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

A Formal Characterization of Garbage Collection

Danny House

Master of Science

Graduate Department of Computer Science

University of Toronto

2002

A storage manager provides its client with an abstract view of memory. We give a full formal presentation of this abstract view. This abstraction allows garbage collection techniques to be used. Discussions of garbage collection typically start by asserting that it is sufficient to preserve the reachable subgraph. We use HOL to give a fully verified proof of this theorem.

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

I would like to acknowledge the contributions of Marsha, Albert, Benet, and Arie.

Contents

begin questions from July 12

What is the area I am working in, and why is it interesting? What is missing in the existing state of this field, and which missing bits do I want to address? Who else is working in this area, and what is the state of the art? What are the immediate goals of this thesis? How will I expand on these goals in the future?

Q: What is the area I am working in, and why is it interesting? A: This work is in the area of formal methods applied to garbage collection. Garbage collection is interesting because of all the ways that it reduces the opportunities for programmers to make errors. Formal methods are interesting because they present new opportunities to get machine support for the construction of reliable software. In order for garbage collecting systems to realize their full value, they must be completely reliable.

Q: What is missing in the existing state of this field, and which missing bits do I want to address? A: The problems of reasoning about garbage collection are not easily addressed in formal methods which avoid side-effects and aliasing. There have been some of articles proving that specific garbage collection algorithms are correct. These have mostly been about the on-the-fly algorithm of Dijkstra et. al. We have only been unable to find a single formal treatment of the fundamental theorem, the heap may be changed so long as the reachable subgraph is preserved

Q: Who else is working in this area, and what is the state of the art? A: Goguen Reynolds a couple others

Q: What are the immediate goals of this thesis? A: To prove the fundamental theorem the heap may be changed so long as the reachable subgraph is preserved To prove that a specific way of changing the heap does preserve the reachable subgraph.

Q: How will I expand on these goals in the future?

end questions from July 12

begin drivel

I like to build things. In particular, I like to write programs. Recently I began to feel that formal methods (FM) could help me write programs, but I don't know how to bridge the gap. I can use FM to analyse abstractions, and thus avoid certain kinds of mistakes, but very few of the programs I have ever worked on would benefit from this. The ones that would benefit all involved complicated, homegrown and obscure algorithms. The majority of software is built out of standard algorithms with various minor alterations to fit or take advantage of local circumstances.

As a learning experience, I built a lisp interpreter (about 15 years ago). At no point did I require any unusual algorithms. Garbage collection (GC) was one of the more interesting aspects of building it, and choosing the GC strategy was an interesting design problem. During this construction there were many places where I had to convince myself that the code was correct, and these are the places where I would like support from an FM tool. I still don't know how to get that support.

In the course of this thesis, I have worked on a very small part of this problem. The only thing that I have found is that it is safe to perform GC (no surprise there, but the experience has been good). In order to get even this small result, I needed to model memory, graphs, and computation. All of these have been modeled before, but, as when writing most software, the models were not trivially reusable.

In order to get the rest of the way to my goal, at least the following must be done:

- some FM tool must be made relatively easy to use - the FM tool must include a large library of standard models (PVS and HOL already have large parts of mathematics, but not so much of physics, economics, agents, astronomy, or the many other problem domains in which we apply software) - some language (L) must be fully formally modeled - compilation must be the most that is needed to execute L - L must be a popular language for software development - the FM tool must provide extensive support for reasoning about L - the FM tool must include a large library of standard algorithms

I remain convinced that FM will eventually be common in software development. Unfortunately, eventually looks like at least 20 years from now.

end drivel

begin drivel

Most software developers have not even heard of formal methods (FM). After my first encounter with FM (about 10 years ago), I tried to explain the ideas and goals to a colleague in the workplace, and he said "but that's just what I do anyway". Although this is true, and in spite of the fact that my friend is exceedingly gifted in many ways, and has a masters degree in pure mathematics, it is not at all clear the either he or I could productively apply formal methods to the work we were doing at that time.

Originally, my goals were quite ambitious, and would have required a full formal model of some machine or programming language. Such a model would also be a prerequisite for providing my friend with machine support for what he "just does anyway".

In the course of my work on this thesis, I have applied formal methods to the problem of garbage collection. Because of the scale of the problem, I have not yet taken it to the level of a programming language (although this work has felt a lot like programming). Nonetheless, it is clear that HOL is much too difficult to work with - there is no way that it could be used by even 10I have worked with. This is also true of PVS, SMV, SPIN and CW.

It has been suggested that the reason that FM is having problems penetrating the workplace is because the practitioners are not good at abstraction. This may well be a significant reason why typical specifiers and designers cannot find good uses for today's tools, but it is difficult to believe that they will even try.

HOL makes simple things difficult. Although it provides the syntax for the theorem (x1,x2 SUBSET) ==¿ P it requires quite a bit of work to proove it.

end drivel

begin drivel

My experience in the workplace suggests that it is still true that a large percentage of today's professional software developers have a limited amount of relevent training.

On the other hand, there does not appear to be a strong correlation between relevent training and quality code. The key ingredient is attitude - quality code is written by people that care about quality.

end drivel

Chapter 1

Introduction

In this thesis we will prove that garbage collection is safe. To do this, we must first say what it means to be safe. Then we must give a definition of garbage collection, and show that it satifies our notion of safety.

Garbage collection is something that is done as part of providing programmers with a relatively simple model of memory. In broad terms, this means that garbage collection is safe if it does not change the outcome of computation. Because of this, we need to describe the model of memory that is to be presented, and give a model of computation in which it is clear where memory sits. Having done this, we will be able to say that garbage collection is part of the implementation of the memory model, and assert that the safety condition is just the requirement that the implementation meet its specification.

