Abstract

The Mechanized Verification of Garbage Collector Implementations

Andrew Evan McCreight

2008

Languages such as Java, C♯, Standard ML, and Haskell use automated memory management to simplify development and improve reliability by eliminating the need for programmers to manually free unused objects. The cost of this improved programmer experience is that the implementation of the language becomes more complex, requiring a garbage collector. Garbage collectors are becoming increasingly sophisticated to adapt them to high-performance, concurrent and real-time applications, making internal collector invariants and the interface with user programs (*mutators* in garbage collector parlance) subtle and difficult to implement correctly.

My thesis is that treating the garbage collected heap as an abstract data type allows the specification and verification of flexible and expressive garbage collector-mutator interfaces, and that the mechanized verification of garbage collector implementations using Hoare-style logic is both practical and effective. The mutator-garbage collector interface I describe in this paper is expressive enough to be used with verified mutators, while being abstract enough that a single interface can be used with both stop-the-world and incremental collectors. It is also powerful enough to be used with collectors that require read or write barriers.

In this dissertation, I describe my framework for the mechanized verification of garbage collector implementations. I discuss the formal setting, and my approach to garbage collector interfaces. I also describe the specification and verification of the Cheney and Baker copying collectors. The Baker collector is an incremental collector

that requires a read barrier. Finally, I discuss some practical tools I developed for program verification using separation logic in the Coq proof assistant.

The Mechanized Verification of Garbage Collector Implementations

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Andrew Evan McCreight

Dissertation Director: Zhong Shao

December 2008

Contents

List of Figures

Acknowledgments

I would like to thank my advisor Zhong Shao for all of his guidance throughout my graduate career. From my very first year, he was always available to instruct, advise and challenge me. He was never short of energy or ideas, and my work is much stronger because of our collaboration.

I would like to thank my thesis readers, Neal Glew, Paul Hudak and Carsten Schürmann for reading my dissertation and providing thoughtful comments. I am also grateful to the anonymous reviewers of my submitted papers for their comments that improved my work.

I would also like to thank Chunxiao Lin and Long Li for helping to test the garbage collector interface, in part by verifying mark-sweep collectors and example mutators. Their work helped ensure that the interface could actually be used.

I am indebted to Xinyu Feng, Hai Fang and the other members of the Yale FLINT group for many interesting and informative discussions over the years.

I thank my parents for their love and support. Finally, I thank Melanie, whose love and encouragement inspired me to do my best.

[0]This research is based on work supported in part by the National Science Foundation under grants CCR-0208618 and CCR-0524545 as well as gifts from Intel and Microsoft Research. Any opinions, findings, and conclusions contained in this document are those of the author and do not reflect the views of these agencies.

Chapter 1

Introduction

Software is becoming larger and more difficult to develop. One particular difficulty of development is memory management: allocating new objects when they are needed and freeing them when they are not. One way to simplify the problem of freeing objects is to use a *garbage collector* [Jones and Lins 1996]. A garbage collector (GC) is a procedure, usually part of a programming language's runtime system, that examines memory to find and reclaim objects that are unreachable by the user program (the *mutator*). This eliminates many sources of errors, such as dangling pointers (using a pointer after freeing it) and some types of space leaks (not freeing a pointer before it becomes unreachable).

Functional languages such as Common LISP [Graham 1996], Standard ML [Milner et al. 1997] and Haskell [Jones 2003] have long used garbage collection, and over the last decade the combination of convenience and improved reliability has led to the use of garbage collection in popular languages such as Java [Gosling et al. 2005] and C♯ [Hejlsberg et al. 2006]. It is even possible to use specially designed *conservative* garbage collectors [Boehm and Weiser 1988] with languages such as C [Kernighan and Ritchie 1978] and C++ [Stroustrup 2000] that were not originally designed to

be garbage collected.

There are also several high-confidence systems that use type systems or logic to improve the reliability of programs that rely on a garbage collector to simplify reasoning about the user program. These include proof-carrying code for C and Java [Necula and Lee 1998, Colby et al. 2000] and typed assembly language [Morrisett et al. 1999], as well as more recent efforts such as Microsoft Research's Singularity project (which is implementing as much of an operating system as possible in C♯) and a semantics-preserving compiler from a functional language to assembly [Chlipala 2007]. What all of this work has in common, besides a reduced set of trusted components, is that the safety guarantees of the entire system critically depend on the correct implementation of a garbage collector that cannot be checked within the system. A verified garbage collector capable of being combined with a verified mutator would go a long way towards addressing this "missing link" of safety.

While garbage collection makes it simpler to *use* a language, it makes it harder to *implement* a language, as a GC is now required. Even a basic garbage collector is difficult to implement correctly, and few implementations use basic GCs. In fact, garbage collectors are becoming more sophisticated as they are used more widely, increasing performance demands. Garbage collectors are also being applied to entirely new domains, such as concurrent and real-time environments, which have their own special requirements [Doligez and Gonthier 1994, Bacon et al. 2003]. All of this conspires to create a situation where internal collector invariants, as well as the interface between the mutator and collector, are subtle and difficult to implement and debug. For instance, at a very basic level, the mutator and collector must agree on what constitutes a reachable object. Also, in some collectors, the mutator must access objects only by special collector-provided functions to maintain the integrity of the system.

How can garbage collector reliability be assured despite these demands? One way to is to use *static verification*, whereby properties of a program are established by examining the program and not by running it. There are a variety of approaches to the static verification (henceforth referred to simply as *verification*) of programs, such as type systems [Pierce 2002], Hoare logic [Hoare 1969] and separation logic [Reynolds 2002] (which can be thought of as a Hoare logic for reasoning about complex memory invariants). If some property of a program is statically verified, then that property holds on the program, as long as the verification system is sound. This increases confidence in the program at the cost of increasing the work of the implementer. I believe that this trade-off is worthwhile for garbage collectors, as the verification effort can be amortized across every program that uses the collector. Confidence can be further increased by *mechanizing* the verification, which means that the result of verification can be checked by a machine, in contrast to a proof that exists only on paper.

My thesis has two parts. The first part of my thesis is that treating the garbage collected heap as an abstract data type allows the specification and verification of flexible and expressive garbage collector-mutator interfaces. I will demonstrate this by first describing how to define a GC interface in terms of an abstract data type (ADT), in Chapter 4, and show how this interface can be used by a verified mutator. Then, in Chapter 5 I will show how this interface can be used to describe the heaps of a variety of collectors. As part of Chapters 6 and 7 I will show how two collectors can be verified to match my ADT-based interface. The interface I will describe is expressive enough to be used with verified mutators, general enough to be used with collectors requiring read or write barriers, and abstract enough that a single interface can be used for both stop-the-world and incremental collectors. I do not attempt to support all of the features needed by a production-quality garbage collector such as

a realistic root set.

The second part of my thesis is that the mechanized verification of garbage collector implementations using Hoare-like logic is both practical and effective. I will demonstrate the practicality of mechanized verification of garbage collectors by describing the specification and verification of implementations of the Cheney [Cheney 1970] and Baker [Baker 1978] copying collectors in Chapters 6 and 7. The Baker collector is an incremental copying collector. My work is, as far as I am aware, the first mechanized verification of the Baker collector. I will also discuss the tools I developed to aid my verification in Chapter 8. I will demonstrate the effectiveness of my approach by showing how to link these collectors with verified mutator programs, resulting in a closed, verified and garbage collected program.

In addition to the above chapters, in Chapter 2 I will discuss the formal setting I am working in. This includes the abstract machine I use, the program logic SCAP [Feng et al. 2006] (which stands for Stack-based Certified Assembly Programming) I use to verify programs (along with a variant of SCAP named WeakSCAP), and separation logic [Reynolds 2002], which I use to define the complex memory predicates needed for garbage collectors. In Chapter 3, I will discuss how I use the module system of Coq to reason about abstract data types in a Hoare logic setting. In Chapter 9 I will discuss related work, future work, and conclude.

The full Coq proof scripts for my dissertation are available online [McCreight 2008]. The research regarding garbage collector interfaces described in this dissertation was first published in McCreight et al. [2007]. The work described in this dissertation is applied to a typed assembly language combined with a conservative GC in Lin et al. [2007]. Chunxiao Lin and Long Li proved some of the machine-related lemmas used in my collector proofs, and verified mutators using the interface, which lead to the addition of some basic properties of the representation predicate.

Chapter 2

Formal Setting

2.1 Introduction

In this chapter, I discuss the formal foundation of my work. To create a machine-checked verification of a garbage collector, every detail of the machine used to implement the collector and the logics used to reason about the collector must be formalized. First, I discuss the abstract machine I use, which is a fairly standard assembly-level machine with a small-step semantics. Then, I discuss SCAP [Feng et al. 2006] (pronounced "ess-kap"), the Hoare-style program logic I use to verify programs. Raw SCAP is difficult to work with, because it requires that every program point is given an explicit specification. To avoid this problem I define a new variant of SCAP, called WeakSCAP, that automatically derives a specification for each program block.

An expressive logic is needed to reason about the complex memory invariants of garbage collectors. To this end, I use separation logic [Reynolds 2002]. I define the various predicates of separation logic that I use, and discuss their properties, both in the abstract and relative to the machine.

After that, I discuss the particular idiom I use for program specifications throughout my verification work. Then I apply what I have discussed in this chapter to perform an example verification, then conclude.

2.2 The machine

Here I define the syntax and semantics of the underlying formal machine I use to implement and verify programs. The formal machine is assembly level, and uses word-aligned addresses. Address alignment is modeled to allow the lowest bit to be used to distinguish pointers from non-pointers. The word size of the machine is 4, so addresses are all multiples of 4. The most unrealistic aspect of this machine is that it does not use fixed precision integer arithmetic. Instead, register and memory values are all natural numbers, and arithmetic operations are all the normal operations on the naturals. Overall, this is a fairly standard machine, with a small-step semantics.

This description is part of the *trusted computing base* of my system. This means that there is no way of checking, within the system, that this definition is correct. I will give the syntax and dynamic semantics of the machine.

2.2.1 Syntax

The machine is based on MIPS-like assembly code [Sweetman 2006]. I give the syntax for the machine in Figure 2.1. A register r is one of 32 different registers from r0 to r31. r0 always has the value 0. Natural numbers are given by w and f, and are used to indicate numerical data and function pointers, respectively. Addresses l are used for data pointers, and are multiples of 4.

A *command* c is a non-control flow instruction, and is either the addition of two registers, the addition of a register and a constant, the subtraction of two registers,

$$
\begin{array}{lll}
(\mathit{Reg}) & r & ::= \{\mathsf{r}k\}^{k\in\{0\ldots31\}} \\
(\mathit{Nat}) & w, f & ::= 0 \mid 1 \mid 2 \mid \ldots \\
(\mathit{Addr}) & l & ::= 0 \mid 4 \mid 8 \mid \ldots \\
(\mathit{Command}) & c & ::= \mathsf{addu}\ r_d, r_s, r_t \mid \mathsf{addiu}\ r_d, r_s, w \\
 & & \quad \mid \mathsf{subu}\ r_d, r_s, r_t \mid \mathsf{sltu}\ r_d, r_s, r_t \mid \mathsf{andi}\ r_d, r_s, 1 \\
 & & \quad \mid \mathsf{lw}\ r_d, w(r_s) \mid \mathsf{sw}\ r_s, w(r_d) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= c; \mathbb{I} \mid \mathsf{beq}\ r_s, r_t, f; \mathbb{I} \mid \mathsf{bne}\ r_s, r_t, f; \mathbb{I} \\
 & & \quad \mid \mathsf{j}\ f \mid \mathsf{jal}\ f, f_{ret} \mid \mathsf{jr}\ r_s \\
(\mathit{Memory}) & \mathbb{M} & ::= \{l \rightsquigarrow w\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{r \rightsquigarrow w\}^* \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{M}, \mathbb{R}) \\
(\mathit{CodeMem}) & \mathbb{C} & ::= \{f \rightsquigarrow \mathbb{I}\}^* \\
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C}, \mathbb{S}, \mathbb{I})
\end{array}
$$

Figure 2.1: Machine syntax

the Boolean calculation of whether one register is less than another or not, the bitwise 'and' of a register with 1, a load or a store. I only allow bitwise 'and' with 1 because that is all that is needed to implement the garbage collectors I will verify, and this specialized case is easier to implement. In the load and store operations, the constant is an offset from the base pointer.

An instruction sequence $\mathbb{I}$ is either a command or a branch followed by an instruction sequence, a tail call to a function f, a call to a function f with a return pointer of f_{ret}, or a jump to a function pointer held in a register. The explicit return pointer for calls is non-standard, but simplifies the semantics. This instruction could be implemented with a standard call that falls through to the next instruction upon return, by following it with a direct jump to the return pointer.

A *memory* $\mathbb{M}$ is a finite partial mapping from addresses to natural numbers, while a *register file* $\mathbb{R}$ is a total mapping from registers to natural numbers. The *data state* $\mathbb{S}$ is a memory plus a register file. A *code memory* $\mathbb{C}$ is a partial mapping from natural numbers to instruction sequences. Finally, a *program* $\mathbb{P}$ is a code memory, a data state and an instruction sequence. The instruction sequence in a program

if $c =$	then $\text{Next}_c(\mathbb{M}, \mathbb{R}) =$	
addu r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + \mathbb{R}(r_t)\})$	
addiu r_d, r_s, w	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) + w\})$	
subu r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s) - \mathbb{R}(r_t)\})$	
sltu r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 1\})$	if $\mathbb{R}(r_s) < \mathbb{R}(r_t)$
sltu r_d, r_s, r_t	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 0\})$	if $\mathbb{R}(r_s) \geq \mathbb{R}(r_t)$
andi $r_d, r_s, 1$	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 1\})$	if $\mathbb{R}(r_s)$ is odd
andi $r_d, r_s, 1$	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow 0\})$	if $\mathbb{R}(r_s)$ is even
lw $r_d, w(r_s)$	$(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{M}(\mathbb{R}(r_s) + w)\})$	if $\mathbb{R}(r_s)+w \in \text{dom}(\mathbb{M})$
sw $r_s, w(r_d)$	$(\mathbb{M}\{\mathbb{R}(r_d)+w \rightsquigarrow \mathbb{R}(r_s)\}, \mathbb{R})$	if $\mathbb{R}(r_d)+w \in \text{dom}(\mathbb{M})$

Figure 2.2: Command semantics

is the current instruction sequence being executed, and serves the same function as a program counter in a more conventional machine. This abstraction does not affect anything, as the garbage collections I will be reasoning about do not explicitly manipulate the program counter.

