Memory Management: An Abstract Formulation of Incremental Tracing

Healfdene Goguen* Richard Brooksby† Rod Burstall‡

17 July 1999

Abstract

We present an abstract model of computer memory, in the form of directed, labeled graphs with simple transitions corresponding to valid memory operations. We also introduce a refinement of this basic model of memory that incorporates features of memory management. After giving some examples of how this refined model captures various incremental tracing algorithms, we investigate properties of the two representations and show that they are behaviorally equivalent. The proofs have been formally verified in the proof assistant Coq.

1 Introduction

1.1 Overview

Memory management is for several reasons an essential component of constructing modern large applications. First, space can be at a premium in large applications, and some mechanism is needed for returning unused space to the program efficiently. Secondly, memory management takes a significant portion of total program execution time, and naive implementations can lead to extremely inefficient programs. Finally, memory errors, such as accessing freed memory cells, in programs are notoriously difficult to find. Although sophisticated tools exist for uncovering various kinds of memory faults, it is safer to build more reliable memory management into the design. For these reasons, large programs need schemes to manage allocating and freeing of memory efficiently.

The view that a more disciplined approach to memory management is an important aspect of program development is filtering its way from the functional programming community into the broader programming community and more traditional state-based languages. For example, Java includes garbage collection as part of the language, and there are various packages for performing memory management in C and C++. With this increased importance, we believe that a formal understanding of memory management also becomes more important.

There are many schemes for memory management, ranging from explicit allocation and deallocation of memory to sophisticated generational and replication garbage collectors. We include references [6, 9] to several excellent introductions to the field of memory management.

1.2 Incremental Tracing

In this paper we concentrate on understanding a particularly error-prone area of garbage collection, incremental tracing. Tracing is the first of two phases in garbage collection, and consists of a systematic search through memory distinguishing those memory cells that are in use by the program from those that are not. The algorithms we are interested in are incremental because

*AT&T Labs, 180 Park Ave., Florham Park NJ 07932, USA. Most of this author's work was carried out at the LFCS, The King's Buildings, University of Edinburgh, EH9 3JZ, Scotland.

†Geodesic Systems, 414 North Orleans Street, Suite 410, Chicago, IL 60610, USA. This author was at Harlequin Ltd. at the time of the collaboration.

‡LFCS, The King's Buildings, University of Edinburgh, EH9 3JZ, Scotland.

the program can be using memory as the memory management system is searching for the unused cells or de-allocating those that have been found.

The tracing phase is excellent for formalization because there is a widely understood informal model for it, involving coloring the nodes of the graph black, grey or white. This informal understanding is remarkably general and can model the great majority of incremental (and non-incremental) tracing algorithms. Furthermore, verifying incremental tracing algorithms by hand is notoriously difficult, as demonstrated by errors in early papers such as that by Ben-Ari [1].

The second phase of garbage collection, the collection phase, consists of de-allocating those cells that are not in use by the program. We do not concentrate on the collection phase in this paper because, unlike the tracing phase, there is no unified understanding of this phase. We leave a general treatment of collection, under development now, to a further paper.

1.3 A General Theory of Memory Management

As far as we are aware there is no general theory of incremental tracing, let alone memory management, in the literature. There have been some formal treatments of garbage collection [4, 5, 7, 8], but these do not offer a general framework for understanding the problem. We see our paper primarily as a move toward an abstract understanding of memory management.

The main contribution of this paper is the presentation of abstract formulations of computer memory and incremental tracing over computer memory, and a proof of their behavioral equivalence. This model is sufficiently abstract that it is easy to formalize. It also has a clear correspondence with incremental tracing algorithms, and so can be used as a vehicle for communication between academic and industrial collaborators.

All definitions and proofs discussed in this paper have been carried out in the proof assistant Coq [3], a type-theory based system developed at INRIA. In this paper we have deliberately avoided any discussion of issues involving the underlying logic of the proof assistant, preferring to present the results in a naive set-theoretic style, because we believe that the abstractions are the most important contribution of this work.

2 A Model of Memory

A naive model of memory on a computer might include a CPU with registers, and various external forms of memory such as random-access memory, disks, and tapes. The CPU accesses memory other than its registers by using particular instructions with offsets.

Starting from this simple view of memory, we abstract to a model of memory as a directed, labeled graph with roots, with certain simple restrictions. The roots correspond to registers, and the labels correspond to offsets to access external memory.