We know that garbage collection is an optimization technique, but we need to be more specific. We go on to characterize garbage in the programmers model of memory, and then show that the garbage does not need to be preserved by the implementation.

WHAT DOES IT MEAN TO BE SAFE?

An optimization is safe when it does not change the specified behaviour of a program. In the case of garbage collection, the program is most often a language interpreter or run-time environment, so we need a very general description of its behaviour. For this reason we assume that the program is divided into two parts, a client and a storage manager, and that the division is clean enough that the overall behaviour of the program will be the same so long as the storage manager meets the specification that will be given below.

We have choosen to model the behaviour of a program that uses garbage collection using a varient of the situation calculas. This representation closely resembles the trace-assertion approach.

We model the machine as an action/observer system (to be defined later).

We will assume that the machine state is comprised of two disjoint parts, one of which will be called the heap. We assume that the heap can only be changed by a small range of actions, addHandle, newCell, setLink, getLink, setData (these will be formalized later). Additionally, we assume the existance of one more kind of action, getData, that does not affect the heap. We postulate the existance of some set of On this view, a program is a sequence of actions, some of which come from those just named,

Let M be an action-observer system, which we will call the machine. We define a program to be a sequence of actions followed by an observer.

Suppose we are able to decompose M into two independent subsystems, M1 and M2 such that

Let M' be a subsystem of M. Assume that none of the actions not in M' change the state of M'. Assume that none of the actions not in M' depend on the state of M'. Assume that none of the observers not in M' depend on the state of M'.

we identify a subsystem, A1 of action-observer system A we say that A1 is a module of A if their observers are independent

an observer is a function on the state it does not affect the state in a larger system, the observer returns a value in the larger system, the observer is an action observers of subsystems are actions of supersystems there is no clear reason to limit the amount of change that an observer can cause in the supersystem's state an action of the subsystem changes the state of the subsystem it does not change the state of the supersystem the changes to the subsystem do not depend on the state of the supersystem an observer of the subsystem changes the state of the supersystem it does not change the state of the subsystem the changes to the supersystem do depend on the state of the subsystem

Chapter 2

Introduction

It has often been observed that a large proportion of software bugs arise from either using pointers improperly or releasing memory at the wrong time. The effective use of these problematic techiques is essential to the construction of many important applications such as operating and database management systems. Garbage collecting storage managers are often recommended as a way to reduce the occurence of these bugs. This makes the study of garbage collection techniques valuable.

Existing approaches to formal methods do not address side-effects, pointers, resource allocation and update-in-place well. All of these play a role in garbage collection. This suggests that a rigorous study of garbage collection may lead to new ways of extending formal methods.

In this thesis we give a formal specification of a very simple storage manager (SM) and a garbage collecting storage manager (GCSM). We also say what it means for the GCSM to simulate the SM, and then prove that it does.

2.1 What are Storage Managers?

We take a storage manager to be a machine that allows us to store and update data and the links between data items, to assign and reassign names to data, and to retrieve data using names and links. A data item may have any number of names or incoming links. For simplicity, we assume that there may be any number of outgoing links for each data item. Thus a storage manager is a machine that responds to the following commands

$$AddHandle(h : handle)$$
$$NewCell(h : handle)$$
$$SetLink(h1 : handle, i : num, h2 : handle)$$
$$GetLink(h1 : handle, h2 : handle, i : num)$$
$$SetData(h : handle, d : data)$$
$$d : data = GetData(h : handle)$$

Notice that this point of view never mentions deleting anything. The user of a storage manager would prefer to just keep adding and updating.

2.2 Related Work

2.2.1 Garbage Collection

The liturature on garbage collection is huge, but is focused on how to collect garbage in various settings, and how to do it efficiently. A good book is Jones. A good survey is ???. Garbage collection as-such is not really relevent to this thesis - it does not address any particular real algorithm. This area is not formal.

2.2.2 Refinement

This formulation of garbage collection and the related proofs of correctness fall under the category of data-refinements.

Observational Equivalence

Milner.

Data Refinement

'Data Refinement: Model-Oriented Proof Methods and their Comaprison', Willem-Paul de Roever, Kai Engelhardt

What kind of simulation am I proving? L, L-1, U, U-1?

What would be the transformer in Ric's approach, if it turns out that my work is an L-simulation.

2.2.3 Actions and Observers

Situation Calculas

Manna & Waldinger

Trace Assertions

Parnas

2.2.4 Garbage Collection and Formal Methods

In this section we review the section of the liturature that applies formal methods to garbage collection.

¿¿¿¿Reynolds - Separation Logic: A Logic for Shared Mutable Data Structures This is a logic for talking about stores and heaps.

Devides the state into a store and a heap. As usual, stores map variables to values. Heaps map addresses to values, which are always integers. Addresses have activity bits. Uses 'cons', v:=[e] (lookup), [e]:=e' (mutation), 'dispose'. I don't use dispose. Like Parnas' trace assertions, Reynods assumes that memory faults occur when the arguements to 'lookup', 'mutation' or 'deallocation' are inappropriate (ie, when an address is used that does not have its activity bit set). Uses a transition relation between 'configurations', but the input configuration is always a state+command while the output configuration is always either a state or 'abort'. Distinguishes between expressions and commands. Expressions do not have side effects and do not reference the heap. Describes his semantics of assertions (page 4) as extending classical semantics to an instance of the 'resource semanitcs' of David Pym.