I write $X(y)$ for the binding of y in the map X. This is undefined if y is not in the domain of X. For register files $\mathbb{R}$, $\mathbb{R}(\mathsf{r0})$ is defined to be 0. I write $X\{y \rightsquigarrow z\}$ for the map X modified so that y maps to z, replacing an old binding if present.

To simplify the presentation, I use a few notational shorthands. A state $\mathbb{S}$ can be thought of as being a mapping from registers and addresses to natural numbers. In this vein, I write $(\mathbb{M}, \mathbb{R})(l)$ for $\mathbb{M}(l)$ and $(\mathbb{M}, \mathbb{R})(r)$ for $\mathbb{R}(r)$. In other words, when I write $\mathbb{S}(\mathsf{r1})$, I mean the binding of register $\mathsf{r1}$ in the register file of state $\mathbb{S}$. Similarly, I write $(\mathbb{M}, \mathbb{R})\{l \rightsquigarrow w\}$ for $(\mathbb{M}\{l \rightsquigarrow w\}, \mathbb{R})$ and $(\mathbb{M}, \mathbb{R})\{r \rightsquigarrow w\}$ for $(\mathbb{M}, \mathbb{R}\{r \rightsquigarrow w\})$. I also write $(\mathbb{M}, \mathbb{R})$ for $\mathbb{M}$ and $(\mathbb{M}, \mathbb{R})$ for $\mathbb{R}$ when it is clear from context. When it is not, I define *memOf*$(\mathbb{M}, \mathbb{R})$ to be $\mathbb{M}$ and *rfileOf*$(\mathbb{M}, \mathbb{R})$ to be $\mathbb{R}$.

2.2.2 Dynamic semantics

The dynamic semantics of the machine is fairly standard. First I define the dynamic semantics of commands as a partial function, in Figure 2.2. This function $\text{Next}_c(\mathbb{S})$ has two arguments: the command c being executed and the data state $\mathbb{S}$ in which the command is being executed. This function returns a new data state reflecting the result of executing the command. The three arithmetic operations look up the register operands, compute the result, then store the result in register r_d. The less-than instruction evaluates to 1 if the value of r_s is less than the value of r_t, and to 0 otherwise. The 'and' instruction evaluates to 1 if the operand is odd, and 0 otherwise.

The load and store instructions load and store from memory, as expected, but are only defined when the address being loaded from or stored to is in the domain of memory. A consequence of this restriction is that the heap cannot be grown by writing to an unused memory address. This is necessary for *local reasoning*: if a block of code is verified assuming the heap has some domain, then the block of code will not alter memory outside of that domain. This property allows the *frame rule* or alternatively *heap polymorphism*, described in Section 2.7.

The small-step semantics for programs are given by a relation $\mathbb{P} \longmapsto \mathbb{P}'$, defined in Figure 2.3. I write $\mathbb{P} \longmapsto^* \mathbb{P}'$ for the reflexive, transitive closure of $\mathbb{P} \longmapsto \mathbb{P}'$. The step taken depends on the current instruction sequence. If this is a command followed by another sequence, the command is executed, and then execution continues with the remainder of the instruction sequence. For branches, if the test succeeds, then the instruction sequence corresponding to the function label f is retrieved from $\mathbb{C}$, then executed. Otherwise, the remainder of the current instruction block is executed. In either case, the data state is unchanged. For a direct jump, the instruction sequence

$$\frac{\text{Next}_c(\mathbb{S}) = \mathbb{S}'}{(\mathbb{C}, \mathbb{S}, (c; \mathbb{I})) \longmapsto (\mathbb{C}, \mathbb{S}', \mathbb{I})}$$

$$\frac{\mathbb{S}(r_s) = \mathbb{S}(r_t) \quad f \in \text{dom}(\mathbb{C})}{(\mathbb{C}, \mathbb{S}, (\mathsf{beq}\ r_s, r_t, f; \mathbb{I})) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{C}(f))} \qquad \frac{\mathbb{S}(r_s) \neq \mathbb{S}(r_t)}{(\mathbb{C}, \mathbb{S}, (\mathsf{beq}\ r_s, r_t, f; \mathbb{I})) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{I})}$$

$$\frac{\mathbb{S}(r_s) \neq \mathbb{S}(r_t) \quad f \in \text{dom}(\mathbb{C})}{(\mathbb{C}, \mathbb{S}, (\mathsf{bne}\ r_s, r_t, f; \mathbb{I})) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{C}(f))} \qquad \frac{\mathbb{S}(r_s) = \mathbb{S}(r_t)}{(\mathbb{C}, \mathbb{S}, (\mathsf{bne}\ r_s, r_t, f; \mathbb{I})) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{I})}$$

$$\frac{f \in \text{dom}(\mathbb{C})}{(\mathbb{C}, \mathbb{S}, \mathsf{j}\ f) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{C}(f))} \qquad \frac{\mathbb{S}(r_s) \in \text{dom}(\mathbb{C})}{(\mathbb{C}, \mathbb{S}, \mathsf{jr}\ r_s) \longmapsto (\mathbb{C}, \mathbb{S}, \mathbb{C}(\mathbb{S}(r_s)))}$$

$$\frac{f \in \text{dom}(\mathbb{C})}{(\mathbb{C}, \mathbb{S}, (\mathsf{jal}\ f, f_{ret})) \longmapsto (\mathbb{C}, \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\}, \mathbb{C}(f))}$$

Figure 2.3: Program step relation

corresponding to f is retrieved from $\mathbb{C}$ and becomes the current instruction sequence. A register jump is similar, except that the function label is retrieved from the register file of the current data state. Finally, a call is like a direct jump, except that register r31 is set to the value of the return pointer prior to the jump.

2.3 SCAP

The basic program logic I use is SCAP [Feng et al. 2006], which stands for Stack-based Certified Assembly Programming. SCAP is designed to reason about assembly programs that involve program calls. In a conventional Hoare logic [Hoare 1969], a specification is a *precondition*, a *postcondition*, and some *auxiliary variables* that relate the two. The precondition describes what must hold to safely execute a block of code and the postcondition describes the state after the block of code has been executed. By contrast, an SCAP specification has two parts, a *precondition* and a *guarantee*. The precondition p is a state predicate that describes what kind of states are safe to execute the block in, as in conventional Hoare logic. A guarantee

g describes the behavior of the current procedure, and is a binary state relation that relates the state at the current program point to the point at which the current procedure returns. This is similar to the specifications of Hoare Type Theory [Nanevski et al. 2007].

Guarantees have a certain elegance because they allow one to easily relate the initial and final states. For instance, to specify that the values of certain registers do not change (which is needed to specify callee-saved registers), the guarantee can require that the values are the same in both states. In conventional Hoare logic an auxiliary variable would be needed. Generally, what would be an auxiliary variable in conventional Hoare logic is in SCAP just a universal quantification inside of a guarantee, so no additional work is required to to represent auxiliary variables.

On the other hand, in my experience this style leads to a lot of redundancy between the precondition and the guarantee, as I will explain in Section 2.7. While this can mostly be factored out with a bit of discipline, it is not clear to me that guarantees are, in this setting, the best approach. In any event, the difference between precondition-postcondition specifications and precondition-guarantee specifications is fairly small in practice.

2.3.1 Instruction sequence typing rules

A precondition p has type $\mathit{State} \to \mathit{Prop}$, and a guarantee has type $\mathit{State} \to \mathit{State} \to \mathit{Prop}$. *State* is the type of states $\mathbb{S}$, and *Prop* is the type of propositions in whatever logic is being used. I write σ for code block specifications (p, g), and Ψ for code memory specifications, which are partial mappings from function labels f to specifications σ. The rules for well-formed instruction sequences are given in Figure 2.4. The judgment $\Psi; (p, g) \vdash \mathbb{I}$ ok is intended to hold if, assuming a code memory satisfying the specifications in Ψ, and assuming a data state $\mathbb{S}$ such that $p\ \mathbb{S}$ holds, then

$\boxed{\Psi; \sigma \vdash \mathbb{I}\ \text{ok}}$

$$\frac{\Psi; (p', g') \vdash \mathbb{I}\ \text{ok} \quad \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \exists \mathbb{S}'.\text{Next}_c(\mathbb{S}) = \mathbb{S}' \wedge p'\ \mathbb{S}' \wedge \forall \mathbb{S}''.\ g'\ \mathbb{S}'\ \mathbb{S}'' \rightarrow g\ \mathbb{S}\ \mathbb{S}''}{\Psi; (p, g) \vdash c; \mathbb{I}\ \text{ok}} \quad (\textsc{SeqOk})$$

$$\frac{\Psi(f) = (p', g') \quad \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow p'\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}'}{\Psi; (p, g) \vdash \mathsf{j}\ f\ \text{ok}} \quad (\text{J})$$

$$\frac{\begin{array}{l}\Psi(f) = (p', g') \quad \Psi; (p'', g'') \vdash \mathbb{I}\ \text{ok} \\ (iop, op) \in \{(\mathsf{beq}, =), (\mathsf{bne}, \neq)\} \\ \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \\ \quad (\mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \rightarrow p'\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}') \wedge \\ \quad (\neg \mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \rightarrow p''\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g''\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}')\end{array}}{\Psi; (p, g) \vdash iop\ r_s, r_t, f; \mathbb{I}\ \text{ok}} \quad (\textsc{BrOk})$$

$$\frac{\begin{array}{l}\Psi(f) = (p', g') \quad \Psi(f_{ret}) = (p'', g'') \\ \forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow\ p'\ \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\} \wedge \forall \mathbb{S}'.\ g'\ \mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\}\ \ \mathbb{S}' \rightarrow \\ \quad p''\ \mathbb{S}' \wedge \forall \mathbb{S}''.\ g''\ \mathbb{S}'\ \mathbb{S}'' \rightarrow g\ \mathbb{S}\ \mathbb{S}'' \\ \forall \mathbb{S}, \mathbb{S}'.\ p'\ \mathbb{S} \rightarrow g'\ \mathbb{S}\ \mathbb{S}' \rightarrow \mathbb{S}(\mathsf{r31}) = \mathbb{S}'(\mathsf{r31})\end{array}}{\Psi; (p, g) \vdash \mathsf{jal}\ f, f_{ret}\ \text{ok}} \quad (\textsc{Call})$$

$$\frac{\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow g\ \mathbb{S}\ \mathbb{S}}{\Psi; (p, g) \vdash \mathsf{jr}\ \mathsf{ra}\ \text{ok}} \quad (\textsc{Return})$$

Figure 2.4: Well-formed instruction sequences

an execution starting with $\mathbb{I}$ starting in state $\mathbb{S}$ will either not terminate, or will run without getting stuck until the current function reaches a return statement, in some state $\mathbb{S}'$ such that $g\ \mathbb{S}\ \mathbb{S}'$ holds.

The rules for each instruction sequence are derived fairly directly from the operational semantics. Weakening is included within each rule (instead of with a separate weakening rule) to simplify inversion on the sequence formation rules.

A sequence beginning with a command is okay, if whenever a command c is executed in a state $\mathbb{S}$ satisfying the precondition p, some state $\mathbb{S}'$ is always produced. Additionally, this new state must satisfy the precondition p' of the remainder of the current instruction block. Finally, the result of executing the command c followed by executing the rest of the instruction block (specified by g') must imply the guarantee g, where $\mathbb{S}''$ is the state from which the current procedure will eventually return. A direct jump is well-formed if the precondition of the jump instruction is stronger than the precondition of the block being jumped to, and the guarantee of the block being jumped to is weaker than the current guarantee. A branch is similar to a direct jump, except that there are two possible next states, and the result of the test is reflected into the rule. For instance, if a `beq` instruction is being executed, then if execution jumps to f the registers the instruction tested must be equal.

The rule for call looks complex, but it is just the composition of two direct jumps, with the additional twist that the call sets the value of `r31`, following the operational semantics of the abstract machine. The state $\mathbb{S}$ is the state before the call, while $\mathbb{S}'$ is the state after the call returns. Finally, $\mathbb{S}''$ is the state in which the current procedure eventually returns. This rule requires that it is safe to jump to f, and that after running f it will be safe to jump to f_{ret}.

The last part of the call rule is a kind of a side condition that requires that the function being called preserves the value of the return pointer. This is a central bit

of cleverness of the SCAP system, because it means that the callee does not have to reason about the fact that r31 contains some unknown function pointer. Instead, the callee only has to guarantee that it will leave the value of r31 alone (or more precisely, that it will restore the value before it returns). The difficult business of reasoning about the fact that the return pointer contains a function is left to the caller. Fortunately, the caller (in contrast to the callee) knows precisely what value the return pointer is, so its job is made much easier!

Notice that the only parameters to this side condition are the specifications of the function being called, so it only needs to be shown once for each function. Conceptually, the functions in Ψ are split into two categories: those that can be invoked via a call or direct jump, and those that can only be invoked via direct jump.[1]

Finally, the rule for return holds if the precondition is stronger than the guarantee, where both of the arguments to the guarantee are the current state. This is because the second argument to the guarantee is the return state for the current procedure, which is in fact the same state as the state going into a return instruction.

2.3.2 Top level typing rules

The rules for a well-formed program are given in Figure 2.5.[2] The first rule, $\Psi \vdash p$ WFST, holds if a state satisfying p contains a well-formed call stack assuming a code memory described by Ψ. p describes the state at the point of a return. The base case describes the top-level function, so there is no other function to return to. Here, for all states $\mathbb{S}$, $p\ \mathbb{S}$ must not hold, because it is never safe to return. The next

[1] It should be possible to syntactically distinguish these two classes in Ψ by giving them different "types", and then moving this register preservation side condition into the top-level code memory rule.

[2] In the actual Coq implementation these rules are implicitly baked into a more primitive program logic CAP_0 that lacks a built-in notion of function call and return, but this can just be thought of as a technique for simplifying the soundness proof of SCAP.