2.1 Pre-Graphs and Graphs

We begin by introducing the notion of pre-graph, which simply characterizes the raw data of a graph, without guaranteeing its well-formedness or consistency. Intuitively the pre-graph represents the memory cells as they exist physically on the computer, without reflecting the structure built up when a program is running.

We parameterize our definitions by a set N of *pre-nodes* and a set L of *labels*. The pre-nodes represent memory cells, and the labels represent pointers or offsets to memory cells.

Definition 2.1 (Directed Labeled Pre-Graph with Roots) *A* directed labeled pre-graph with roots *has two subsets* Node *and* Root *of* N *and a subset* Edge *of* $N \times L \times N$.

We shall simply write pre-graph in place of directed labeled pre-graph with roots.

We also introduce a predicate *GraphP* characterizing those pre-graphs that are actually graphs. This predicate says that a well-behaved graph G must satisfy the following axioms:

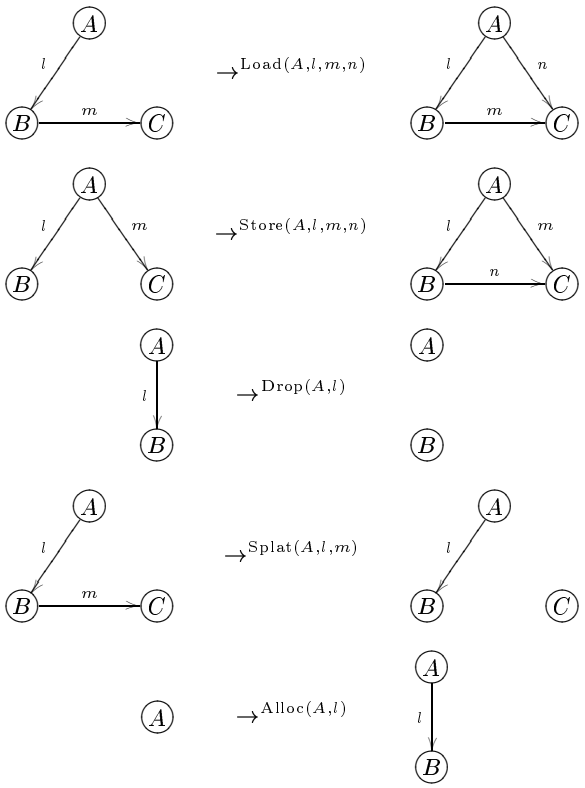
- Every root is a node.
- The endpoints of every edge are nodes.
- If there are two edges from a by label l, with endpoints b and c, then $b = c$.
- Any graph G always has an infinite number of pre-nodes that are not nodes in G.

We say that G is a graph if GraphP(G).

We use a simple graphical notation for our pre-graphs, which are easily translated into statements about the formal objects.

2.2 Memory Actions on Graphs

We can consider basic operations over this simple model of memory. All of these operations start from the roots, or the CPU, and affect the external memory in some way.



Here, all actions require that A be a root, and the last action is subject to the requirement that B is not a node in the original graph. We write $G \to^{\alpha} H$ if one of the above actions on the graph G produces the graph H.

The model we have introduced is a purely local presentation of memory. We do not require any global properties about the accessibility of nodes in the graph, nor do we require operations on the graph more than two indirections away from the root. Complex operations can be implemented as sequences of the basic operations we have introduced.

The operations introduced correspond closely to the usual memory operations available in a computer. For example, the Load operation corresponds to bringing a pointer stored in external memory into the registers on the CPU. The Store operation corresponds to placing a pointer stored in the CPU into external memory. The Drop and Splat operations model clearing pointers, either in the CPU or in external memory. The Alloc action creates a new node in the graph and links it by the label l to root A.

There are many possible restrictions of our formulation that would remain interesting. For example, in LISP we only have two possible labels for edges, *car* and *cdr*. We have not investigated such restrictions in our work so far. Memory management for Java is more complex, but we believe it is still within our framework.

The following proposition is proved by induction on possible actions:

Proposition 2.2 *If* $G \rightarrow^{\alpha} H$ *and* GraphP(G) *then* GraphP(H).