Need to look at SPACE 2001: Informal Proceedings of Workshop on Semantics, Program analysis and Computing environments for Memory Management in which Yang's proof of the Schorr-Waite algorithm is presented. This is very relevent.

Says that garbage collection for his logic raises some very difficult problems, as it may falsify assertions that should never be falsified. What he means is that if garbage collection is applied to the heap that his logic discribes, then the new heap may falsify these assertions. Thus his logic allows too much to be said about the heap.

Reynolds says Calcagno, O'Hearn and Bornat have some interesting alternatives.

Goguen Gives a formalization of memory. Has a great deal in common with this thesis. Is focused on the three-color marking technique used in garbage collection, but avoids the actual issue of collection.

Dijkstra et. all Ben Ari a couple others

2.2.5 Graph Theory

Only used basic results. All basic results had to be proven in HOL, since the HOL community does not have any existing support. Two articles in HOL liturature, not very helpful.

Problems with Existing Applications of Formal Methods

generally restricted definition of correctnes:

1) does not free reachable nodes

2) eventually frees all unreachable nodes

Additionally, we want to know that the structure of the graph and the data at each node are preserved.

Most formal treatments of garbage collection deal with collectors that do not move the nodes in memory. Under these conditions, it looks easy to show data and structure are preserved.

Examples of Limited Correctness Criteria

halfdene goguen

experience with predicate abstraction

other cav paper

2.3 Background

Typed-lambda calculas. Does not support natural manipulation of terms. Easy to express relevent concepts and theorems. No support for graph-theory. Does have limited support for transitive closure of relations.

2.3.1 HOL

2.4 Notation

THIS SECTION IS CURRENTLY SOMETHING OF A RUBISH BIN

In what follows, $\mathbb{B}$ denotes the set of Boolean values $\{\top, \bot\}$ and $\mathbb{N}$ denotes the natural numbers.

If R and S are sets, then we use S^R to denote the set of total functions from R to S.

By viewing sets as characteristic functions, we use $\mathbb{B}^R$ to denote the set of all subsets of R, the *powerset* of R.

Let $[j, ..., k]$ denote the set of integers ranging from j to k inclusively.

By viewing arrays as functions on integer ranges, we use $R^{[0,\ldots,k-1]}$ to denote the set of zero-origined, size k arrays of elements of R.

For a set S, S^* is the set of finite lists of elements of S, $[\,]$ stands for the empty list, and $h::t$ denotes the list obtained by adding element h to the front of list t.

We will use $S_1 \oplus S_2$ to denote the disjoint union of sets S_1 and S_2 while $|S|$ is the cardinality of S.

$x \in S$ means that x is an element of set S.

Rather than invent a new symbol, we will use the formula $|S| \in \mathbb{N}$ to mean that S has finite size.

$\emptyset$ is the empty set.

We will present tuples thus: $\langle x, y, z\rangle$.

PUT A DEFINITION FOR X:A$\rightarrow$ B$\rightarrow$ C HERE

When we have an object which is a tuple (eg, $A = \langle a, b\rangle$), and there is no chance of ambiguity, we will sometimes refer to its components directly (eg, a). If there is a need to explicitly specify the object then we will use the form $A.b$. When we have nested objects (eg, $A = \langle a, \langle b, c\rangle, d\rangle$), and it will not cause confusion, we will quietly flatten them (eg, $A.c$ or just b).

If T is a tuple with components $t_1, ..., t_k, t_{k+1}, ..., t_m$, we will use

$$T \texttt{ with } t_1 \leftharpoondown e_1, ..., t_k \leftharpoondown e_k$$

to mean the tuple that is the same as T except at the components $t_1, ..., t_k$, and which has the value of component t_i equal to e_i for i between 1 and k.

By the usual abuse of notation, we will quietly apply functions and relations to subsets of their domain. If $\mathcal{R} \subseteq X \times Y$ and $S \subseteq X$, then $\mathcal{R}S = \{y \mid myExistsx \in S\langle x, y\rangle \in \mathcal{R}\}$.

We presume that variables are implicitly typed, and that all free variables are implicitly universally quantified over their domains.

2.5 Outline of this Thesis

Chapter 3

Motivation

The use of many garbage collection algorithms rests on the following informal assertion:

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

In this thesis we present a machine-checked proof that this is true. There are many assumptions that lie behind this assertion, and many fundamental concepts that must be formally defined before we can embark on the proof itself. Here are just a few of the questions that we must answer in preparation for the proof.

- What is a machine?
- What is the result of a computation?
- What does it mean to pause the machine?
- What is memory?
- What is a pointer?
- What does it mean to alter memory while the machine is paused?

- Which parts of memory are reachable?

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 randomness.

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 or *transition function*. 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, observing some aspects of the state that follows the last action, and interpreting that *observation* as the answer to some question.

Example 3.1 (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 3.2 (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 summarize, a *machine* is something that has a *state* and responds to *actions* by changing its state according to some *transition function*.

What is the result of a computation? As children we learn to compute, a task that consists of accepting a sequence of instructions, using a pencil and some paper to carry out an appropriate sequence of activities, and producing an answer at the end. In this context, the machine is the student that performs the task, plus the paper and pencil; so "performing an action on the machine" means "giving an instruction to the student". (From now on we use the words *action* and *instruction* interchangeably.) It is understood that aspects such as hair color and pitch of voice are irrelevant. When the answer is expected to be a diagram, the instructions say so and the teacher looks at the paper submitted, and nothing else.