$\boxed{\Psi \vdash p \text{ WFST}}$ (Well-formed call stack)

$$\frac{\forall \mathbb{S}.\ p\ \mathbb{S} \rightarrow \mathit{False}}{\Psi \vdash p \text{ WFST}} \quad \text{(Base)}$$

$$\frac{\Psi(f) = (p, g) \quad \forall \mathbb{S}.\ p_0\ \mathbb{S} \rightarrow \mathbb{S}(\mathsf{r31}) = f \wedge p\ \mathbb{S} \qquad \Psi \vdash (\lambda \mathbb{S}'.\ \exists \mathbb{S}.\ p_0\ \mathbb{S} \wedge g\ \mathbb{S}\ \mathbb{S}') \text{ WFST}}{\Psi \vdash p_0 \text{ WFST}} \quad \text{(Frame)}$$

$\boxed{\Psi' \vdash \mathbb{C} : \Psi}$ (Well-formed code memory)

$$\frac{\forall f \in \mathrm{dom}(\Psi).\ \Psi'; \Psi(f) \vdash \mathbb{C}(f) \text{ ok}}{\Psi' \vdash \mathbb{C} : \Psi} \quad \text{(CdHp)}$$

$\boxed{\vdash \mathbb{P} \text{ ok}}$ (Well-formed program)

$$\frac{\Psi \vdash \mathbb{C} : \Psi \quad p\ \mathbb{S} \quad \Psi; (p, g) \vdash \mathbb{I} \text{ ok} \quad \Psi \vdash (g\ \mathbb{S}) \text{ WFST}}{\vdash (\mathbb{C}, \mathbb{S}, \mathbb{I}) \text{ ok}} \quad \text{(Prog)}$$

Figure 2.5: Typing rules for SCAP

case corresponds to a function that *can* be returned from. The precondition p_0 must imply that the return register $\mathsf{r31}$ contains the function pointer f (which is the caller of the current function), and that the precondition of the function f (according to Ψ) holds. Next, if the caller of the current function returns it must also be in a state $\mathbb{S}'$ with a well-formed call stack, where $\mathbb{S}'$ is a state related to the current state $\mathbb{S}$ by the guarantee of f, which is g, and that p_0 held on $\mathbb{S}$.

Next is the rule that describes a well-formed code memory. This rule simply requires that if the code memory specification Ψ maps f to a specification, then the actual code in $\mathbb{C}$ must satisfy that specification. In this judgment, Ψ' is the code memory specification assumed from the environment.

The final rule describes well-formed programs. First, the code memory $\mathbb{C}$ must satisfy a specification Ψ, and it must be closed: it can only assume the existence of code as described by Ψ. Next, the data state $\mathbb{S}$ must satisfy some state predicate p. The current block of code $\mathbb{I}$ being executed must be well-formed with respect to Ψ, p and some guarantee g. Finally, upon return from the current function, the state

will be describable by the state predicate $g\ \mathbb{S}$, and must be a well-formed call stack.

2.3.3 Soundness

The proof of soundness takes advantage of a strengthening lemma:

Lemma 2.3.1 (SCAP sequence strengthening) *If* $\Psi; (p, g) \vdash \mathbb{I}$ *ok and* $(\forall \mathbb{S}.\ p'\ \mathbb{S} \rightarrow p\ \mathbb{S} \wedge \forall \mathbb{S}'.\ g\ \mathbb{S}\ \mathbb{S}' \rightarrow g'\ \mathbb{S}\ \mathbb{S}')$ *then* $\Psi; (p', g') \vdash \mathbb{I}$ *ok.*

Proof: By induction on the SCAP typing derivation. □

The actual soundness theorem is standard:

Lemma 2.3.2 (SCAP soundness) *If* $\vdash \mathbb{P}$ *ok, then there exists some* $\mathbb{P}'$ *such that* $\mathbb{P} \longmapsto \mathbb{P}'$ *and* $\vdash \mathbb{P}'$ *ok.*

Proof: By induction on the SCAP program typing derivation using the SCAP sequence strengthening lemma. □

In my experience, the typing rules are so close to the operational semantics that splitting the soundness proof into the standard preservation and progress proofs does not shorten the proof.

2.4 Weak SCAP

SCAP [Feng et al. 2006] is very expressive, but from a practical perspective, it is *too* expressive because it requires that a specification be given at every program point. One would like to give explicit specifications in as few places as possible, namely at points where two control flow paths come together. In the assembly language described earlier in this chapter, the only join points are function calls and direct jumps. Therefore, given a code memory type Ψ that defines specifications for all

$\boxed{\Psi; \sigma \vdash_W \mathbb{I}\ \text{ok}}$ (Well-formed instruction sequence (weak))

$$\frac{\begin{array}{l}\Psi; (p', g') \vdash_W \mathbb{I}\ \text{ok} \\ p\ \mathbb{S} = \exists \mathbb{S}'.\ \text{Next}_c(\mathbb{S}) = \mathbb{S}' \wedge p'\ \mathbb{S}' \\ g\ \mathbb{S}\ \mathbb{S}'' = \exists \mathbb{S}'.\ \text{Next}_c(\mathbb{S}) = \mathbb{S}' \wedge g'\ \mathbb{S}'\ \mathbb{S}''\end{array}}{\Psi; (p, g) \vdash_W c; \mathbb{I}\ \text{ok}} \quad (\textsc{Seq})$$

$$\frac{\begin{array}{l}\Psi(f) = (p', g') \quad \Psi; (p'', g'') \vdash_W \mathbb{I}\ \text{ok} \\ (iop, op) \in \{(\mathsf{beq}, =), (\mathsf{bne}, \neq)\} \\ p\ \mathbb{S} = \text{if } \mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \text{ then } p'\ \mathbb{S} \text{ else } p''\ \mathbb{S} \\ g\ \mathbb{S}\ \mathbb{S}' = \text{if } \mathbb{S}(r_s)\ op\ \mathbb{S}(r_t) \text{ then } g'\ \mathbb{S}\ \mathbb{S}' \text{ else } g''\ \mathbb{S}\ \mathbb{S}'\end{array}}{\Psi; (p, g) \vdash_W iop\ r_s, r_t, f; \mathbb{I}\ \text{ok}} \quad (\textsc{Br})$$

$$\frac{\Psi(f) = (p, g)}{\Psi; (p, g) \vdash_W \mathsf{j}\ f\ \text{ok}} \quad (\text{J})$$

$$\frac{\begin{array}{l}\Psi(f) = (p', g') \quad \Psi(f_{ret}) = (p'', g'') \\ \forall \mathbb{S}, \mathbb{S}'.\ p'\ \mathbb{S} \to g'\ \mathbb{S}\ \mathbb{S}' \to \mathbb{S}(\mathsf{r31}) = \mathbb{S}'(\mathsf{r31}) \\ p\ \mathbb{S} = p'(\mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\}) \wedge \forall \mathbb{S}'.\ g'\ (\mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\})\ \mathbb{S}' \to p''\ \mathbb{S}' \\ g\ \mathbb{S}\ \mathbb{S}' = \exists \mathbb{S}'.\ g'\ (\mathbb{S}\{\mathsf{r31} \rightsquigarrow f_{ret}\})\ \mathbb{S}' \wedge g''\ \mathbb{S}'\ \mathbb{S}''\end{array}}{\Psi; (p, g) \vdash_W \mathsf{jal}\ f, f_{ret}\ \text{ok}} \quad (\textsc{Call})$$

$$\frac{p\ \mathbb{S} = \text{True} \quad g\ \mathbb{S}\ \mathbb{S}' = (\mathbb{S} = \mathbb{S}')}{\Psi; (p, g) \vdash_W \mathsf{jr}\ \mathsf{ra}\ \text{ok}} \quad (\textsc{Return})$$

Figure 2.6: Typing rules for WeakSCAP

code blocks it should be possible require no specifications for individual instructions. Ideally a verified *verification condition generator* [Necula 1997] would be able to take a program block and its desired specification and automatically derive a proposition that implies that the block matches the given specification.

To fulfill this goal, I define a new variant of SCAP called WeakSCAP (the name is intended to suggest that it involves the weakest precondition), which is constrained enough that for a given instruction sequence and code memory type Ψ there is only a single possible valid specification (assuming Ψ is a function). I show that validity in WeakSCAP implies validity in SCAP, and show how to use WeakSCAP to automatically produce a VC for an instruction sequence. Individual instruction sequences will be verified with WeakSCAP, then transformed into instruction sequences verified with SCAP and combined into verified programs, so no new top-level rules for WeakSCAP are needed. The intention is that specification checked for a block is the weakest specification, but I do not attempt to prove this. Empirically speaking, however, these rules were enough to verify a variety of garbage collectors.

The sequence typing rules for WeakSCAP are given in Figure 2.6. Notice that the specification of each instruction sequence is given entirely in terms of either the specification of a subterm or the code memory type Ψ. Thus if the specification of each program label used in the block is in Ψ the specification of each block can be automatically derived.

For an instruction sequence that starts with a command, the tail $\mathbb{I}$ of the instruction sequence has some specification (p', g'). The precondition of the entire sequence is then that the instruction c can be safely executed to produce some new state $\mathbb{S}'$ such that $\mathbb{S}'$ satisfies the precondition p' of the remaining instruction sequence. The guarantee is similar: the initial state $\mathbb{S}$ is related to the final state $\mathbb{S}'$ via some intermediate state $\mathbb{S}'$ that represents the state after c is executed.

For a branch sequence, the precondition is that if the test succeeds the precondition of the label being jumped to must be satisfied. Otherwise, the precondition of the remainder of the current instruction sequence must be satisfied. Similarly with the guarantee.

For a direct jump, the specification of the jump is simply the specification of the code label being jumped to.

The call rule is once again the most difficult. For the precondition, the precondition of the function being called must be satisfied, after the return register is set, and after the called function is executed the precondition of the return label must be satisfied. The guarantee is the composition of two guarantees. This rule also has the same side condition as regular SCAP: the function call must preserve the value of the the return register.

Finally, the rule for a return statement says that the precondition is True (the top-level rule prevents the function from returning if this is the top-level function) and that the guarantee is simply that the current state and the state being returned to are equal.

The relationship between WeakSCAP and SCAP can be formalized as the following theorem:

Lemma 2.4.1 (WeakSCAP soundness) *If* $\Psi; \sigma \vdash_W \mathbb{I}$ *ok then* $\Psi; \sigma \vdash \mathbb{I}$ *ok.*

Proof: By induction on the WeakSCAP typing derivation. □

In other words, if an instruction sequence $\mathbb{I}$ has the specification σ in WeakSCAP (assuming Ψ), then it also has the specification σ in SCAP (again, assuming Ψ). This rule is also used to turn a WeakSCAP-verified instruction sequence into an SCAP-verified instruction sequence, so a separate soundness lemma for WeakSCAP is not needed.

There is still something missing before WeakSCAP can be used. WeakSCAP is too restrictive, because it allows selecting the specification at *no* program points, whereas it should be possible to select it at *one* program point: the top of an instruction sequence. Fortunately, combining the WeakSCAP soundness lemma with the SCAP weakening lemma produces the needed lemma:

Lemma 2.4.2 (WeakSCAP strengthened soundness) *If* $\Psi; (p, g) \vdash_W \mathbb{I}$ *ok and (for all* $\mathbb{S}$ *and* $\mathbb{S}'$*,* $p'\ \mathbb{S}$ *implies* $p\ \mathbb{S}$ *and* $g\ \mathbb{S}\ \mathbb{S}'$ *implies* $g\ \mathbb{S}\ \mathbb{S}'$*), then* $\Psi; (p', g') \vdash \mathbb{I}$ *ok*

Proof: Directly, by combining Lemmas 2.3.1 and 2.4.1. □

Using this lemma, proving that $\Psi; (p, g) \vdash \mathbb{I}$ ok (in other words, that the instruction sequence $\mathbb{I}$ satisfies specification (p, g) in SCAP) has the following steps:

1. Automatically derive a specification (p', g') such that $\Psi; (p', g') \vdash_W \mathbb{I}$ ok holds.

2. Manually prove that (p, g) is stronger than (p', g'):

$$\forall \mathbb{S}, \mathbb{S}'.\ p\ \mathbb{S} \rightarrow p'\ \mathbb{S} \wedge (g'\ \mathbb{S}\ \mathbb{S}' \rightarrow g\ \mathbb{S}\ \mathbb{S}')$$

3. Combine the previous two steps using Lemma 2.4.2, resulting in the desired proof $\Psi; (p, g) \vdash \mathbb{I}$ ok.

2.5 Separation logic

I have described a formal machine and program logic, but a way to structure the actual specifications is still needed. I need a logic capable of formally reasoning about the complex memory invariants of garbage collectors. To this end, I use *separation logic* [Reynolds 2002], which has already been shown to be expressive enough to

$A * B$	memory split into two parts. A holds on one, B holds on other
$x \mapsto y$	memory has single cell x which contains value y
$!P$	lift proposition P to a memory predicate: P holds, memory is empty
$\exists x : P.\ A$	there exists some $M : P$ such that memory satisfies $A[M/x]$
`true`	holds on any memory
`emp`	memory is empty
$A \wedge B$	memory satisfies both A and B
$\forall_* x \in S.\ A$	memory can be split up into one piece for each element $e \in S$ such that $A[e/x]$ holds on that piece

Figure 2.7: Summary of separation logic predicates

reason about garbage collection algorithms in work such as Yang [2001] for mark-sweep collectors and Birkedal et al. [2004] for copying collectors. Separation logic is a logic for reasoning about program memory.

2.5.1 Predicates

I give a summary of the key separation logic predicates I use in Figure 2.7. A more formal definition is given in Figure 2.8. The key predicate is the *separating conjunction*, written $*$. $A * B$ holds on a memory if it can be split into two disjoint parts such that A holds on one part and B holds on the other part. This connective is similar in spirit to the tensor operator $\otimes$ of linear logic [Wadler 1993]. I write $\mathbb{M} \Vdash A$ for the proposition that the memory predicate A holds on memory $\mathbb{M}$ and write $\mathbb{S} \Vdash A$ for $memOf(\mathbb{S}) \Vdash A$.

The simplest separation logic predicate is $x \mapsto y$, which holds only on memory containing exactly a single memory cell at x, which contains the value y. It is useful to embed propositions that do not involve memory into separation logic predicates, and to reflect this purity syntactically. To this end, I adopt the ! operator from linear logic. $!P$ holds on memory if the memory is empty, and the proposition P holds. Similarly, $\exists x : P.\ A$ holds if there exists some object M with type P (which cannot depend on memory) such that $A[M/x]$ (A with M substituted for x) holds on the

memory. I omit the type P if it is clear from context. `true` holds on any memory. `emp` holds only on empty memory. $A \wedge B$ holds if both A and B hold on the memory.