2.3 Accessibility

We have two notions of accessibility of nodes in a memory graph. First, those nodes in the transitive closure of the edge relation from a root are accessible. Secondly, nodes that can be brought next to the root by a sequence of memory actions are accessible. We show that these two notions of accessibility are equivalent.

We begin by defining inductively these notions of when a node is accessible.

Definition 2.3 (Graph Accessible)

- *A root a in a graph G is graph accessible in G.*
- *If a is graph accessible in G and there is an edge l from a to b in G then b is graph accessible in G.*

Definition 2.4 (Action Accessible)

- *A root a in a graph G is action accessible in G.*
- *If a is a root in a graph G and there is an edge from a to b then b is action accessible in G.*
- *If a is a node in the graph G and is action accessible in H, and $G \rightarrow^{\alpha} H$, then a is action accessible in G .*

We now show that action accessibility implies graph accessibility. We begin by showing that memory actions do not extend the graph accessibility relation. Clearly, this will not apply to newly allocated nodes, so we restrict to those nodes that are in the original graph.

Lemma 2.5 *If G is a graph, d is a node in G, d is graph accessible in H and $G \rightarrow^{\alpha} H$ then d is graph accessible in G.*

Proof By induction on possible actions $G \rightarrow^{\alpha} H$. In each case, we then apply induction on the proof that a is graph accessible in H.

We consider the case Load(A, l, m, n). Suppose we have added an edge n from d to c. Then if a proof of accessibility uses n then we instead use edges l and m, and otherwise if we use an edge p that is not n then p is also in G.

Proposition 2.6 *If a is action accessible in G then it is graph accessible in G.*

Proof By induction on a being action accessible in G. The cases of a being a root or one edge away from the root are trivial, and the inductive case follows by Lemma 2.5.

We now show that all accessible nodes in the memory graph can be reached by a sequence of memory actions. The proof relies on there being two labels, l and m, that are not used out of the root r from which a is accessible.

Proposition 2.7 *If a is graph accessible in G then it is action accessible in G.*

Proof The proof proceeds by induction on a being graph accessible in G. If a is a root or if there is a root r and an edge n from r to a then a is immediately action accessible in G. Otherwise, we are in the case that there is a node b accessible from the root r and an edge n from b to a. For a path of length 2 from the root r to a we add an edge l from r to a using a Load action, and from then on we alternate pairs of actions $\mathrm{Load}(r, l, n, m)$ and $\mathrm{Drop}(r, l)$ with pairs of actions $\mathrm{Load}(r, m, n, l)$ and $\mathrm{Drop}(r, m)$. This yields a sequence of actions that finishes with a graph with the edge l or m linking r to a.

From now on, we shall simply say accessibility for graph accessibility, since the two definitions are equivalent and graph accessibility is the more direct notion.

3 Garbage Collection by Incremental Tracing

The model of memory introduced in the previous section does not take account of the bookkeeping underlying a program's use of memory. Normally, a program will allocate new blocks of memory, use them for some amount of time, and then free the memory, either explicitly by telling a memory management system or implicitly by having removed all pointers to the memory.

In interactive applications the user should not be able to notice interventions by the memory management system. Hence, the program and the memory management system should be interleaved, with the memory management system able to do work at any point while the program is running. Because the program is active changing the memory while the memory management system is trying to determine inaccessible pointers, the program is commonly called the mutator.

One common technique to allow this is incremental tracing. Abstractly, the memory management system traverses the graph from the registers, marking memory cells as it passes through them, and keeping enough information that changes in the graph by the program do not confuse it. Once there are no more nodes to traverse in the graph of active memory, the rest of memory is returned to the program.

We formalize this notion of incremental tracing using a definition of colored graphs over the colors *White*, *Grey* and *Black*. Intuitively, nodes are white if they have not been considered, marking a node grey means that it has been considered but its children have not, and marking a node black means that the node has been completely traversed by the tracing algorithm.

This approach of considering that the tracing algorithm assigns colors to the objects in the graph in its traversal is well-established in the memory management community, and has existed at least since the seminal paper by Dijkstra et al. [2].

3.1 Colored Graphs

We now introduce the definitions for colored graphs. This is the formal counterpart of graphs for the development of memory management systems with incremental tracing.

Definition 3.1 (Colored Pre-Graph) *A* colored pre-graph *is a pre-graph with the further set* $\mathit{Color} \subseteq N \times \{\mathrm{White}, \mathrm{Grey}, \mathrm{Black}\}$.