In the same way, the result of a computation is not the final state of the machine, or even the final state of the observable parts of the machine. In the foregoing discussion of machines, we said the user must make an observation and interpret it. We expect the user's decision of what to observe at the end is coordinated with her choice of the list of instructions, which we call the *program*; so there is a sense in which the program tells the machine where to put the answer. We say that the *result of the computation* is the result of the observation performed after the program is fed to the machine (*i.e.*, after the actions are performed on the machine). Notice that the result of the computation is the actual observation, not its interpretation. For us, the *computation* is the collection of activities carried out by the user to feed the program to the machine and make the observation at the end, and the activities of the machine as it *executes* the program (*i.e.*, reacts to the actions performed upon it).

Example 3.3 (Divisibility) *Consider using a 4-function calculator to determine if 3274 is evenly divisible by 7. Our hypothetical user is familiar with the calculator to be used and knows that it displays integers without a decimal point. Based on this, she chooses to press the buttons 3, 2, 7, 4,* $\div$*, 7, and* $=$ *in order, and makes the observation "there is a decimal point in the display". The machine is the calculator, the program is the sequence of button pushes, the result of the computation is "there is a decimal point", and the interpretation is "3274 is not divisible by 7".*

To summarize, the *result of a computation* is the *observation* that is made after a *program*, which is a sequence of *instructions*, has been *executed* by a machine.

What does it mean to pause the machine? When watching an old pendulum regulated grandfather clock, we see the the pendulum swing back and forth. In time with the swing, we see the second hand jump. If we open the back and look inside, we see gears moving. If we imobilizing the pendulum, all movement stops. If we release the pendulum, movement begins again, just as if it had never been interrupted.

Our intuition is that as a machine executes a program, it moves through a sequence of states. To pause the machine is to temporarily halt this motion. If we include hidden states in the sequence, then it need not be in direct correspondence to the sequence of instructions that makes up the program.

Chapter 4

Definitions

4.1 What is a machine?

At a very low level, a machine may be viewed as something that has a state and responds to an instruction by changing state.

Definition 4.1 (Low-Level Machine) *A* Low-Level Machine *or* LLM *is a 4-tuple,*

$$\langle S, s_0, \Sigma, \delta \rangle,$$

where

- S *is the (possibly infinite) set of* states*,*
- $s_0 \in S$ *is the* initial state*,*
- Σ *is the (possibly infinite) set of* inputs*, and*
- $\delta : \Sigma \to S \to S$ *is the (total)* transition function*.*

We use the terms input*,* instruction *and* action *synonymously. We obtain the* extended transition function*, $\widehat{\delta}$, from δ by repeated application; so $\widehat{\delta} : \Sigma^* \to S \to S$ is defined by*

$\widehat{\delta}([\,], s) = s$*, and*

$\widehat{\delta}(a :: w, s) = \widehat{\delta}(w, \delta(a, s))$*,* *where* $s \in S$, $a \in \Sigma$, *and* $w \in \Sigma^*$. *A* program, *p, is an element of* Σ^*, *and* feeding it to a machine *is equivalent to evaluating* $\widehat{\delta}(p, s_0)$.

THERE IS A KIND OF TRANSITIVE CLOSURE RELATION FROM δ TO $\widehat{\delta}$, BUT SINCE δ IS NOT A RELATION IN $S \times S$, AND NEITHER IS $\widehat{\delta}$, A QUICK STATEMENT LIKE "THUS, $\widehat{\delta}$ IS THE TRANSITIVE CLOSURE OF δ FROM s_0 " SEEMS TO BE AN ABUSE OF TERMINOLOGY.

4.2 Action/Observer Systems

Definition 4.2 (Action/Observer System) *An* Action/Observer System *or* AOS *is a 4-tuple,*

$$\langle S, s_0, \Sigma_a, \delta, O, \Sigma_q, \Omega \rangle,$$

where

- S *is the (possibly infinite) set of* states,
- $s_0 \in S$ *is the* initial state,
- Σ_a *is the (possibly infinite) set of* actions,
- $\delta : \Sigma_a \to S \to S$ *is the (total)* transition function
- O *is the (possibly infinite) set of* observables,
- Σ_q *is the (possibly infinite) set of* queries, *and*
- $\Omega : \Sigma_q \to S \to O$ *is the (total)* observation function.

As before, we obtain the extended transition function, $\widehat{\delta}$, *from* δ *by repeated application; so* $\widehat{\delta} : \Sigma_a^* \to S \to S$ *is defined by*

$\widehat{\delta}([\,], s) = s, and$

$\widehat{\delta}(a :: w, s) = \widehat{\delta}(w, \delta(a, s)),$

where $s \in S$, $a \in \Sigma_a$, *and* $w \in \Sigma_a^*$. *We say that a* computation *is specified by a pair,* $\langle p, q \rangle$, *where* $p \in \Sigma_a^*$ *is a* program *and* $q \in \Sigma_q$ *is a* query, *and the* result of the computation *is*

$$\mathcal{R}(p, q) = \Omega(q, \widehat{\delta}(p, s_0)).$$

4.3 Memory Graphs

Definition 4.3 (Memory Graph) *A* Memory Graph *or* MG *is a 9-tuple,*

$$\langle N, \boxdot, H, \rho, \mathcal{L}_e, \uparrow, \mathcal{L}_n, \mu, \Delta \rangle,$$

where

- N *is the (possibly infinite) set of* nodes,
- $\boxdot \in N$ *is the* nil *node,*
- H *is the (possibly infinite) set of* handles,
- $\rho : H \to N$ *is the (total)* root function,
- $\mathcal{L}_e$ *is the (possibly infinite) set of* edge labels,
- $\uparrow: N \to \mathcal{L}_e \to N$ *is the (total)* link function,
- $\mathcal{L}_n$ *is the (possibly infinite) set of* node labels,
- $\mu \in \mathcal{L}_n$ *is the* unspecified *or* default *node label, and*
- $\Delta : N \to \mathcal{L}_n$ *is the (total)* data function.