The connectives $*$ and $\wedge$ bind more weakly than $\mapsto$ and are right associative. In other words, $a \mapsto b * a' \mapsto b' * a'' \mapsto b''$ means the same thing as $(a \mapsto b) * ((a' \mapsto b') * (a'' \mapsto b''))$.

The final important separation logic predicate I will need to describe garbage collector memory is iterated separating conjunction, which was introduced in Birkedal et al. [2004], and is written $\forall_* x \in S.\ A$. Informally, this means that the memory can be split up into one piece for every element k of the finite set S, such that $A[k/x]$ holds on the piece for that element, and all of the pieces are disjoint. This is very useful for reasoning about object heaps. If S is a set containing the addresses of objects, and A is a predicate that specifies that memory contains a well-formed object x, then $\forall_* x \in S.\ A$ describes memory containing all of the objects in S. These objects do not overlap, and are all well-formed. I will define this predicate more formally in the next section.

There are a few useful standard abbreviations I use. $x \mapsto -$ is defined to be $\exists v.\ x \mapsto v$. In other words, the address x contains some unknown value. $x \mapsto y, z$ is defined as $(x \mapsto y) * (x+4 \mapsto z)$. In other words, the memory is a pair where the first element is x and the first and second elements of the pair are y and z. This notion can be generalized to an arbitrary number of memory cells, so that, for instance, $x \mapsto a, b, c$ is a memory with three adjacent cells containing a, b and c, starting at address x. I may also combine these two types of abbreviations. For instance, $x \mapsto -, -$ is a pair at address x with some unknown contents.

2.5.2 Basic properties

There are many useful properties of these predicates. I will give a few of them to help guide an intuitive understanding of these operations. I write $A \Rightarrow B$ for $\forall \mathbb{M}.\ \mathbb{M} \vdash A \rightarrow \mathbb{M} \vdash B$ and $A \Leftrightarrow B$ for $\forall \mathbb{M}.\ \mathbb{M} \vdash A \leftrightarrow \mathbb{M} \vdash B$.

$*$ is associative: $A * (B * C) \Leftrightarrow (A * B) * C$

$*$ is commutative: $A * B \Leftrightarrow B * A$

`emp` can be freely added or removed: $A * \texttt{emp} \Leftrightarrow A$

`true` can always be added (but not removed): $A \Rightarrow A * \texttt{true}$

Memory cannot contain two bindings for a single address x: $\neg(\mathbb{M} \vdash x \mapsto - * x \mapsto -)$

(To see why this is true, consider the following: assume a memory $\mathbb{M}$ such that $\mathbb{M} \vdash x \mapsto - * x \mapsto -$. From $*$, there must exist two memories $\mathbb{M}_1$ and $\mathbb{M}_2$ with disjoint domains such that $\mathbb{M}_1 \vdash x \mapsto -$ and $\mathbb{M}_2 \vdash x \mapsto -$. But from $x \mapsto -$ the domains of both $\mathbb{M}_1$ and $\mathbb{M}_2$ must contain x, so there is a contradiction.)

`true` can be combined with or split with itself: $\texttt{true} * \texttt{true} \Leftrightarrow \texttt{true}$

An implication can be applied to one part of $*$ without disrupting the other part. For instance, if $A \Leftrightarrow A'$, then $A * B \Leftrightarrow A' * B$.

$\wedge$ can be distributed over $*$: $(A \wedge B) * C \Rightarrow (A * C) \wedge (B * C)$.

The distribution does not hold in the other direction: $(A * C) \wedge (B * C) \not\Rightarrow (A \wedge B) * C$. To see why this is the case, consider a memory $\mathbb{M}$ that contains two cells at addresses x and y (where $x \neq y$). It must be that $(x \mapsto - * \texttt{true}) \wedge (y \mapsto - * \texttt{true})$, because both $x \mapsto - * \texttt{true}$ and $y \mapsto - * \texttt{true}$ hold on the memory. In the first case, the `true` represents the part of the memory containing y and in the second case it represents the part containing x. But it is not true that $(x \mapsto - \wedge y \mapsto -) * \texttt{true}$, because a memory cannot both consist entirely of a cell at x and entirely of a single

cell at y, when $x \neq y$.

2.5.3 Machine properties

To verify programs the behavior of the machine must be related to the various separation logic predicates. The two memory operations are reading and writing. I want to use separation logic predicates to describe when these operations are safe, and to reflect the result of these operations back into a separation logic predicate.

For safety, if $\mathbb{M} \vDash x \mapsto - * A$ holds, then x is in the domain of $\mathbb{M}$ and can be safely read from or written to in $\mathbb{M}$.

A second type of property shows what happens to a separation logic predicate when one of these operations is executed on the machine.

For a read operation, if $\mathbb{M} \vDash x \mapsto w * A$ holds, then $\mathbb{M}(x) = w$, so reading the address x in memory will produce the value w.

For a write operation, if $\mathbb{M} \vDash x \mapsto - * A$ then $\mathbb{M}\{x \rightsquigarrow w\} \vDash x \mapsto w * A$. In other words, if the value w is written to address x, the part of the predicate describing x is updated to reflect the write, and the rest is left undisturbed. This demonstrates the utility of the separating conjunction: if an address is updated, the rest of the memory is unchanged, so any property of the rest of the memory (in this case A) will remain will still hold after the write.

2.6 Implementation

My implementation uses a combination of shallow and deep embeddings [Wildmoser and Nipkow 2004]. The abstract machine (including programs) and the program logic use a deep embedding, by representing each component as a Coq data type. This allows the abstract machine to have a very different semantics than that of Coq.

$$\mathbb{M} \vdash x \mapsto y ::= \mathbb{M} = \{x \mapsto y\}$$
$$\mathbb{M} \vdash !P ::= P \wedge \mathbb{M} \vdash \texttt{emp}$$
$$\mathbb{M} \vdash \exists x : P.\ A ::= \exists x : P.\ \mathbb{M} \vdash A$$
$$\mathbb{M} \vdash \texttt{true} ::= \text{True}$$
$$\mathbb{M} \vdash \texttt{emp} ::= \mathbb{M} = \{\}$$
$$\mathbb{M}_1 \cup \mathbb{M}_2 \vdash A_1 * A_2 ::= dom(\mathbb{M}_1) \cap dom(\mathbb{M}_2) = \emptyset \wedge \mathbb{M}_1 \vdash A_1 \wedge \mathbb{M}_2 \vdash A_2$$
$$\mathbb{M} \vdash A \wedge B ::= (\mathbb{M} \vdash A \ \wedge \ \mathbb{M} \vdash B)$$

Figure 2.8: Definition of separation logic predicates

I have developed extensive tactic support to make reasoning about this deep embedding as easy as possible. The full Coq implementation is available online [McCreight 2008].

On the other hand, the specifications are represented by a shallow embedding, as they are represented directly as Coq propositions. This allows me to easily take advantage of the powerful Coq logic when defining and reasoning about specifications. If instead my specifications were deeply embedded I would have to define a logic and show that it is sound from scratch. While in some sense this would be a constant overhead, as every SCAP program could take advantage of a single logic, it would still present additional work to be done.

A deep embedding of the predicate logic would be required if I carried out my work in Twelf [Pfenning and Schürmann 1999], as it is not possible to embed terms of the meta-logic $\mathcal{M}_\omega^+$ [Schürmann 2000] in the object logic LF [Harper et al. 1993]. This is the cost of Twelf's powerful ability to use higher-order abstract syntax.

Next I will describe the actual implementation of the separation logic predicates. I implement separation logic predicates in predicate logic as memory predicates using a shallow embedding. Once I have defined the predicates in this way, the various properties of these predicates I have previously described can be proved.

component	lines
data structures	5859
abstract machine	433
program logics	1821
separation logic	4405

Figure 2.9: Formal foundation implementation line counts

Iterated separation conjunction is inductively defined on the finite set:

$$\mathbb{M} \vdash \forall_* x \in \{\}.\ A ::= \mathbb{M} \vdash \texttt{emp}$$

$$\mathbb{M} \vdash \forall_* x \in S.\ A ::= \mathbb{M} \vdash A[k/x] * \forall_* x \in S - \{k\}.\ A \text{ where } k \in S$$

Line counts of various parts of the formal foundation are given in Figure 2.9. These line counts include white space and comments. The first line, data structures, covers various data structures that I use throughout the development. The largest number of lines of this component is my representation of finite sets and finite maps, and various properties of these structures. The second line covers the abstract machine, as described in Section 2.2. Once I have defined finite maps, it does not take that many lines to define the machine. The next section, program logics, includes SCAP and WeakSCAP, as described in Section 2.4 and 2.3, along with the relevant properties. It also includes proofs of many properties of the underlying machine. The last line, separation logic, includes the definition of the separation logic connectives described in Section 2.5, and many properties of these connectives, including those described already plus many more. Finally, it also includes many tactics I have written to make reasoning using separation logic in Coq easier. I will describe these tactics in greater depth in Chapter 8.

The trusted computing base of my development includes parts of the first two lines. (The statement of soundness of the machine is included in the definition of the

machine.) It does not include all of the lines because those line counts also include proofs of the properties of the components of the machine, which do not need to be trusted. Of course, this is only a small part of the overall TCB, which also includes Coq, and the operating system and hardware needed to run Coq.

2.7 Specification style

My specifications tend to follow a certain style. In this section, I will explain the convention and the reasoning behind certain design decisions. My main deviation from "traditional" separation logic [Reynolds 2002] is that I do not explicitly support the frame rule, which is as follows:

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

p is the precondition of the command c, while q is the post condition. r is another memory predicate that is does not contain any variables modified by c. Basically this says that if it can be shown that a command is okay using a certain part of memory, then anything that can be added to memory will not be modified by executing the program. This allows the specification of a program component to be modular, by allowing the specification to only describe the part of memory that the program component directly interacts with, while allowing it to be used by other program components that might have a larger memory.

There are two problems with an explicit frame rule. First, such a rule must be proved, and second, it is not entirely clear what a frame rule would look like for SCAP-style specifications. Instead, I explicitly include the polymorphism provided by the frame rule into every program specification. This turns out not to cause any problems, because my specifications are already fairly large, and the modifications required cause little or no additional difficulty with the proofs, due to my separa-

$$myPre(\mathbb{S}) ::= \exists x_0, x_1, \ldots . \ \mathbb{S} \vdash A * \texttt{true} \quad \wedge \quad P$$

$$\begin{array}{l} myGuar(\mathbb{S}, \mathbb{S}') ::= \\ \quad (\forall x_0, x_1, ..., B. \\ \quad\quad \mathbb{S} \vdash A * B \quad \wedge \quad P \rightarrow \\ \quad\quad \exists y_0, y_1, \ldots . \ \mathbb{S}' \vdash A' * B \quad \wedge \quad P') \wedge \\ \quad \forall r \in \{r_1, r_2, ..., r_k\}. \ \mathbb{S}(r) = \mathbb{S}'(r) \end{array}$$

Figure 2.10: Specification example

tion logic infrastructure. I will explain the modifications required in the context of explaining the format I use for specifications.

The general format of my specifications is given in Figure 2.10. A precondition has a set of variables $x_0, x_1, ..., x_n$ that are existentially quantified. These variables relate the pure and impure part of the specification. The impure part of the specification, $\mathbb{S} \vdash A * \texttt{true}$ specifies the memory (and possibly part of the register file). As noted before, this is an abbreviation for $memOf(\mathbb{S}) \vdash A * \texttt{true}$. $\vdash$ binds more tightly than the regular logical $\wedge$. A is a memory predicate describing the part of the memory needed by this program component, and will contain some of the existentially quantified variables (and may contain references to the register file of $\mathbb{S}$). The `true` is the first part of my explicit support for frame reasoning. This allows memory to contain extra parts not described by A. Finally, the proposition P describes the register file and the variables x_n.

The second part of my explicit frame rule is in the guarantee. The guarantee is split into two parts. Generally speaking, the first part describes what happens to the memory while the second describes what happens to the registers. To describe what happens to the memory, the specification must first quantify over a number of variables. This first set of variables, shown as x_0 and so on in the figure, serves the same role as auxiliary variables do in traditional Hoare logic by relating the

initial state to the final state. In traditional Hoare logic, auxiliary variables are handled outside of the specifications proper, requiring the implementation of some kind of variable binding structure, which in logics such as Coq can be tedious. I believe that the ability to represent them within the specification itself is the key advantage of SCAP-style specifications. Of note here is that universal instead of existential quantification must be used for the auxiliary variables. This is because I want to support *all* possible values for these variables, not merely some. The final auxiliary variable B is a memory predicate. This is the second component of my explicit support for frame reasoning. This B is a memory predicate that describes the remainder of the memory, serving the same role as r in the frame rule above.

The next part of the guarantee serves to constrain the auxiliary variables and is similar to the precondition. For instance, the constraint that B in fact describes part of the initial memory must be enforced. This enforcement has pure and impure components, like the precondition. The impure component is $\mathbb{S} \vDash A * B$ while the pure component is P. A and P are often the same as in the precondition of this procedure (but do not have to be). Finally, the specification describes the final state. As before, there are three components: an impure component, a pure component, and existential variables relating them. The impure component $\mathbb{S}' \vDash A' * B$ requires that while part of the memory has changed and can now be described by A', the rest of the memory has been untouched and can still be described by B.

The second and simpler component of my guarantees is a description of what happens to the register files. Generally, this simply states that all registers in some fixed set of registers have not changed, but can include other information about how one register has been copied to another. It is of course not always possible to describe what happens to the register files without referring to the memory (for instance, if the specification will state that a register contains the value of some memory location

```
lw r3,0(r1)     //  r3 = *r1
lw r4,0(r2)     //  r4 = *r2
sw r4,0(r1)     //  *r1 = r4
sw r3,0(r2)     //  *r2 = *r3
jr ra           //  return
```

Figure 2.11: Swap assembly implementation

it must refer to the memory), so some description of what happens to the register file must happen inside of the part devoted to the memory. Separating register file specifications from the memory specifications, where possible, makes it easier to use register file specifications.

