_{\mathtt{R}}$	*p.*34	Is an address reachable in a rooted memory?
`reachablyContained`	*p.*48	Does one managed memory state reachably contain another?
`reachablyEq`	*p.*62	Are two managed memory states reachably equivalent?
`removeRN`	*p.*39	Defines the remove-root-name actions for $\mathbb{M}_m$.
$\mathtt{removeRN}_{\mathtt{R}}$	*p.*35	Remove a root name from a rooted memory.
`rm`	*p.*37	The rooted memory field of a managed memory state.
`rnMap`	*p.*32	The root name map field in a rooted memory state.
`rootNamesAgree`	*p.*45	Do the root names lead to the same address in both managed memory states?
$\mathtt{rod}_{\mathtt{a}}$	*p.*17	The number on a single rod of the abacus-machine.
$\mathbb{S}$	*p.*17	A set of states.
s_0	*p.*17	The initial state of $\mathbb{M}$.
$\mathbb{S}_m$	*p.*37	The set of managed memory machine states.
$\mathbb{S}_r$	*p.*24	The set of states for the RAM machine.
`sameContents`	*p.*45	Does the cell have the same contents in both managed memory states?
$\mathtt{setContents}_{\vec{\mathtt{C}}}$	*p.*30	Set cell contents in a cell memory.
`setData`	*p.*42	Defines the set-data actions for $\mathbb{M}_m$.
$\mathtt{setData}_{\mathtt{C}}$	*p.*29	Set data in a cell-contents record.

$\mathtt{setData}_{\vec{\mathtt{C}}}$	$p.30$	Set data in a cell memory.
$\mathtt{setData}_r$	$p.25$	Defines the set-data actions for the RAM machine.
$\mathtt{setData}_{\mathtt{R}}$	$p.35$	Set data in a rooted memory.
$\mathtt{setPF}_{\mathtt{C}}$	$p.29$	Set a pointer field in a cell-contents record.
$\mathtt{setPF}_{\vec{\mathtt{C}}}$	$p.30$	Set a pointer field in a cell memory.
setPF	$p.41$	Defines the set-pointer-field actions for $\mathbb{M}_m$.
$\mathtt{setPF}_{\mathtt{R}}$	$p.35$	Set a pointer field in a rooted memory.
unused	$p.37$	The unused cells in a managed memory state.
update	$p.12$	Change the value of a function at one point.
$\mathbb{V}$	$p.17$	A set of viewpoints.
$\mathtt{v}_{\mathtt{a}}$	$p.17$	View the abacus-machine.
$\mathbb{V}_m$	$p.42$	The managed memory viewpoints.
vpCounts	$p.51$	Does this viewpoint count in this managed memory state?
$\mathbb{V}_r$	$p.26$	The set of viewpoints for the RAM machine.
$x\ \mathtt{with}\ y \leftharpoonup z$	$p.14$	Record x with field y updated to z.

Bibliography

[1] *Report of the Inquiry Into the London Ambulance Service, February 1993.* The Communications Directorate, South West Thames Regional Authority ISBN 0-905133-70-6.

[2] *Funk & Wagnalls Canadian College Dictionary.* Fitzhenry & Whiteside, Markham, Ontario, Canada, 1963.

[3] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 89–100. MIT Press, 1991.

[4] Andrew W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

[5] Howard Bowman, John Derrick, and Richard E. Jones. Modelling garbage collection algorithms. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, California, USA, August 14–17, 1994. ACM.

[6] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999.

[7] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, Carel S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[8] Healfdene Goguen, Richard Brooksby, and Rod M. Burstall. Memory management: An abstract formulation of incremental tracing. In *Types for Proofs and Programs, International Workshop TYPES'99*, pages 148–161. Springer, 2000.

[9] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[10] The HOL system. WWW Page. http://hol.sourceforge.net/.

[11] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, USA, 1979.

[12] Paul Jackson. Verifying a garbage collection algorithm. In *Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, September 1998.

[13] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996.

[14] Richard E. Jones. The garbage collection bibliography. WWW Page, 1996. http://www.cs.ukc.ac.uk/people/staff/rej/gcbib/gcbib.html.

[15] Samuel N. Kamin. *Programming Languages - An Interpreter-Based Approach*. Addison-Wesley, 1990.

[16] Zohar Manna and Richard Waldinger. Problematic features of programming languages: A situational-calculus approach. *Acta Informatica*, 16:371–426, 1981.

[17] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[18] Simon L. Peyton Jones and David R. Lester. Implementing functional languages: a tutorial. Prentice Hall, 1992.

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

[20] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.