The set *Color* in colored pre-graphs captures the possibility that the data structure representing colored pre-graphs may allow a node to have several colors if the internal representation is inconsistent and does not represent a colored graph.

A colored pre-graph satisfies the predicate *ColorP* if it satisfies the following conditions:

- It satisfies the predicate *GraphP*.
- The set *Color* is functional, that is for all $a \in N$ there is a unique c in $\{\mathrm{White}, \mathrm{Grey}, \mathrm{Black}\}$ such that $(a, c) \in \mathit{Color}$.
- There is a finite number of nodes that are white or grey.

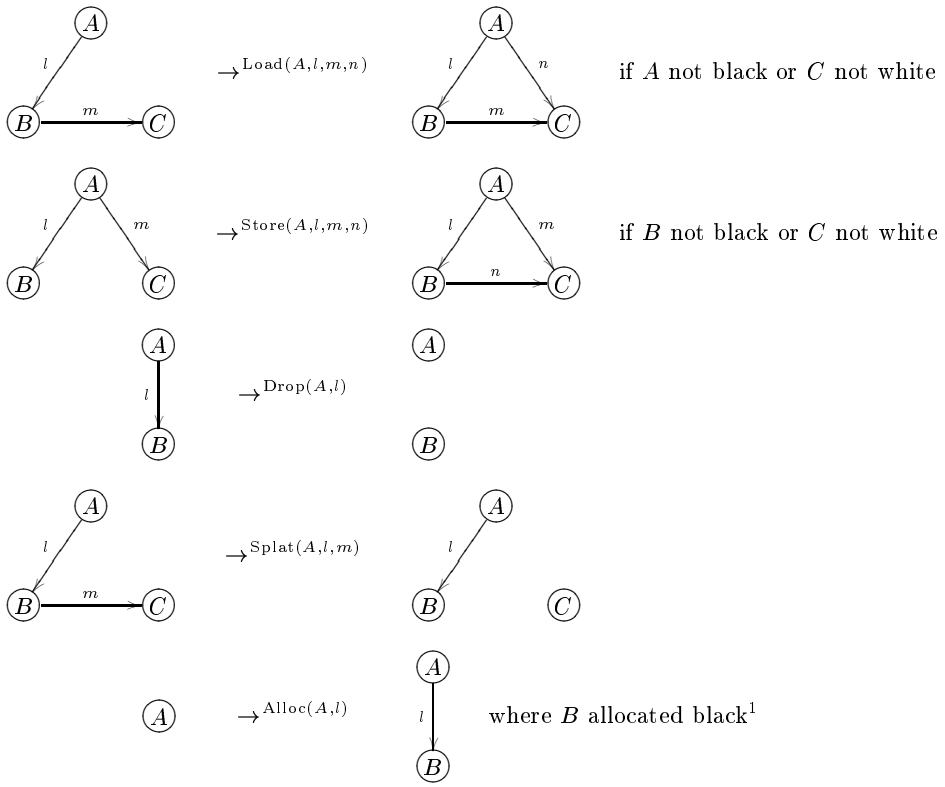
Similar to pre-graphs, we say that C is a colored graph if it satisfies the predicate *ColorP*.

We write $|C|$ for the underlying pre-graph associated with C. Notice that if C is a colored graph then $|C|$ is a graph.

We do not require colored graphs to have a finite number of nodes in order to be able to do tracing on it. We only require that the number of white and grey nodes be finite, since this represents the portion of the graph that is actually being traversed. This allows us to consider infinite graphs, or effectively infinite graphs where we do not have control over the entire graph, like the Internet.

3.2 Memory Actions over Colored Graphs

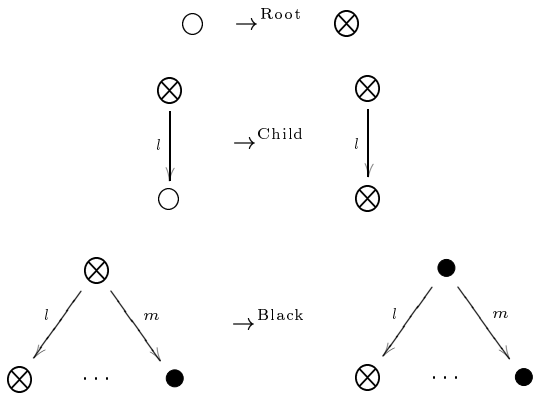
We can now formulate the actions of the mutator in the setting of colored pre-graphs. We must maintain the invariant that there are no links from a black node to a white one, which informally models our intuition that the grey fringe represents the nodes that are in the process of being considered. To preserve the invariant, there are restrictions on the operations for loading and storing memory addresses.