2.8 Example

To conclude this chapter, I will present an example procedure along with its specification, then informally describe how to verify that the code matches the specification. The procedure I will be verifying is a simple pointer swap routine, defined in Figure 2.11. The left column is the actual assembly implementation, while the right column is the implementation in a C-like language. The procedure has two arguments, registers r1 and r2. Each argument is a pointer to a single word of data. The procedure swaps the contents of the cells r1 and r2 point to using temporary registers r3 and r4, then returns.

I will verify an "alias-free" specification for the swap routine that does not allow the two pointers that are having their contents swapped to be equal. While separation logic can elegantly describe situations where aliasing is banned entirely, it is not as clean when it is allowed. This will not cause problems for my GC specifications. The specification for the swap routine, made up of the precondition and the guarantee, is given in Figure 2.12. The specification follows the style described in the previous

$$
\begin{array}{l}
\mathit{swapPre}(\mathbb{S}) ::= \mathbb{S} \vdash \mathbb{S}(\mathsf{r1}) \mapsto - \ * \ \mathbb{S}(\mathsf{r2}) \mapsto - \ * \ \mathtt{true} \\
\\
\mathit{swapGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall x_1, x_2, A. \\
\qquad \mathbb{S} \ \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_1 \ \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_2 \ \ * A \rightarrow \\
\qquad \mathbb{S}' \vdash \mathbb{S}'(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}'(\mathsf{r2}) \mapsto x_1 \ * A) \wedge \\
\quad \forall r \notin \{\mathsf{r3}, \mathsf{r4}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 2.12: Swap specification

section.

The precondition specifies when it is safe to run the swap function. The state $\mathbb{S}$ is the state in which the function is called. Because the locations stored in registers r1 and r2 are read from and written to, the values of those registers, $\mathbb{S}(\mathsf{r1})$ and $\mathbb{S}(\mathsf{r2})$, must be in the domain of the memory of $\mathbb{S}$. $\mathbb{S}(\mathsf{r1}) \mapsto -$ specifies that a chunk of the memory contains some value at address $\mathbb{S}(\mathsf{r1})$, while $\mathbb{S}(\mathsf{r2}) \mapsto -$ does the same for address $\mathbb{S}(\mathsf{r2})$. Because these predicates are separated by $*$, the values of r1 and r2 cannot be aliased (*i.e.,* are not equal). While this is not necessary, it allows me to demonstrate $*$. Aside from those two addresses that must be valid, anything else is allowed to be in memory, as indicated by `true` (which holds on any memory).

For the guarantee, the argument $\mathbb{S}$ is the state before the swap function is called, and $\mathbb{S}'$ is the state after the swap returns. The guarantee has two parts. The first part, which is the first three lines of the guarantee, describes what happens to the memory when the swap function is called. The second part, which is the last line, describes what happens to the registers. The second part simply says that the values of all the registers (aside from the two temporary registers r3 and r4) stay the same.

The first part is first quantified over x_1, x_2 and A. x_1 and x_2 must be the initial values of the memory locations that registers r1 and r2 point to. A is a memory predicate that describes the rest of the memory, and is used to encode the fact

that the swap function is *polymorphic* over the rest of the memory. The next line enforces the vision of the quantified variables: arbitrary values of x_1, x_2 and A are not allowed. This line is similar to the precondition, in that it describes the initial state. Now, though, the exact contents of the memory locations pointed to by the argument registers are specified. Finally, the third line describes what the final memory looks like: r1 now points to a location containing x_2, r2 now points to a location containing x_1. Everything else has been left alone, so the rest of the memory can still be described by A, even though the actual value of A is unknown.

2.8.1 Verification

To verify that this block of code matches the specification I have given, I must prove that for all Ψ, $\Psi; (\mathit{swapPre}, \mathit{swapPost}) \vdash \mathit{swapImpl}$ ok holds, where *swapImpl* is the implementation of the swap function given in Figure 2.11. Any code memory type Ψ can be used because swap does not call any other functions. This is the SCAP formation judgment.

Next, I want to apply the three steps for proving an SCAP specification given at the end of Section 2.4. These steps are to first automatically derive a WeakSCAP specification, then show that my desired specification (*swapPre*, *swapPost*) is stronger than this WeakSCAP specification, then combine the results of the previous two steps using a lemma to produce a proof of $\Psi; (\mathit{swapPre}, \mathit{swapPost}) \vdash \mathit{swapImpl}$ ok. The result of this is that there is a single theorem that must be manually proved: that the desired specification is stronger than the automatically derived WeakSCAP specification.

Before I show the exact specification I will make a couple of definitions to simplify things. First, in my actual implementation, instead of defining Next as a partial function, I lift it to a total function Next' using a standard option type: if with

$$
\begin{array}{l}
\forall \mathbb{S}, \mathbb{S}'.\ \mathit{swapPre}(\mathbb{S}) \rightarrow \\
\quad (\text{do } \mathbb{S}_1 \leftarrow \text{Next}_{\mathsf{lw\ r3,0(r1)}}(\mathbb{S}) \\
\quad \text{do } \mathbb{S}_2 \leftarrow \text{Next}_{\mathsf{lw\ r4,0(r2)}}(\mathbb{S}_1) \\
\quad \text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\
\quad \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\
\quad \text{True}) \wedge \\
\quad (\text{do } \mathbb{S}_1 \leftarrow \text{Next}_{\mathsf{lw\ r3,0(r1)}}(\mathbb{S}) \\
\quad \text{do } \mathbb{S}_2 \leftarrow \text{Next}_{\mathsf{lw\ r4,0(r2)}}(\mathbb{S}_1) \\
\quad \text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\
\quad \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\
\quad \mathbb{S}_4 = \mathbb{S}') \rightarrow \\
\qquad \mathit{swapGuar}(\mathbb{S}, \mathbb{S}')
\end{array}
$$

Figure 2.13: Swap validity lemma

the original Next $\text{Next}_c(\mathbb{S}) = \mathbb{S}'$, then for the lifted Next$'$, $\text{Next}'_c\mathbb{S} = \text{Some } \mathbb{S}'$. If instead $\text{Next}_c(\mathbb{S})$ is undefined, then for the lifted Next$'$, $\text{Next}'_c\mathbb{S} = \text{None}$. I can do this because it is decidable for the original Next whether or not it is defined for a particular command and state. In the remainder of the paper, when I refer to Next, I am really referring to the lifted Next$'$.

Then, I want to define a convenient notation for stringing together a series of evaluation steps, where each evaluation step might fail (by returning None), as I just defined. I do this using Haskell-style [Jones 2003] do-notation for what is in essence a "Maybe" monad. There are two cases:

$$(\text{do } x \leftarrow \text{Some } \mathbb{S};\ P) ::= P[\mathbb{S}/x]$$

$$(\text{do } x \leftarrow \text{None};\ P) ::= \text{False}$$

In other words, if an evaluation succeeds, producing some state $\mathbb{S}$, then $\mathbb{S}$ is substituted for x in the proposition P. If the evaluation fails, then the entire evaluation will "fail" by turning into False. False cannot be proved, so verification will fail.

Putting all of this together produces the lemma given in Figure 2.13: if I can prove

this lemma, I will have shown that the swap procedure matches the specification I gave. This lemma is the *verification condition* [Necula 1997] of the swap function, for the specification I gave.

First, there are some states $\mathbb{S}$ and $\mathbb{S}'$, which are the initial and final states of the swap function. The initial state will satisfy the precondition of the swap function, *swapPre*. Given that, I must show two things. First, that the function will return without getting stuck. Second, that the state in which the function returns ($\mathbb{S}'$) is related to the initial state $\mathbb{S}$ by the guarantee *swapGuar*. To show that the function will return without getting stuck, I must show that each instruction can be safely executed. To show that the guarantee will be satisfied, I must show that if a state $\mathbb{S}'$ is produced by executing the block one instruction at a time, then it will be related to the initial state by the guarantee.

The alert reader will notice that there is a lot of redundancy between the two subgoals: in both portions, the safe execution of each individual instruction must be shown, and the result reasoned about. This is an unfortunate artifact of WeakSCAP. It might be possible to avoid this by using a different verification condition generator, but it is not that bad in practice, because both sequences of instructions can be stepped through in parallel, because they have the same inputs. However, there is still some redundancy to deal with, due to the structure of the guarantee.

I will informally describe how this lemma is proved. First $\mathbb{S}$, $\mathbb{S}'$ and *swapPre*($\mathbb{S}$) are introduced into the set of hypotheses. Then I reason about the first load instruction. This instruction loads from register $\mathsf{r1}$ and places the result in register $\mathsf{r3}$. As I said, the state $\mathbb{S}$ being loaded from satisfies *swapPre*. In other words, $\mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto - * \mathbb{S}(\mathsf{r2}) \mapsto - * \texttt{true}$ holds.

If the reader recalls, the definition of $a \mapsto -$ is an abbreviation for $\exists v.\ a \mapsto v$. Thus, eliminating the existentials in the precondition produces some values x_1 and

$$\frac{\begin{array}{l}\mathbb{S}_1 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\} \\ \mathbb{S}_1 \vDash \mathbb{S}_1(\mathsf{r1}) \mapsto x_1 * \mathbb{S}_1(\mathsf{r2}) \mapsto x_2 * \mathtt{true}\end{array} \quad \begin{array}{l}\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\} \\ \mathbb{S}_2 \vDash \mathbb{S}_2(\mathsf{r1}) \mapsto x_1 * \mathbb{S}_2(\mathsf{r2}) \mapsto x_2 * \mathtt{true}\end{array}}{\begin{array}{l}(\text{do } \mathbb{S}_2 \leftarrow \text{Next}_{\mathsf{lw\ r4,0(r2)}}(\mathbb{S}_1) \\ \text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\ \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\ \text{True}) \wedge \\ (\text{do } \mathbb{S}_2 \leftarrow \text{Next}_{\mathsf{lw\ r4,0(r2)}}(\mathbb{S}_1) \\ \text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\ \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\ \mathbb{S}_4 = \mathbb{S}') \rightarrow \\ \quad \mathit{swapGuar}(\mathbb{S}, \mathbb{S}')\end{array} \quad \begin{array}{l}(\text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\ \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\ \text{True}) \wedge \\ (\text{do } \mathbb{S}_3 \leftarrow \text{Next}_{\mathsf{sw\ r4,0(r1)}}(\mathbb{S}_2) \\ \text{do } \mathbb{S}_4 \leftarrow \text{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3) \\ \mathbb{S}_4 = \mathbb{S}') \rightarrow \\ \quad \mathit{swapGuar}(\mathbb{S}, \mathbb{S}')\end{array}}$$

Figure 2.14: Reasoning about swap loads

x_2 such that $\mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_2 \ * \ \mathtt{true}$ holds.

The next step is to apply the lemmas relating separation logic predicates to the MIPS-like machine discussed in Section 2.5.3. One of these lemmas, in conjunction with the precondition, implies that loading from the address stored in $\mathsf{r1}$ and storing the result in register $\mathsf{r3}$ will update the value of $\mathsf{r3}$ in the state with x_1. In other words, $\text{Next}_{\mathsf{lw\ r3,0(r1)}}(\mathbb{S}) = \text{Some } (\mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\})$. I define $\mathbb{S}_1$ to be $\mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}$. Also, the memory of $\mathbb{S}_1$ is the same as the memory of $\mathbb{S}$, and the values of registers $\mathsf{r1}$ and $\mathsf{r2}$ have not changed, so I can conclude that: $\mathbb{S}_1 \vDash \mathbb{S}_1(\mathsf{r1}) \mapsto x_1 \ * \ \mathbb{S}_1(\mathsf{r2}) \mapsto x_2 \ * \ \mathtt{true}$ holds.

Rewriting the proposition in Figure 2.13 using these equalities and simplifying the do-notation where possible leads to the goal being the left side of Figure 2.14. The propositions above the horizontal line are the hypotheses. (The additional hypotheses that $\mathbb{S}$ is a state, and that x_1 and x_2 are natural numbers are omitted for simplicity.)

The same steps are used to reason about the second load instruction. The lemmas for reasoning about the machine using separation logic require that the part of the separation logic proposition that describes the address being loaded from is at the beginning, so the associativity and commutativity properties of $*$ must be used to

$\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\}$
$\mathbb{S}_3 = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}$
$\mathbb{S}_3 \vdash \mathbb{S}_3(\mathsf{r1}) \mapsto x_2 * \mathbb{S}_3(\mathsf{r2}) \mapsto x_2 * \texttt{true}$
(do $\mathbb{S}_4 \leftarrow \mathrm{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3)$
True) $\wedge$
(do $\mathbb{S}_4 \leftarrow \mathrm{Next}_{\mathsf{sw\ r3,0(r2)}}(\mathbb{S}_3)$
$\mathbb{S}_4 = \mathbb{S}'$) $\rightarrow$
swapGuar$(\mathbb{S}, \mathbb{S}')$

$\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\}$
$\mathbb{S}_4 = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\}$
$\mathbb{S}_4 \vdash \mathbb{S}_4(\mathsf{r1}) \mapsto x_2 * \mathbb{S}_4(\mathsf{r2}) \mapsto x_1 * \texttt{true}$
True $\wedge$
$\mathbb{S}_4 = \mathbb{S}' \rightarrow$
swapGuar$(\mathbb{S}, \mathbb{S}')$

Figure 2.15: Reasoning about swap stores

show $\mathbb{S}_1 \vdash \mathbb{S}_1(\mathsf{r2}) \mapsto x_2 * \mathbb{S}_1(\mathsf{r1}) \mapsto x_1 * \texttt{true}$, but I will leave the application of these lemmas implicit to simplify the presentation. The new intermediate state is $\mathbb{S}_2$. This results in the goal on the right side of Figure 2.14.

The same approach can be used to step through the two store instructions. The only difference is that store instructions update the memory instead of the register file, so the lemmas described in Section 2.5.3 must be used to reflect the store back into the separation logic predicate. Stepping through the first and second store instructions result in the left and right goals given in Figure 2.15. I left $\mathbb{S}_2$ in place instead of simplifying it to keep the line from getting too long. I also did some simplifications, such as $\mathbb{S}_2(\mathsf{r4}) = x_2$.