When

$N = \{\boxdot\}$,

$H = \emptyset$,

$\uparrow = \lambda n \in N, e \in \mathcal{L}_e \cdot \boxdot, and$

$\Delta = \lambda n \in N \cdot \mu$,

we say that the memory graph is empty. *We call the node n a* root *when ρ associates it with some handle; $\exists h \in H \cdot n = \rho(h)$.*

```
connected start end m = ?k. end = m.link start k


reachableIn 0 m = (nil \INSERT (IMAGE m.root m.handles))
reachableIn (SUC j) m = (reachableIn j m)
  \UNION {end\mid  ?start. start \IN reachableIn j m
      /\ connected start end m}


reachable n m = ?j. n \IN reachableIn j m


reachableNodes m = {n\mid reachable n m}


reachableMemory m = (m with
    <\mid  nodes := reachableNodes m;
       link := \n i. if reachable n m then m.link n i
      else nil;
       data := \n. if reachable n m then m.data n
    else unspecifiedData
     \mid >)


handlePath =
    <\mid  startHandle : handle;
```

```
       indexes     : edgeLabel list
     \mid >

nilPath = <\mid  indexes : edgeLabel list \mid >

validHandlePath hp m = hp.startHandle \IN m.handles

followIndexPath [] n m = n
followIndexPath (hd::tl) n m = m.link (followIndexPath tl n m) hd

indexPathStaysIn [] n m = n \IN m.nodes
indexPathStaysIn (h::t) n m
    = indexPathStaysIn t n m /\ m.link (followIndexPath t n m) h \IN m.nodes

endOfNilPath (np:nilPath) m = followIndexPath np.indexes nil m

endOfHandlePath hp m = followIndexPath hp.indexes (m.root hp.startHandle) m

rootedPath =  rootIsHandle of handlePath
    \mid  rootIsNil of nilPath

validRootedPath (rootIsHandle hp) m = validHandlePath hp m
validRootedPath (rootIsNil np) m = T

endOfRootedPath (rootIsHandle hp) m = endOfHandlePath hp m
endOfRootedPath (rootIsNil np) m = endOfNilPath np m
```

```
homomorphism hom g1 g2
    = (    (hom nil = nil)
/\ (IMAGE hom g1.nodes \SUBSET g2.nodes)
/\ (!h. h \IN g1.handles ==> (hom (g1.root h) = g2.root h))
/\ (!n i. n \IN g1.nodes ==> (hom (g1.link n i) = g2.link (hom n) i))
/\ (!n. n \IN g1.nodes ==> (g1.data n = g2.data (hom n))))

homomorphic g1 g2 = g1.handles \SUBSET g2.handles /\ ?hom. homomorphism hom g1 g2

isomorphism hom g1 g2 = (homomorphism hom g1 g2 /\ BIJ hom g1.nodes g2.nodes)

isomorphic g1 g2 = (g1.handles = g2.handles) /\ ?iso. isomorphism iso g1 g2
```

4.4 Memory System

MUST TALK ABOUT WHY `setData` TAKES A HANDLE, NOT A NODE

Consider changing the label of a root node in a memory graph, m. We do this by updating the data function, Δ, so that when Δ is applied to the node indicated by handle h, it yields the new value, d, and at all other nodes Δ is unchanged.

$$\texttt{setData}\ (h, d, m) = m\ \texttt{with}\ \Delta \leftharpoonup \lambda n \cdot\ \texttt{if}\ n = \rho(h)\ \texttt{then}\ d\ \texttt{else}\ \Delta(n).$$

This puts us in a position to (informally) define a low-level machine, $\langle S, s_0, \Sigma, \delta \rangle$, where

- S is the set of all memory graphs,
- s_0 is an empty memory graph,
- $\Sigma = \mathcal{H}$ is the set of all *possible handles*, and

- $\delta =$ `setData` .

(We will address $\mathcal{H}$ in a formal definition later.)

Now consider extending this machine to support the addition of handles. We add the new handle, h, to a memory graph, m, by inserting it in the set of handles, H, and updating the root function, ρ, so that the new handle leads to $\boxdot$ and the old handles are left alone. The resulting memory graph is

$$\texttt{addHandle}\ (h, m) = m \texttt{ with } H \leftharpoondown H \cup h, \rho \leftharpoondown \lambda h_1 \cdot \texttt{ if } h_1 = h \texttt{ then } \boxdot \texttt{ else } \rho(h_1).$$

This leads to a new machine with the same S and s_0, but

- $\Sigma = \{\langle\ \texttt{addHandle}\ , h\rangle \mid h \in \mathcal{H}\} \oplus \{\langle\ \texttt{setData}\ , h, d\rangle \mid h \in \mathcal{H} \wedge d \in \mathcal{L}_n\}$, and
- $\delta = \lambda a \in \Sigma, m_1 \cdot \texttt{ if } \pi_1(a) = \texttt{ addHandle then addHandle } (\pi_2(a), m) \texttt{ else setData } (\pi_2, a), m)$