Again, similar to graph actions, A is a root, and the last action is subject to the requirement that B is not a node in the original graph.

We also have actions representing the possible work the tracer can do. These operations merely manipulate the colors of the graph, and do not alter the structure of the graph in any way.

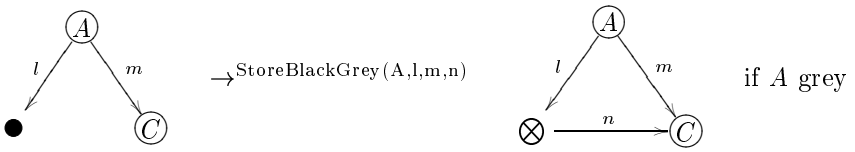
[1] This rule could be generalized so that if A is not white then B can be allocated grey or white, but we have not proved anything about such a rule.



In the rule Root, the node must be a root.

The mutator actions over colored graphs are carefully formulated so as not to change colors of objects themselves. We consider that the tracer is solely responsible for changing colors in the graph, which may correspond to copying objects and updating pointers, or updating data kept in an internal representation of the tracer.

The approach of separating tracer actions from memory actions, and restricting memory actions so that they can only occur when the resulting graph will not violate the invariant, is very general. However, there is one situation that is not captured by this abstraction. If we are storing a pointer to an object in a black node then for efficiency reasons it is often desirable in practice to turn the black node grey directly. However, for the tracer this represents a loss of progress. We therefore include this as a single mutator action, *StoreBlackGrey*, which when storing into a black object with a grey root will mark the black object grey and do the store:



if A grey

subject to A being a root. This allows the unobservable tracer actions to be terminating at any point, even if the mutator can create more work for the tracer through an observable action.

We now discuss how two common incremental tracing algorithms fit the abstraction of actions over colored graphs.

3.3 Mark–Sweep Collectors

Mark–sweep collectors traverse the graph structure as the mutator sees it, finding the transitive closure of the graph and marking each object that is in the graph in some way to record that the object was reached. The objects that are currently being considered are stored in a stack or a queue. The algorithm adds the unmarked children of the object at the head of the queue, and then removes the head from the queue and marks it. Once the queue is empty, and therefore the connected objects have been determined, the collector examines memory to find those objects that have not been marked and reclaims the space they occupy.

The color abstraction captures this algorithm naturally. Objects that have not been marked are white. Objects that are in the queue of objects still being considered are grey. Objects that have been marked are black.

In this framework, the roots are considered grey throughout tracing, only being changed to black when the graph has been fully traversed. This means we only need to prevent pointers to white objects being written into black objects, which is called a write barrier.

3.4 Copy Collectors

A copy collector is implemented by using two equal sized pools of memory, the *to-space* and the *from-space*. A pointer in to-space indicates where to allocate new objects, and is incremented with each allocation. When this pointer reaches the end of to-space, indicating that there is no further memory, the to- and from-spaces are reversed or flipped, and objects in the new from-space that are accessible from the root are copied to the new to-space. Objects copied from from-space to to-space are scanned to find pointers to other objects in from-space, and those objects are also copied to to-space: a pointer in from-space indicates the last object whose children have been considered.

Conceptually, the black objects are those that have been copied from from-space to to-space above the pointer, whose children have also all been considered. The grey objects are those copied and below the pointer in from-space, whose children have still not been considered. The white objects are those that have not been copied from from-space to to-space.

Attempts to access an object in from-space must be trapped and the object copied to to-space, because it is accessible. This is called a read barrier.

3.5 Condemned Zones

A common technique in memory management systems is to select a particular area of memory for garbage collection, sometimes referred to as the *condemned zone*. In this setting all objects outside of the condemned zone are considered black, and those objects within the condemned zone that have pointers to them from outside are considered to be the roots, and hence grey. This gives us a technique for garbage collecting a finite subgraph of infinite or effectively infinite graphs, such as the Internet, because only finitely many nodes need to be traversed by the algorithm.