The part of the goal corresponding to safety has been reduced to True, and so can be trivially solved. The value of the final state $\mathbb{S}'$ is now defined in terms of the initial state $\mathbb{S}$. All that remains is to show that the guarantee actually holds. Things look promising: it looks like $\mathbb{S}_4$ has appropriately swapped the values that were initially in the state. But there is more work to be done, due to the structure of the guarantee.

Before proceeding, I perform some simplifications. $\mathbb{S}_4 = \mathbb{S}'$, so all instances of $\mathbb{S}_4$ can be replaced with $\mathbb{S}'$. The definition of the swap guarantee can also be unfolded.

$$\frac{\begin{array}{l}\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\} \\ \mathbb{S}' = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\} \\ \mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1 * \mathbb{S}(\mathsf{r2}) \mapsto x_2 * \mathtt{true} \\ \mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 * \mathbb{S}(\mathsf{r2}) \mapsto x_1 * \mathtt{true}\end{array}}{\begin{array}{l}(\forall x_1', x_2', A. \\ \quad \mathbb{S} \ \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_1' * \mathbb{S}(\mathsf{r2}) \mapsto x_2' * A \rightarrow \\ \quad \mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2' * \mathbb{S}(\mathsf{r2}) \mapsto x_1' * A) \wedge \\ \forall r \notin \{\mathsf{r3}, \mathsf{r4}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)\end{array}} \qquad \frac{\begin{array}{l}\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\} \\ \mathbb{S}' = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\} \\ \mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1 * \mathbb{S}(\mathsf{r2}) \mapsto x_2 * \mathtt{true} \\ \mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1' * \mathbb{S}(\mathsf{r2}) \mapsto x_2' * A\end{array}}{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2' * \mathbb{S}(\mathsf{r2}) \mapsto x_1' * A}$$

Figure 2.16: Initial reasoning about the guarantee

$$1 \quad \frac{\begin{array}{l}\mathbb{S}_2 = \mathbb{S}\{\mathsf{r3} \rightsquigarrow x_1\}\{\mathsf{r4} \rightsquigarrow x_2\} \\ \mathbb{S}' = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\} \\ \mathbb{S} \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_2 \ * A\end{array}}{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_1 \ * A} \qquad 2 \quad \frac{\begin{array}{l}\mathbb{S}' = \mathbb{S}_2\{\mathbb{S}(\mathsf{r1}) \rightsquigarrow x_2\}\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\} \\ \mathbb{S}_2 \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_1 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_2 \ * A\end{array}}{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_1 \ * A}$$

$$3 \quad \frac{\begin{array}{l}\mathbb{S}' = \mathbb{S}_3\{\mathbb{S}(\mathsf{r2}) \rightsquigarrow x_1\} \\ \mathbb{S}_3 \vDash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_2 \ * A\end{array}}{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_1 \ * A} \qquad 4 \quad \frac{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_1 \ * A}{\mathbb{S}' \vdash \mathbb{S}(\mathsf{r1}) \mapsto x_2 \ * \ \mathbb{S}(\mathsf{r2}) \mapsto x_1 \ * A}$$

Figure 2.17: Final reasoning about the guarantee

Finally, for reasons that will soon be clear, I add back what is known about the initial memory: $\mathbb{S} \vDash \mathbb{S}'(\mathsf{r1}) \mapsto x_1 \ * \ \mathbb{S}'(\mathsf{r2}) \mapsto x_2 \ * \ \mathtt{true}$. I also replace occurrences of $\mathbb{S}'(\mathsf{r1})$ and $\mathbb{S}'(\mathsf{r2})$ with $\mathbb{S}(\mathsf{r1})$ and $\mathbb{S}(\mathsf{r2})$ to simplify things (these equalities can be shown by reasoning about $\mathbb{S}$ and $\mathbb{S}'$), and drop another hypothesis that is no longer needed. This results in the situation depicted in Figure 2.16. The second part of the goal ($\forall r \notin \{\mathsf{r3}, \mathsf{r4}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)$) is straightforward to show, because the register files of $\mathbb{S}$ and $\mathbb{S}'$ are the same, except at $\mathsf{r3}$ and $\mathsf{r4}$. This leaves reasoning about the memory, which looks like the right side of that figure after the new hypothesis is introduced.

The first problem I must deal with is that the values I have to show are swapped (x_1' and x_2') are different than the values I know about (x_1 and x_2). Fortunately, a

single address can only hold a single value, so I can prove a lemma that states that if $\mathbb{S} \vdash a \mapsto v * A$ and $\mathbb{S} \vdash a \mapsto v' * A'$, then $a = a'$. This lemma is widely useful. With this lemma, and the third and fourth hypotheses, I can show that $x_1 = x_1'$ and $x_2 = x_2'$. Once I have dropped a now-useless hypothesis, I am left with goal 1 in Figure 2.17.

I have to step forward through the instructions once again, this time following the definition of $\mathbb{S}'$ instead of explicitly tracing the execution of the commands. This is the redundancy that results from my style of guarantees that I mentioned before. To do this, I can use the same lemmas I used before. The first two steps set registers, so they do not do anything to the memory, resulting in goal 2 in Figure 2.17. Finally, I apply the memory update lemma twice, resulting in goals 3 and 4 in that same figure. S_3 is defined to be $\mathbb{S}_2\{\mathbb{S}(\texttt{r1}) \rightsquigarrow x_2\}$. In the final goal, a hypothesis exactly matches the goal, so I have proved the goal.

I have now proved that the verification condition holds, so I have shown that the swap function correctly implements the specification. The actual Coq proof corresponding to the informal proof described in this section (in `Swap.v`) is 26 lines that are not white space or comments, and the level of abstraction of the mechanized proof is fairly similar to the informal description given in this section, due to the use of specialized separation logic tactics. More aggressive tactics could further reduce the size of the proof, but the sort of basic reasoning about memory described in this section accounts for only a very small portion of the verification of garbage collectors (maybe 5% of the total lines of proof), so it did not seem worth improving for my current work. Instead, most of the difficulty is in reasoning about higher-level properties, as I will show in upcoming chapters.

2.9 Conclusion

In this chapter, I have described the formal setting for my work. I first defined the standard formal assembly-level machine I will use to implement the garbage collectors. I then described the existing Hoare-logic style program logic SCAP [Feng et al. 2006] I use to verify programs, and a variant of this logic WeakSCAP that is easier to work with. Next, I gave an overview of separation logic [Reynolds 2002], which I use to describe and reason about the complex memory invariants of standard GC algorithms. Separation logic is critically important to making my work practical. After that, I described how I implement everything in Coq, and the standard form that many of my specifications take. The chapter was concluded with a detailed example of how the components described in the chapter can be combined to verify the specification of a swap function written in assembly.

Chapter 3

Abstract Data Types for Hoare Logic

3.1 Introduction

In the previous chapter, I described the formal machine I use to implement garbage collectors and the logics I will use to reason about those programs. I now spend the next couple of chapters building a layer of infrastructure on top of this basic framework for reasoning about mutator-garbage collector interfaces. One of the core ideas of my approach to GC interfaces is to treat the garbage collected heap as an abstract data type (ADT). As a step towards describing this idea in detail, in this chapter I make the notion of an ADT more concrete and discuss how I represent ADTs in my setting of a deeply embedded Hoare logic.

I first discuss the basic idea of ADTs for GC interfaces to motivate the use of ADTs. After that, I begin with a simple notion of ADT, then discuss how ADTs can be adapted to a setting that requires a phase distinction between compile-time specification and runtime evaluation. Next, I discuss the standard approach to rep-

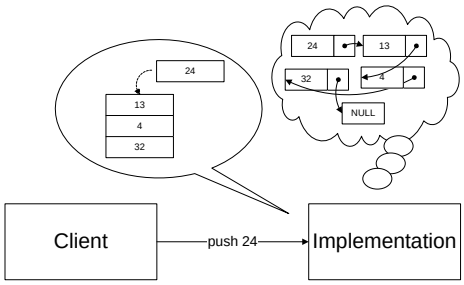

Figure 3.1: Abstract data types and clients

resenting ADTs using ML-style functors [Milner et al. 1997], then describe how I adapt this to support ADTs for SCAP programs. I illustrate the development of the various types of ADTs using a basic stack ADT. Finally, I discuss some related work and conclude.

An ADT is a data structure where the actual implementation is hidden from the user of the data type (the *client*). Figure 3.1 shows the general idea, for the example of a stack. The client is on the left. The client requests that the implementation pushes the value 24 onto the stack. The implementation tells the client that it is pushing 24 onto some abstract stack (represented by the speech bubble on the left). However, the implementation knows that in reality the stack is a linked list (represented by the thought bubble on the right). The client does not care how the stack is actually implemented, as long as the illusion is preserved.

For a garbage collector the entire garbage collected heap, including any auxiliary structures needed by the GC, can be thought of as an abstract data type. The mutator is the client, interacting with the heap via a fixed set of operations such as reading from an object, writing to an object, or allocating from an object. Representing the interface using an ADT enables hiding many implementation-specific details. This will allow reasoning about the mutator without being concerned with the details of a particular collector, which both complicates reasoning and limits the collectors the

type $Stack : Type$

val $empty : Stack$
val $push : Stack \to elt \to Stack$
val $pop : Stack \to Stack \times elt$

proof $pushPopOk : \forall s, x.\ pop(push\ s\ x) = (s, x)$

Figure 3.2: Basic ADT for stacks

mutator can be combined with. I found this was a very natural way to represent the GC interface, even with the complex incremental copying Baker collector [Baker 1978] that requires a read barrier.

3.2 Basic abstract data types

Now I will discuss abstract data types a little more formally. An ADT has three types of components:

1. A data type
2. Operations on the data type
3. High-level rules that describe the effect of the operations

The definitions of the first two are hidden from the client, allowing the client and implementer of the ADT to be defined and verified separately. The implementer of the ADT ensures that the actual implementation satisfies these rules.

For example, I give a basic ADT for stacks containing elements of some type *elt* in some unspecified functional language in Figure 3.2. Following Standard ML [Milner et al. 1997], type-level declarations are prefaced with **type**, while term-level declarations are prefaced with **val**. I also use **proof** for proof declarations, to distinguish

them from type declarations. First, there is a data type *Stack*. Next, there are three operations on *Stack*, the empty stack *empty* (which can be thought of as an operation that takes no arguments), *push* (which takes a stack and a new element, and returns a stack with the new element pushed onto the stack) and *pop*, which takes a stack and returns the top element of the stack, along with the remainder of the stack. The final component of the ADT is the behavioral rule *pushPopOk*, which specifies that if a value v is pushed onto a stack s to get a stack s' performing a pop on s' will return the pair (s, v). I do not specify what happens when a push is performed on an empty stack, allowing the implementer to do anything in that case. This component is a proof.

The client can then use any stack that matches the interface, and the implementation is free to use an array or a linked list or anything else, as long as it can show that the ADT is satisfied. This allows the client and the implementation to be separately checked, then later combined to form a whole well-formed program.

3.3 Indexed abstract data types

The previous description of ADTs has a problem: the high-level rules shown for stacks (part 3 of my definition of ADTs) contain functions. This might be fine if `push` and `pop` are implemented in a purely functional language, but what if they are implemented in a language with side effects, such as assembly? In that case, a *phase distinction* [Shao et al. 2002] should be maintained between static verification and dynamic evaluation. This can be done by specifying each operation by enriching its type, instead of with a separate rule that explicitly mentions the operation. However, as it stands, the abstract data type itself is not capable of retaining any information.

This desire to maintain phase distinction also arises in dependently typed lan-

guages with side effects such as DML [Xi and Pfenning 1999], LTT [Crary and Vanderwaart 2001], and TSCB [Shao et al. 2002], and I can adopt their solution, which is to add an *index* to the abstract type. Instead of the abstract data structures having some type T, they have a type $T\ M$, where M is some specification-level construct that contains all of the information about the data structure that the client cares about during static checking. In other words, instead of T having type *Type*, it will have type $A \rightarrow \mathit{Type}$, where A is the type of the index.

Once this is done, specifications for each operation can be given in terms of these indexed types, using standard universal and existential quantifiers, again following the example of the dependently typed languages I just mentioned.

An indexed abstract data type then has three components:

1. a data type, with an index type A
2. operations on the data type
3. a high-level specification for each operation and basic rules for the data type

3.3.1 Example

Consider the stack example. I will give two stack ADTs in some unspecified dependently typed functional language. In the first ADT, the index will be the depth of the stack. In the second ADT, the index will be a more precise listing of the contents of the stack. For clarity, I follow convention and use $\forall$ and Π for universal quantification over types in propositions and types, respectively. Each ADT has an indexed type for the data structure, and implementations of the three operations, along with a specification for each operation, in the form of a type. Finally, in each interface I include a weakening rule for stacks, to give an example of additional basic rules for the data type I mentioned. I assume the language has some way to explicitly reason

about types. These weakening rules are kind of a subtyping rule that allow the client to forget about elements at the bottom of the stack.

For instance, this rule can be used to give a stack with 5 elements the type of a stack with 3 elements. This is safe to do because there are no special rules in the interface about stacks with specific numbers of elements. If the ADT included an operation *isEmpty* that tests whether a stack is empty or not, with a specification that it returns true when the stack is empty and false when the stack is non-empty, then the weakening rule would not be sound, because then a non-empty stack would also be an empty stack, from the perspective of the types, and there would be no possible implementation of *isEmpty* that matches the specification. On the other hand, if the specification of *isEmpty* does not specify what happens when the stack is empty, the weakening rule is again perfectly fine! As this demonstrates, a variety of factors must be carefully considered when designing an interface, depending on the needs of the client.

For stacks, this weakening rule is rather contrived, but for garbage collectors weakening rules can be quite useful. For instance, it may be useful to forget that a particular register is a root to allow it to be used for values that are not managed by the GC. Weakening rules allow this.