Definition 4.4 (Memory System) *A* Memory System *or* MS *is a triple,*

$$\langle \mathcal{N}, \mathcal{H}, AOS\rangle,$$

where

- $\mathcal{N}$ *is the (possibly infinite) set of* possible nodes*,*
- $\mathcal{H}$ *is the (possibly infinite) set of* possible handles*,*
- $AOS = \langle S, s_0, \Sigma_a, \delta, O, \Sigma_q, \Omega\rangle$ *is an action/observer system,*
- S *is a set of memory graphs, and if* $\langle N, \boxdot, H, \rho, \mathcal{L}_e, \uparrow, \mathcal{L}_n, \mu, \Delta\rangle \in S$*, then*
- $N \subseteq \mathcal{N}$ *is the set of* current nodes*,*
- $H \subseteq \mathcal{H}$ *is the set of* current handles*,*
- s_0 *is an empty memory graph,*
- $O = \mathcal{L}_n$*,*

Chapter 5

Action/Observer Systems and Simulation

In this chapter we will define action/observer systems, and what it means for one such system to simulate another. Later we will model garbage collecting systems using action/observer systems, where the actions are the things that clients can do and observers are the queries that clients can make.

An action/observer system is a 9-tuple ¡ setOfStates, initialState, setOfActions, getActionNext, getActionCheck, rangeOfObservers, setOfObservers, getObserverQuery, getObserverCheck ¿,

where setOfStates is some set of states, initialState IN setOfStates is the initial state, setOfActions is a set of actions, getActionNext:setOfActions-¿setOfStates-¿setOfStates tells us how actions affect states, getActionCheck:setOfActions-¿setOfStates-¿bool tells us if the behaviour of an action is specified for a state, rangeofObservers is some set of values, setOfObservers is a set of observers, getObserverQuery:setOfObservers-¿setOfStates-¿rangeOfObservers tells us how to make observations on states, and getObserverCheck:setOfObservers-¿setOfStates-¿bool tells us if the result of making an observation is specified for a state.

Given an action/observer system, we define an execution function, doAll:(setOfActions

*)-¿setOfStates-¿setOfStates which is required to satisfy doAll emptyWord s = s, and doAll a::r s = (getActionNext a (doAll r)) where a is an action and r is a list of actions. Although any list of actions can be executed, the result may be unspecified. When we want to restrict attention to sequences with specified behaviour we use checkAll:setOfObservers-¿setOfActions *-¿bool, which is required to satisfy checkAll emptyWord s = T, and checkAll a::r = (checkAll r) / (getActionCheck a) (doAll r) with a and r as before.

When there is more than one action/observer system AO, we write doAll(AO) and checkAll(AO) to eliminate ambiguity.

The act of performing an observation is denoted by look:setOfObservers-¿setOfActions *-¿setOfStates-¿rangeOfObservers which is defined by look o actionList s = (getObserverQuery o) (doAll actionList s) As with actions, we can always look, but what we are going to see may not be specified. When we want to restrict attention to observations with specified outcome we use checkLook:setOfObservers-¿setOfActions *-¿setOfStates-¿BOOL, which is defined by checkLook o actionList s = checkAll actionList s / (getObserverCheck o) (doAll actionList s)

In order to talk about observers and observations, we require that

$$\sigma_o o\alpha = \sigma_{al}\alpha \wedge o_\sigma(\delta\alpha),$$

which lets us say that the observation observer o

$\lambda : A* \to O \to V$ is the *'look'*

```
look:O->A*->V is the function

    look o al = o.value (do al)
```

```
Let X and X’ be two action/observer systems.
Let a_map:X.A->X’.A and o_map:X.O->X’.O.
By an abuse of notation, the natural extension of a_map to A* will also be
denoted by a_map.  We say that X’ simulates X when

    X.D_o X.o X.al
==> X’.D_o (o_map X.o) (a_map X.al)
    /\ X.look X.o X.al = X’.look (o_map X.o) (a_map X.al)

    checkLook(X) X.o X.al
==> X’.D_o (o_map X.o) (a_map X.al)
    /\ X.look X.o X.al = X’.look (o_map X.o) (a_map X.al)
```

Chapter 6

Formal Specification of a Storage Manager

We define a storage manager to be an action/observer system in which the actions are addHandle, newCell, setLink, getLink, and setData, and getData is the only observer. We feel that the clearest way to specify such a system is to give an imperative example of one storage manager, and to say that a storage manager is any action/observer system that simulates the one exhibited.

IS THIS THE RIGHT SET OF ACTIONS?

6.1 Is This the Right Thing to Build?

6.2 The Specification

For the sake of our example storage manager, we assume the existance of an infinite set of handles. These are the names that the client gives to the storage manager. In a practical garbage collecting system they would be addresses of stack slots, and the contents of those slots would be the addresses of cells in the heap.

To avoid clutter, we assume that the set of nodes is infinite, and that there is a special node called nil. These will never be exposed to the client program.

We will assume that there is a set of data objects, one of which is called 'unspecified-Data'.

We define a state for our example storage manager to be a 5-tuple, ¡handles, nodes, root, link, data¿, where handles is a set of handles, nodes is a set of nodes, root:handles-¿nodes, link:nodes-¿num-¿nodes, and data:nodes-¿data

Since root, link and data are functions on infinite domains, they are certainly not implementable as presented here. root is the way we determine the roots of the graph.

addHandle (h:handle) newCell (h:handle) setLink (h1:handle, i:num, h2:handle) getLink (h1:handle, h2:handle, i:num) setData (h:handle, d:data) d:data = GetData (h:handle)

Now that we have a definition for simulation between action/observer systems, we will give the definition of one storage manager, and say that anything that simulates it is also a storage manager. We will call this special storage manager

SM = <SM, smEmpty, B U N, smObservers, smActions, smDo, smALspecified, smOspecified, smLook>

We assume the existance of NODE_UNIV and HANDLE_UNIV, the sets of all possible nodes and handles respectively.