3.6 Formal Properties

The tracer actions are non-deterministic. A graph may have several possible actions that the tracer can reasonably perform to make progress towards determining the transitive closure of accessibility in the graph. This non-determinism is intended to represent choice of implementation, rather than choice of action for a particular tracer. However, we can show that this non-determinism of the tracer always gives the same results, so choice of implementation does not affect behavior.

Lemma 3.2 *If* $C \rightarrow^{\alpha} D$ *and* $C \rightarrow^{\beta} D'$ *then there is an* E *such that* $D \rightarrow^{\gamma} E$ *and* $D' \rightarrow^{\delta} E$.

Proof By a double induction on actions $C \rightarrow^{\alpha} D$ and $C \rightarrow^{\beta} D'$. There are two critical pairs, between *Root* and *Child*, and *Black* and *Child*, and both are easy.

Lemma 3.3 *If* $C \rightarrow^{\alpha} D$ *and* C *satisfies* ColorP *then* D *satisfies* ColorP.

Proof By induction on actions $C \rightarrow^{\alpha} D$.

We also have the following important property, that tracer actions preserve the invariant.

Lemma 3.4 (Preservation of Invariant) *If the colored graph* C *satisfies the invariant and* $C \rightarrow^{\alpha} D$ *then* D *satisfies the invariant.*

Proof By induction on actions $C \rightarrow^{\alpha} D$.

From now on we implicitly assume that all colored graphs satisfy the invariant.

4 Correctness of Incremental Tracing

4.1 Behavioral Equivalence

Now that we have defined both a model of ideal memory and a model of incrementally traced memory, we wish to show that the two have the same behavior with respect to program operations on

memory. We do this using the notion of bisimilarity from concurrency to model behavioral equivalence, where we consider tracer actions in the memory management model to be unobservable. In concurrency theory unobservable actions are denoted by writing them τ.

The behavioral equivalence with respect to program operations is the central result of this paper. It is a formal result that captures various of our intuitions about incremental tracing. We know that although tracing may postpone mutator actions, in order to avoid creating a pointer from a black object to a white one, it will never prevent mutator actions indefinitely. We also know that memory management does not change an observer's view of how memory works.

We write $G \Longrightarrow^{\alpha} H$ if $G \rightarrow^{\alpha} G' \rightarrow^{\tau\star} H$.

Definition 4.1 (Bisimulation) *Given two relations R and S on graphs, a relation T between two graphs G and H is a bisimulation if:*

- *If $G \rightarrow^{\alpha}_{R} G'$ then there is an H' such that $H \Longrightarrow^{\alpha}_{S} H'$ and $G'TH'$.*
- *If $H \rightarrow^{\alpha}_{S} H'$ then there is a G' such that $G \Longrightarrow^{\alpha}_{R} G'$ and $G'TH'$.*

Bisimulation, $G \approx_{\rightarrow_R,\rightarrow_S} H$, is the greatest relation that is a bisimulation.

Notice the parametrization of bisimilarity by relations on graphs. This is necessary to allow us to talk about bisimilarity between a graph and a colored graph. In this case, we would take R to be the memory actions on graphs, and S to be the memory actions on colored graphs, with τ-actions being the tracer actions. We omit the subscripts R and S when they are clear from the context.

We begin by demonstrating that colored graphs with the tracer and mutator actions are behaviorally equivalent to their underlying graph with the simple memory actions. The actual proof is not complicated given the results we have already established.

Lemma 4.2 *If T is a bisimulation and CTG then $C \approx G$.*

Proof By a simple coinduction.

We now show that the relation $|C| = G$ is a bisimulation for $\rightarrow_C$ and $\rightarrow_G$, and so $C \approx_{\rightarrow_C,\rightarrow_G} |C|$.

Lemma 4.3 *If $|C| = G$ and $C \rightarrow^{\alpha} D$ then there is an H such that $G \rightarrow^{\alpha} H$ and $|D| = H$.*

Proof By a simple induction on actions $C \rightarrow^{\alpha} D$ we can show that $|C| \rightarrow^{\alpha} |D|$, so we take H to be $|D|$.

Lemma 4.4 *If $|C| = G$ and $G \rightarrow^{\alpha} H$ then there is a D such that $C \rightarrow^{\alpha} D$ and $|D| = H$.*

Proof By induction on actions $G \rightarrow^{\alpha} H$. The difficult cases are Load, Store and Alloc. We consider Load.