This demonstrates another advantage of indexed-based ADTs relative to the basic ADTs: there is finer-grained control over the information exposed by the interface, which by definition makes abstraction easier. It is not clear how to define a weakening rule like this with the equational style of reasoning about operations given in Section 3.2.

type $Stack : Nat \to Type$

val $empty : Stack\ 0$
val $push : \Pi k : Nat.\ Stack\ k \to elt \to Stack\ (k+1)$
val $pop : \Pi k : Nat.\ Stack\ (k+1) \to Stack\ k \times elt$

proof $stackWeaken : \forall k, k' : Nat.\ Stack\ (k+k') \to Stack\ k$

Figure 3.3: Depth-indexed stack ADT

3.3.2 Depth-indexed stack ADT

If the client is only interested in being able to verify that each pop operation is safe (without any additional dynamic checking), the stack ADT given in Figure 3.3 can be used, where the index is a natural number (with type *Nat*) giving the depth of the stack. The empty stack has a depth 0, while the push operation takes a stack with depth k and an element, and returns a stack with depth $k+1$. The pop operation takes a stack with a depth of at least 1 and returns a stack with one less element, along with an element.[1] Finally, the lemma *stackWeaken* says that the last k' elements of a stack of depth $(k+k')$ can be forgotten.

Notice that this ADT gives less information than the basic ADT given in Figure 3.2: nothing is known about the element returned by the pop operation. From the perspective of the interface, it is perfectly fine if the value of type *elt* returned by the pop operation is not actually from the stack. As I said, this is okay if the client only cares about basic "stack safety", where it is statically verified that no pop operation fails. A more complex interface would only complicate things for the client without adding anything. Of course, as the reader will see in the next section, a stack ADT that *does* give as much information as the basic ADT can be defined.

[1] k is a natural number, so it must be at least 0, and thus $k+1$ must be at least 1.

type $Stack : List\ Elt \to Type$

val $empty : Stack\ nil$
val $push : \Pi l : List\ Elt.\ \Pi x : Elt.\ Stack\ l \to elt\ x \to Stack\ (x :: l)$
val $pop : \Pi x : Elt.\ \Pi l : List\ Elt.\ Stack\ (x :: l) \to Stack\ l \times elt\ x$

proof $stackWeaken : \forall l, l' : List\ Elt.\ Stack(l +\!\!+ l') \to Stack\ l$

Figure 3.4: List-indexed stack ADT

3.3.3 List-indexed ADT

The indexed approach can also be used to give a rich interface for stacks, as shown in Figure 3.4. Here the assumption is that the type of elements *elt* is indexed by some type *Elt*. For instance, if stack elements are natural numbers, the type *elt* could be $S_{nat} : Nat \to Type$, where *Elt* is thus *Nat*. This kind of type is known as a *singleton type* [Xi and Pfenning 1999] because every instance of every type $S_{nat}(k)$ has exactly one element.[2]

In this interface, the index type for stacks is *List Elt*, which is the type of (type-level) lists of *Elt*. This list index gives a complete picture of the contents of the stack (to the extent permitted by *Elt*). Lists l have two constructors, the empty list *nil* and a list $x :: l$ with head element x and tail list l. I write $l +\!\!+ l'$ for the type-level append operation that adds list l' to the end of the list l.

The contents of an empty stack is the empty list *nil*. The push operation takes a stack indexed by the list l and an element indexed by x and returns a stack indexed by l with x added to the front $(x :: l)$. The pop operation takes a stack indexed by any non-empty list, and returns a stack containing the tail of the list, along with

[2]While this richer stack interface allows singleton-type-like reasoning, the weakening rule means that multiple stacks can have the same type. Furthermore, a particular implementation might have multiple concrete representations of a single abstract stack. Thus the stack type is not a singleton type.

an element with the same index as the head of the list. Finally, there is again a weakening lemma: if a stack contains the elements in l followed by the elements in l', the elements in l' can be forgotten.

This interface gives as much information to the client as the basic ADT given in Figure 3.2, if *Elt* is a singleton type. An implementation of the pop operation must return the head and tail of the stack instead of arbitrary values. This enables a client to have a very precise specification. For instance, it might be possible to statically verify that a depth-first search that uses this stack correctly finds some element in a graph, if it is present, which would be impossible with the previous stack ADT.

3.4 Functor-based implementation of ADTs

I have discussed the interfaces I use for ADTs, but not how this abstraction will actually be represented. A standard way to represent an ADT client, implementation and interface makes use of an ML-style module system [Paulson 1996], which is available in languages such as Standard ML [Milner et al. 1997], OCaml [Leroy 2000], and, conveniently, the Coq proof assistant [Coq Development Team 2007b].

In this approach, there are three parts of a module system that are relevant to the implementation of ADTs. First, a *module* is an aggregation of types, definitions and theorems. Next, a *module signature* is the type of a module, giving the type of each component. A module signature can be used to hide some or all of the definitions in a module. Finally, a *functor* is kind of a function that takes a module as an argument and returns another module. The argument to the functor must be a module with a particular module signature. The module system ensures that any implementation that matches the signature can be used to instantiate the functor to create a well-formed structure.

ADT component	module system component
interface	module type
implementation	module
client	functor

Figure 3.5: Implementing ADTs using a module system

Module Type *StackSig*.
  type $Stack : Nat \to Type$

  val $empty : Stack\ 0$
  val $push : \Pi k : Nat.\ Stack\ k \to elt \to Stack\ (k+1)$
  val $pop : \Pi k : Nat.\ Stack\ (k+1) \to Stack\ k \times elt$

  proof $stackWeaken : \forall k, k' : Nat.\ Stack\ (k+k') \to Stack\ k$

End *StackSig*.

Figure 3.6: Depth-indexed stack signature

Figure 3.5 shows how each component of an ADT is represented using a module system. Interfaces are module types, while implementations are modules that match the signature of the interface signature. Finally, clients are functors that take as arguments modules matching the interface signature.

The signature that represents the ADT interface describes the entire verified implementation of the ADT, including the abstraction type and the verified implementations of the operations, but not their definitions. For example, Figure 3.6 shows what the signature for the depth-indexed stack ADT might look like in a dependently-typed language with a module system along the lines of Coq. To simplify the presentation, I do not use Coq's concrete syntax, but something equivalent to this signature can be defined in Coq. Notice that I have just wrapped up the interface defined in Figure 3.3 with a little bit of syntax to give it a name, *StackSig*.

In Figure 3.7, I define a module that implements the ADT defined by the module

```
Module ListStack : StackSig.
  type Stack [k : Nat] : Type ::=
    | emptyStack : Stack 0
    | pushStack : Πk : Nat, k' ≤ k + 1. Elt → Stack k → Stack k'

  val empty := emptyStack
  val push [k : Nat] (s : Stack k, e : Elt) := pushStack [k] [k + 1] e s
  val pop [k : Nat] (s : Stack (k + 1)) :=
    match s with
      | pushStack e s' ⇒ (s', e)
      | emptyStack ⇒ fail()
    end

  proof stackWeaken : ∀k, k' : Nat. Stack (k + k') → Stack k := ...

End ListStack.
```

Figure 3.7: Stack implementation using lists

signature *StackSig* using made-up syntax for a dependently typed functional language. The particular definition is not that important, and is just intended to give a bit of a sense of what it might look like. It is based on an inductively defined list type that has as an index the length of the list. The various operations are implemented as one might expect. I omit the proof of *stackWeaken* for simplicity. The main point of this diagram is that, like the signature, I am simply wrapping up the implementation in something to give it a name. For the implementation, I also have declared what the type of the module should be (in the first line), ensuring it correctly implements the stack ADT.

The *signature ascription* (giving a module a type) may or may not hide the definition of *Stack*, but for my purposes the difference is minor. If the definition is exposed, then a client can be specialized to this particular implementation. In this case, the client is not really using an ADT any more, but this does not compromise the safety of the system, which is still ensured by the underlying module system.

```
Functor CalcFn(S : StackSig).
  type rpn : Nat → Type ::=
    | endOp : rpn 0
    | plusOp : Πk : Nat. rpn k → rpn (k + 1)
    | numOp : Πk : Nat. rpn (k + 1) → rpn k

  val interp [k : Nat] (r : rpn k, s : S.stack (k + 1)) ::=
    match s with
      | endOp ⇒ S.pop [1] (s)
      | plusOp r′ ⇒
        let (s′, x) := S.pop [k + 1] (s) in
        let (s″, y) := S.pop [k] (s′) in
          interp [k − 1] (r′, S.push [k − 1] (s″, x + y))
      | numOp r′ x ⇒ interp [k + 1] (r′, S.push[k + 1](s, x))
    end
End CalcFn.
```

Figure 3.8: Client implementation

Instead, it only limits the implementations that can be used by the client.

An example of a client using the depth-based stack ADT I have defined is given in Figure 3.8. The client is a functor *CalcFn* taking a stack implementation S (*e.g.,* any module satisfying the module type *StackSig*) as its argument. This contrived example is a reverse Polish notation calculator where dependent types are used to ensure that the stack will not be exhausted. I assume a clever language that can do some kind of constraint solving, along the lines of DML [Xi and Pfenning 1999]. The main point here is that the stack type, along with the push and pop operations, are taken from the functor argument S. The client does not know what S is. S might not even exist yet.

Finally, I give an example of linking a client with an implementation. This only requires checking that the signature of the client matches the signature of the implementation, and is actually done by instantiating a functor with an appropriate module. A calculator *ListCalc* can be created by instantiating the client *CalcFn* with

the stacks defined in *ListStack*:

Module *ListCalc* := *CalcFn*(*ListStack*).

3.5 ADTs for deep embeddings

So far in this chapter I have reviewed standard approaches to representing abstract data types in the context of functional languages. These would suffice if I were implementing garbage collectors in a language with explicit-proof-based reasoning about memory such as Hoare Type Theory [Nanevski et al. 2007] or GTAL [Hawblitzel et al. 2007]. However, as described in Chapter 2, I use a *deep embedding* [Wildmoser and Nipkow 2004].

In a deep embedding, a "program" is a data structure manipulated by the underlying programming language (in my case, Coq) instead of being written directly in the language, and reasoning is not done about a function written in the language are actually being used (*e.g.,* Coq). Instead, reasoning is carried out on a program written in some other language (*e.g.,* MIPS assembly) that has been given an explicit representation and operational semantics. The state involved with the ADT is no longer implicitly maintained, but is instead passed around explicitly.

Constructing an ADT for a deep embedding requires some adjustments. At a basic level, ADTs are represented in the same way, using the module system: the actual ADT is represented using an indexed type, the interface is a module type, the implementation is a module, and the client is a functor. For simplicity, I implement each operation as an SCAP procedure and separate the definition and verification of the operations, but that is not necessary for my approach.

A deeply embedded ADT has the following components:

Module Type *StackAsmSig*.
  type $Stack : Nat \to State \to Prop$

  val $stackCode : CodeMem$
  type $stackCodeTy : CodeHeapSpec$
  proof $codeMemOk : (stackCodeTy \vdash stackCode : stackCodeTy)$

  proof $emptySpecEq : stackCodeTy(emptyLabel) = emptySpec$
  proof $pushSpecEq : stackCodeTy(pushLabel) = pushSpec$
  proof $popSpecEq : stackCodeTy(popLabel) = popSpec$

  proof $stackCodeTyDomOk : \mathrm{dom}(stackCodeTy) \subseteq \{k \mid 0 \leq k < 100\}$

  proof $stackWeaken : \forall k, k' : Nat.\ \forall s : State.\ Stack\ (k + k')\ s \to Stack\ k\ s$

End *StackAsmSig*.

Figure 3.9: SCAP ADT for stacks in a deeply embedded language

1. A *representation predicate* (instead of an abstract type) which holds on a machine state that contains an implementation of the ADT

2. A *code memory* that contains the complete implementation of all of the operations

3. A *code memory specification* that gives specifications for every block in the code memory

4. A proof that the code memory has the given code memory specification

5. For each operation f, a proof that f's code label is mapped to f's specification

6. An upper bound on the domain of the code memory specification, to allow linking

7. Any auxiliary proofs about the representation predicate

The stack ADT I have been using as a running example is translated to the deeply embedded setting in Figure 3.9. The representation predicate takes two arguments. The natural number argument is the depth of the stack, while the state argument is the fragment of the machine state that contains the ADT. If *Stack* n S holds, then the state S contains (in its entirety) a stack of depth n. This is similar to the abstract type I used before, except that the data structure is represented in the machine state instead of in the underlying language.

The next component is *stackCode* which is all of the code blocks required for the implementation, and *stackCodeTy* which is the specifications for all of the code blocks of the implementation. *Specifications* are used instead of types to describe the behavior of operations. In conventional Hoare logic, a specification would be a precondition and a postcondition, where both are predicates on *State*.

Next, I specify the public operations of *stackCode*, by defining some components of *stackCodeTy*. In this example, *pushSpecEq* is a proof that the specification at code label *pushLabel* is *pushSpec*. These proofs will allow the client to call the operations.

The exact definitions of the code labels (*emptyLabel*, *pushLabel* and *popLabel*) and specifications (*emptySpec*, *pushSpec* and *popSpec*) must be known to both the client and implementer of the ADT. I do not give their definitions as the focus of this chapter is how the components are assembled into larger structures rather than the components themselves. Generally speaking, the specifications will look like those of the swap example defined in Section 2.8 and will be defined in terms of *Stack*.

After this, I specify an upper bound on the domain of *stackCodeTy*. Two blocks of code can be linked only if they do not overlap. (Any block of code that is in the domain of *stackCode* but not *stackCodeTy* is dead code, and can be dropped from *stackCode* at link time.)

Finally, any properties of the underlying ADT are given, as before. *stackWeaken*

is the same stack weakening lemma from before, translated to the deeply embedded setting: if a state contains a stack of depth $k + k'$, then it also contains a stack of depth k.

3.5.1 Example specification

In SCAP, the specification of a function has two parts: a *precondition* describing states in which it is safe to call the function and a *guarantee* relating the states before and after the call. If the state is just memory, the precondition of the pop operation is

$$\lambda\mathbb{S}.\ \exists k.\ \mathbb{S} \vdash \mathit{stackRepr}(k+1) * \texttt{true}$$

In other words, the state must contain the representation of a stack with a depth of at least one. It can also contain anything else. The guarantee of the pop operation is

$$\begin{aligned}&\lambda\mathbb{S}, \mathbb{S}'.\ \forall k, A.\\ &\qquad \mathbb{S} \vdash \mathit{stackRepr}(k+1) * A \rightarrow\\ &\qquad \mathbb{S}' \vdash \mathit{stackRepr}(k) * A\end{aligned}$$

In other words, the final state contains a stack with a depth one less than the stack in the initial state, and the rest of the state is not changed.