Each state in SM is a tuple

```
<nodes, nil, handles, root, link, data>
```

nodes SUBSET NODE_UNIV is the current set of nodes, and always contains nil.

handles SUBSET HANDLE_UNIV is a set of objects by which the roots of the graph can be found.

root:handles->nodes allows us to find the roots of the graph.

link:nodes->N->nodes defines the edges of the graph.

data:nodes->N records information stored by the client.

The initial state is smEmpty, and by definition,

```
smEmpty.nodes = {nil},

smEmpty.handles = {},
```

```
    smEmpty.root = smEmpty.link = nil, and

    smEmpty.data = 0.
```

Before defining smObservers, we will define smObserverNames. There are three sets of observer names,

```
    smIsNilNames = {(smIsNil h) | h in HANDLE_UNIV},

    smSameNodeNames = {(smSameNode h1 h2) | {h1, h2} SUBSET HANDLE_UNIV},

    smDataNames = {(smData h) | h1 in HANDLE_UNIV}.
```

Note: smIsNil, smSameNode and smData are arbitrary distinct constants.

We define

```
    smObserverNames = smIsNilNames U smSameNodeNames U smDataNames.
```

We also define a function, smNameToObserver:smObserverNames->smObservers.

For handles h, h1 and h2, and storage manager states s,

```
    (smNameToObserver (smIsNil h)).specified = \s. h in s.handles

    (smNameToObserver (smIsNil h)).value = \s. (nil = s.rootMap h)

    (smNameToObserver (smSameNode h1 h2)).specified
= \s. {h1, h2} SUBSET s.handles

    (smNameToObserver (smSameNode h1 h2)).value
= \s. (s.rootMap h1 = s.rootMap h2)

    (smNameToObserver (smData h)).specified = \s. h in s.handles

    (smNameToObserver (smData h)).value = \s. s.dataMap h
```

We define smObservers to be the range of smNameToObserver.

We use the same approach to define smActions. First we define the following sets of action names:

```
    smAddHandleNames,

    smRemoveHandleNames,
```

```
    smNewCellNames,

    smSetLinkNames,

    smGetLinkNames, and

    smSetDataNames.
```

We will give two examples here, and present the rest in appendix A.

```
    smAddHandleNames = {(smAddHandle h) | h in HANDLE_UNIV}

    smSetLinkNames

= {(smSetLink h1 i h2) | {h1, h2} SUBSET HANDLE_UNIV /\ i in N}
```

Following the same pattern, we define smActionNames to be the union of these name sets, and the function smNameToAction:smActionNames->smActions is used to define smActions. Here is part of the definition of smNameToAction:

```
    (smNameToAction (smAddHandle h)).specified = \s. ~(h in s.handles)

    (smNameToAction (smAddHandle h)).next
```

```
= \s. <s.nodes, nil, {h} U s.handles, s.root, s.link, s.data>

    (smNameToAction (smSetLink h1 i h2)).specified

= \s. {h1, h2} SUBSET s.handles

    (smNameToAction (smSetLink h1 i h2)).next

= \s. <s.nodes, nil, s.handles, s.root,

\h, j. if h = h1 /\ j = i then (s.root h2)

  else (s.link (s.root h) j),

s.data>
```

The definitions of smDo, smALspecified, smOspecified and smLook follow from the definition of action/observer systems.

Chapter 7

Specification of a Garbage Collector

The each operation in the garbage collector does exactly the same thing as its corresponding operation in the storage manager, and then reduces the graph to its reachable subgraph.

The reachable-subgraph-reduction is overspecified, in that it indicates the values of root, link and data in areas of the graph that are not reachable. This was done to make the specification as simple as possible. This way it is clear that there is no garbage present at any time, and we cannot accidentally get a proof of something that is not relevent.

Chapter 8

Verifying the Specification

First we prove that, so long as there is always a graph-isomorphism between the reachable subgraphs of the storage manager and the garbage collector, then the observables are always the same.

This reachable-subgraph-isomorphism is the invarient.

Then we prove that the invarient is preserved by each operation.

Chapter 9

Towards a New Formal Method

Chapter 10

Conclusions / Limitations / Future Work

10.1 Conclusions

We have provided a reusable characterization of garbage collection that can be extended in the following ways.

We have proven that hyper-aggresive garbage collection does not affect the result of the computation.

10.2 Limitations

Real garbage collectors work in environments where pointers change over periods of time. We assumed that the updates were atomic.

We avoided any of the gritty details needed for a real collector.

10.3 Future Work

There is a lot more that needs to be done.

In order to be a more reusable framework for garbage collection, various assumptions must be made parameterizable.