We know that a is a root, that there is an edge l from a to b and an edge m from b to c, and that there is no node d such that there is an edge n from a to d.

We consider the possible colors of a. If it is white or grey then we can simply perform a Load, because the side-condition is satisfied. If a is black then we need to consider the possible colors of c, where if c is black or grey then we can perform a Load. Finally, if c is white we need to consider the colors of b. b cannot be white or black because this contradicts the invariant. Hence b is grey, and we can perform a Load.

Lemma 4.5 *$C \approx_{\rightarrow_C,\rightarrow_G} |C|$ for all colored graphs C.*

Proof $|C| = G$ is a bisimulation by Lemmas 4.3 and 4.4, and so $C \approx_{\rightarrow_C,\rightarrow_G} |C|$ by Lemma 4.2.

4.2 Termination

We now show that tracer actions terminate, which guarantees that there cannot be an infinite sequence of τ-actions preventing an observable action.

Lemma 4.6 *For all color graphs C, the τ-actions starting from C terminate.*

Proof By double induction on the number of white and grey nodes in C.

Suppose $C \rightarrow^{\tau} C'$. By case analysis on this we show that C' is terminating, and so by definition of termination C is terminating.

- Root. Then the number of white nodes in C' is less than those in C, so by the induction hypothesis C' is terminating.
- Child. Again the number of white nodes in C' is less than that in C.
- Black. Then the number of grey nodes in C' is less than those in C, and the number of white nodes stays constant, so by the induction hypothesis C' is terminating.

5 Conclusions and Further Work

We have introduced a new formal model of memory that captures the basic operations simply and cleanly. We have extended this model to incremental tracing using the common intuition of colors from the memory management community, and showed that the two models of memory are behaviorally equivalent. Furthermore, we have established that bisimilarity is an equivalence and a congruence for the natural operations on graphs. This work lays the foundation for further research into verifying implemented algorithms for incremental tracing.

There are many areas for further work. The most important is the study of incremental collection in addition to incremental tracing. There is an intuitive action for collecting that is simple to formalize. In the ideal representation of memory, if a graph G has an inaccessible node a then G should be able to transition unobservably to the graph G where the node a is removed. In the colored graph representation, if all of the roots in a colored graph C are black, there are no grey nodes in C and there is a white object a then C should be able to transition unobservably to the graph C where the node a has been removed.

However, the more complex notion to capture in this framework is the resetting of the collector. Different implementations will lead to dramatically different ways of coloring the graph on reset. For example, if the whole of memory is being collected by a copy collector then all nodes can be colored white on reset, but if we use condemned zones then only a restricted subset of the nodes will be colored white, with the majority remaining black and a few being colored grey. It is because of this wide variation of possible resetting behaviors that we have not yet formalized the collection phase.

Another important area that we have left unexplored is the actual verification that a particular implementation of memory and memory management satisfy the axioms we have set out. This seems to be an interesting project that could be proposed as a Master's thesis.

A direction for further work is to investigate *similarity* for the tracer actions, which says that a collection of actions over the colored graphs is a restriction of the tracer actions. This notion can capture refinement or implementation of the full class of possible tracer actions by a more particular algorithm.

Acknowledgments

This paper is an overview of work on memory management by the LEGO project, including Rod Burstall, Adriana Compagnoni, Healfdene Goguen, Paul Jackson, Thomas Kleymann and John Longley, together with Richard Brooksby and David Jones at Harlequin Ltd. Adriana Compagnoni provided important encouragement in the preparation of the paper.

This work was partially financed by the British funding body EPSRC. Harlequin Ltd. also provided important financial support, including partial support of one of the authors.

The diagrams in this paper were formatted using Kristopher Rose's XY-pic package.

References

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

[2] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the Association for Computing Machinery*, 21(11):966–975, November 1978.

[3] Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA-Rocquencourt, Feb. 1993.

[4] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. volume 1102 of *Lecture Notes in Computer Science*. Springer, July 1996.

[5] P. Jackson. Verifying a garbage collection algorithm. In *Proceedings of TPHOLs*, 1998.

[6] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic memory Management*. John Wiley & Sons, 1996.

[7] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995. ACM.

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

[9] P. R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*. Accepted, in revision.