Chapter 11

Appendix - Detailed Specs

¿¿BACKGROUND types: node, data, handle constants: nil IN node, unspecifiedData IN data We postulate an axiom saying that there are infinitely many nodes, so that there is a function aNodeNotIn : node set -¿ node set for which FINITE s -¿ (aNodeNotIn s IN s)

¿¿MANAGED MEMORY GRAPH OBJECT

smGraph wellFormed_def emptySmGraph_def connected_def reachableIn_def reachable_def reachableNodes_def reachableSMG_def handlePath nilPath validHandlePath_def followIndexPath_def indexPathStaysIn_def endOfNilPath_def endOfHandlePath_def rootedPath validRootedPath_def endOfRootedPath_def homomorphism_def homomorphic_def isomorphism_def isomorphic_def

Hol_datatype ‘smGraph = ¡— handles : handle -¿ bool; nodes : node -¿ bool; root : handle -¿ node; link : node -¿ num -¿ node; data : node -¿ data —¿‘;

val wellFormed_def = Define ‘wellFormed s = (IMAGE s.root s.handles SUBSET s.nodes / (!i n. n IN s.nodes ==¿ s.link n i IN s.nodes))‘;

val emptySmGraph_def = Define ‘emptySmGraph = ¡— handles := ; nodes := nil; root := λ h. nil; link := λ n i. nil; data := λ n. unspecifiedData —¿‘;

val connected_def = Define ‘connected start end s = ?k. end = s.link start k‘;

val reachableIn_def = Define ‘(reachableIn 0 s = (nil INSERT (IMAGE s.root s.handles))) / (reachableIn (SUC j) s = (reachableIn j s) UNION end— ?start. start IN reachableIn j s / connected start end s)‘;

val reachable_def = Define ‘reachable n s = ?j. n IN reachableIn j s‘;

val reachableNodes_def = Define ‘reachableNodes s = n—reachable n s‘;

val reachableSMG_def = Define ‘reachableSMG s = (s with ¡— nodes := reachableNodes s; link := λ n i. if reachable n s then s.link n i else nil; data := λ n. if reachable n s then s.data n else unspecifiedData —¿)‘;

Hol_datatype ‘handlePath = ¡— startHandle : handle; indexes : num list —¿‘;

Hol_datatype ‘nilPath = ¡— indexes : num list —¿‘;

val validHandlePath_def = Define ‘validHandlePath hp s = hp.startHandle IN s.handles‘;

val followIndexPath_def = Define ‘(followIndexPath [] n s = n) / (followIndexPath (hd::tl) n s = s.link (followIndexPath tl n s) hd)‘;

val indexPathStaysIn_def = Define ‘(indexPathStaysIn [] n s = n IN s.nodes) / (indexPathStaysIn (h::t) n s = indexPathStaysIn t n s / s.link (followIndexPath t n s) h IN s.nodes)‘;

val endOfNilPath_def = Define ‘endOfNilPath (np:nilPath) s = followIndexPath np.indexes nil s‘;

val endOfHandlePath_def = Define ‘endOfHandlePath hp s = followIndexPath hp.indexes (s.root hp.startHandle) s‘;

Hol_datatype ‘rootedPath = rootIsHandle of handlePath — rootIsNil of nilPath‘;

val validRootedPath_def = Define ‘(validRootedPath (rootIsHandle hp) s = validHandlePath hp s) / (validRootedPath (rootIsNil np) s = T)‘;

val endOfRootedPath_def = Define ‘(endOfRootedPath (rootIsHandle hp) s = endOfHandlePath hp s) / (endOfRootedPath (rootIsNil np) s = endOfNilPath np s)‘;

val homomorphism_def = Define ‘homomorphism hom g1 g2 = ((hom nil = nil) / (IMAGE hom g1.nodes SUBSET g2.nodes) / (!h. h IN g1.handles ==¿ (hom (g1.root

h) = g2.root h)) / (!n i. n IN g1.nodes ==¿ (hom (g1.link n i) = g2.link (hom n) i)) / (!n. n IN g1.nodes ==¿ (g1.data n = g2.data (hom n))))‘;

val homomorphic_def = Define ‘homomorphic g1 g2 = g1.handles SUBSET g2.handles / ?hom. homomorphism hom g1 g2‘;

val isomorphism_def = Define ‘isomorphism hom g1 g2 = (homomorphism hom g1 g2 / BIJ hom g1.nodes g2.nodes)‘;

val isomorphic_def = Define ‘isomorphic g1 g2 = (g1.handles = g2.handles) / ?iso. isomorphism iso g1 g2‘;

¿¿STORAGE MANAGER

A storage manager is a managed-memory-graph with the operations

smAddHandleAction_def smNewCellAction_def smSetLinkAction_def smGetLinkAction_def smSetDataAction_def

and the observer

smGetDataObserver_def

val smAddHandleAction_def = Define ‘smAddHandleAction h = ¡— specified := λ s. (h IN s.smg.handles); next := λ s. (s with smg := rawAddHandle h s.smg) —¿‘;

val smNewCellAction_def = Define ‘smNewCellAction h = ¡— specified := λ s. (h IN s.smg.handles); next := λ s. (s with smg := rawNewCell h s.smg) —¿‘;

val smSetLinkAction_def = Define ‘smSetLinkAction h1 i h2 = ¡— specified := λ s. (h1; h2 SUBSET s.smg.handles); next := λ s. (s with smg := rawSetLink h1 i h2 s.smg) —¿‘;

val smGetLinkAction_def = Define ‘smGetLinkAction h1 h2 i = ¡— specified := λ s. (h1; h2 SUBSET s.smg.handles); next := λ s. (s with smg := rawGetLink h1 h2 i s.smg) —¿‘;

val smSetDataAction_def = Define ‘smSetDataAction h d = ¡— specified := λ s. (h IN s.smg.handles); next := λ s. (s with smg := rawSetData h d s.smg) —¿‘;

and the observer

val smGetDataObserver_def = Define ‘smGetDataObserver h = ¡— specified := λ s. h IN s.smg.handles; value := λ s. rawGetData h s.smg —¿‘;

¿¿GARBAGE COLLECTOR

The garbage collector used in this thesis is almost identical to the storage manager. It has the same state, but the actions are .....

¿¿SIMULATION

Chapter 12

Appendix - Detailed Proofs

The actual proofs.