3.6 Related work and conclusion

There has been other work on extending separation-logic-based program logics to support reasoning about abstraction, such as O'Hearn et al. [2004] and Parkinson and Bierman [2005]. These papers use memory management as an example of a case where abstraction is useful, by showing how to use their respective approaches to reason about implementations of `malloc` and `free`. These approaches both require

extending the program logic with additional rules for reasoning about abstraction, complicating the proof of soundness of the program logic. In contrast, my approach allows the program logic to be left unmodified by taking advantage of the expressiveness of the meta-logic to apply a standard technique for the representation of ADTs. This is a perhaps rare instance where having a mechanized implementation of a program logic makes things easier. Another (likely minor) difference is that my work is at the assembly level, whereas theirs is at a more C-like level.

O'Hearn et al. [2004] add abstraction to separation logic with a *hypothetical frame rule* that allows a collection of components (corresponding to the ADT operations) to be verified with a *resource invariant* r that is a separation logic predicate in their pre- and post-conditions. When the client is verified, this r is removed from the specification of the operations. The standard frame property of separation logic ensures that the client is unable to alter the state in such a way that r no longer holds.

One drawback of their system is that r must be a *precise* predicate, which means it distributes with $*$ (and any other predicate) over $\wedge$. In addition, their hiding is "all-or-nothing": either a cell is totally exposed to the client or it is totally hidden. I believe that my approach is more flexible, and allows more fine-grained abstraction. Also, as pointed out by Parkinson and Bierman [2005], this approach cannot handle more than one instance of the abstraction they have created, while mine can. For instance, a program could only contain a single stack. This is not a severe restriction for garbage collection, as a single program is usually only going to have a single garbage collected heap. More seriously, their example malloc does not support variable sized blocks, and Parkinson and Bierman [2005] claim that the logic of O'Hearn et al. [2004] is such that it cannot support variable sized blocks while still maintaining abstraction. This would seem to be because they cannot maintain

per-allocated-block auxiliary information, which is critical to proper abstraction for a garbage collector as well as malloc.

Parkinson and Bierman [2005] add abstraction by extending separation logic with the notion of an *abstract predicate*, which is the same as my *representation predicate*. The difference lies in how the predicate is added to the system. Instead of using a module system to enforce the abstraction, as I do, they build it into their program logic: the definition of an abstract predicate is exposed within a particular program scope. Within that scope, a rule in the program logic allows the definition to be unfolded, allowing the verification to take advantage of the implementation. Outside of the scope, the definition is opaque. Their system is powerful enough to reason about a malloc with auxiliary information.

However, their system is more coarse-grained than mine, because at a particular point in the program the abstract predicate is either completely concrete or completely abstract. Their system does not allow the implementation to export *properties* of the abstract predicate (such as *stackWeaken* in the stack example I presented) to the client. I do not see any reason their system could not be extended to support exporting properties of abstract predicates in this way, but this does demonstrate another advantage I get by making use of a standard technique instead of developing a custom extension of the program logic. They also demonstrate an extension of their approach to reason about Java-like inheritance, which I have not investigated.

In this chapter, I have described my approach to implementing abstract data types in a deeply embedded setting using a module system. This is critical to my garbage collector verification work, as I will use abstract data types to represent and reason about garbage collected heaps. This allows hiding implementation details of the collectors from the mutator, which in turn allows for verifying a mutator once and then combining it with different collectors to produce a verified garbage collected

program. My approach is similar to existing work, but I believe it is simpler (at least in practical terms) and more general, because I am applying an existing powerful approach (using a module system to represent abstract data types).

Chapter 4

The Garbage Collector Interface

4.1 Introduction

In this chapter, I describe a general approach to the formal specification of the interface between garbage collectors and mutators (where a mutator is simply a program that uses a garbage collector). The core idea is to treat the garbage collected heap as an *abstract data type* (ADT). Operations that interact with objects, such as reading and writing, become part of the interface of the ADT. My approach allows *modular verification* and *abstraction*, while remaining highly expressive.

Formalizing an interface enables the modular verification of garbage collectors and mutators: a collector can be verified once to match the interface, then later combined with different mutators, without verifying the collector again. Similarly, a mutator can be verified to be compatible with a particular garbage collector interface, then combined with any collector that satisfies that interface to form an entire verified program, without any additional verification work required. This matches up with the way in which garbage collectors and mutators are commonly used in an unverified setting, where a garbage collector is implemented as part of a language runtime

system, or in a library.

My interface allows the abstraction of implementation details, which simplifies the verification of the mutator and allows more collectors to be used with a given mutator. For instance, in a mark-sweep collector the existence of mark bits can be hidden, and in a Cheney copying collector the fact that all allocated objects are contiguous can be hidden. These are details that are critical to the proper functioning of the garbage collector, but that should neither affect nor be affected by the mutator.

My interface is also expressive, along a couple of dimensions. First, it can be tuned to verify anything from type safety to partial correctness of the mutator.[1] Secondly, it is expressive enough to verify a wide variety of collectors, including those requiring read or write barriers, such as incremental collectors.

This chapter focuses on describing the interface, which has been implemented in Coq as a module signature following the technique described in the previous chapter. I first give an overview of how I use the ADT framework described in the previous chapter to create an interface for garbage collection. Next, I describe what the abstract state looks like, and how it is related to the concrete state via a *representation predicate*. I then discuss some standard properties of the representation predicate. Then I give the abstract specifications of the three core garbage collector operations: reading, writing and allocation. After that, I discuss the parts of the interface that prove that the operations match those specifications. Finally, I show how coercions can be used to change the interface of a collector and conclude. In Chapter 5, I will discuss how various collectors implement the interface.

[1] I cannot verify total correctness only because the program logic I am using cannot reason about termination.

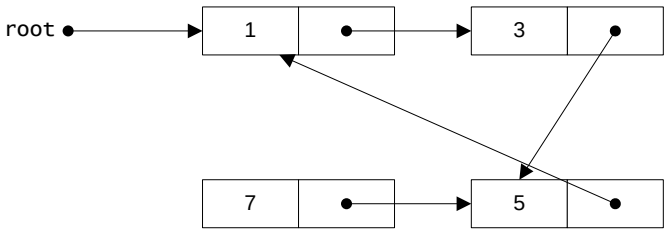

Figure 4.1: Abstract heap

4.2 Garbage collection as ADT

Now I will give a brief overview of how I use abstract data types (ADTs) (as described in Chapter 3) to reason about the mutator-garbage collector interface. At a basic level, the client of the ADT is the mutator, and the implementation of the ADT is the entire verified collector. The actual abstract data type is the entire garbage collected heap along with the root set. The operations of the ADT are all of the operations that interact with the garbage collected heap, such as reading to and writing from fields of garbage collected objects, and allocation.

As I described in Section 3.5, an ADT can be given an *index* that contains the public information about the state of the ADT. To allow very expressive reasoning about the garbage collected heap, the index I will use for my GC ADT is a high-level *abstract state* that describes the contents of every object in the heap and the values of all of the roots. This abstract state is in contrast to the low-level *concrete state* of the implementation. The use of an ADT allows hiding things such as auxiliary data structures, heap constraints maintained by the collector, and even more complex implementation details, such as whether the collector is stop-the-world or incremental. In an assembly-level machine, the state used by the garbage collector includes some of the registers and part of the memory.

I give an example abstract state in Figure 4.1. This state has a single root `root`

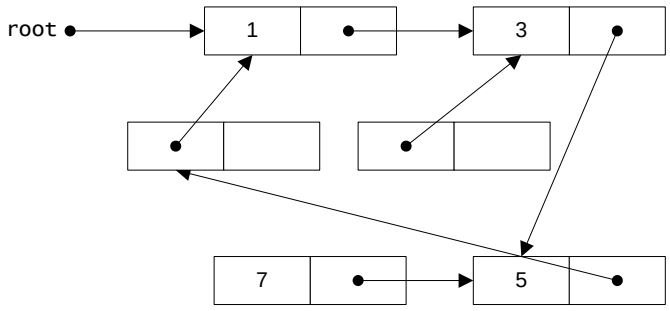

Figure 4.2: Partially copied concrete heap

and four objects. The first three objects form a cycle, while the fourth object points into the cycle, and is unreachable from the root. I will use this as a running example.

To get an idea of what sort of complexity this abstract view might be hiding, consider Figure 4.2. This is a possible concrete implementation of the abstract state in Figure 4.1, if the GC is an incremental copying collector. In this concrete state, objects 1 and 3 have been copied, while objects 5 and 7 have not. The old copies of objects 1 and 3 are in the second row, and point to the new copies, to allow the GC to later forward pointers. Notice that object 5 still points to the old copy of object 1. The collector must be implemented in a way that will prevent the mutator from discovering what is really going on.

The operations of the ADT are the various collector operations, such as *reading* from a field of an object, *writing* to the field an object, and *allocating* a new object. Besides its basic action, each operation may result in some collection activity, resulting in unreachable objects disappearing from the abstract state. The precise set of operations will depend on the nature of the garbage collector and the particular needs of the client. As seen in Section 3.5.1, the specification of each of these operations will be given in terms of the representation predicate. The abstract state contains only allocated objects, so allocation causes the abstract state to grow, while collection may cause the abstract state to shrink. Calcagno et al. [2003] show that it is

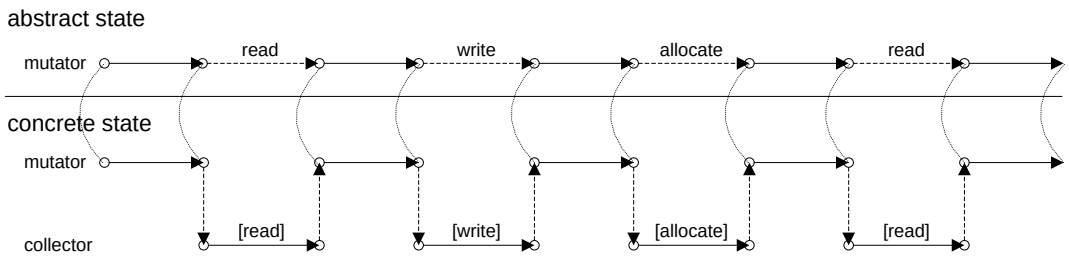

Figure 4.3: Abstract and concrete machines

possible to define a more sophisticated logic that disallows reasoning about unreachable objects, but that should be fairly orthogonal to the concerns I am examining here.

Figure 4.3 gives an overview of the entire system. The mutator sees a series of abstract states as evaluation proceeds, signified by the top line. When the collector, for instance, reads from an object, the mutator sees the high level behavior *read*, which is a binary state predicate. On the other hand, the collector sees a series of concrete states as evaluation proceeds. Each concrete state is related to a corresponding abstract state by the representation predicate, indicated on the diagram by the curved lines connecting states above and below the line dividing the abstract and concrete world. To preserve the illusion for the mutator, the concrete behavior of the collector must be some lowered version of the high-level behavior. I write $[\mathit{read}]$ for the lowered version of *read*. A function $[g]$ that lowers a guarantee g can be defined as follows, if *repr* is the representation predicate for the collector:

$$[g](\mathbb{S}, \mathbb{S}') ::= \forall \mathbb{A}.\ \mathit{repr}(\mathbb{A}, \mathbb{S}) \rightarrow \exists \mathbb{A}'.\ \mathit{repr}(\mathbb{A}', \mathbb{S}') \wedge g(\mathbb{A}, \mathbb{A}')$$

In other words, if the initial concrete state $\mathbb{S}$ contains the representation of some abstract state $\mathbb{A}$, then the final concrete state $\mathbb{S}'$ will contain the representation of some abstract state $\mathbb{A}'$ such that $\mathbb{A}$ and $\mathbb{A}'$ are related by g.

4.3 The abstract state

As I just discussed, the GC interface to the garbage collected heap is described in terms of an *abstract state*. Abstract states have two components: memory containing objects (the *object heap*) and a set of values that the mutator can immediately access and which may point into the object heap (the *root set*).

To make the following discussion clearer, I will now define the concrete and abstract states, as defined by my assembly implementation. Concrete states are the states $\mathbb{S}$ defined in Section 2.2: a pair consisting of a memory $\mathbb{M}$ and a register file $\mathbb{R}$. As a reminder, a memory is a partial function from addresses to natural numbers, while a register file is a function from registers to natural numbers. An address is a natural number divisible by four.

An abstract state $\mathbb{A}$ is a concrete state $\mathbb{S}$ plus a set of registers R. This set of registers is the root set for the collector. Only the registers in the root set are managed by the collector, allowing the mutator to manipulate some values without dealing with the mutator-collector interface. Having an explicit root allows different registers to be roots at different points in the program. For instance, after reading from an object the result is placed into register `v0`, which then becomes a root.

I define the abstract state as a variant of the concrete state for two reasons. First, this allows me to reuse my tools for reasoning about concrete states to reason about abstract states. Second, making the abstract and concrete states as close as possible shows precisely what is being abstracted by the interface. But this is not necessary and the abstract state could be made higher level. For instance, the object heap could be defined as a partial mapping from addresses to objects, instead of as a flat address space.

4.3.1 Representation predicate

As described in Section 3.5, the representation predicate relates the abstract state to the concrete state, and describes how the ADT is implemented. Each garbage collector has its own representation predicate. For simplicity, assume some particular representation predicate *repr*. The representation predicate defines how the objects and internal data structures of the collector are laid out. For instance, in a mark-sweep collector, this predicate will specify that each object has a mark bit somewhere, and that all of the objects are contiguous, to enable the sweep phase. All of this can be hidden from the mutator. In a simple copying collector, this predicate will specify that there are two semi-spaces of the same size, where all of the objects are located within a single semi space. In more complex incremental collectors, the objects themselves are abstracted.

Conceptually, *repr* can be thought of as having the type $astate \to cstate \to Prop$, where *astate* is the type of abstract states, *cstate* is the type of concrete states and *Prop* is the type of propositions. However, in my implementation, the type of *repr* is more complex. More specifically, I split the memory and register file of the concrete state, so that *repr* may be used more naturally within separation logic predicates. I also separate the root set from the rest of the abstract state. The consequence of all of this is that the type of *repr* in my implementation is $State \to RegFile \to RegSet \to Memory \to Prop$. The *State* is the memory and register file of the abstract state, while the *Memory* and *RegFile* are the memory and register file of the concrete state. Finally, the *RegSet* is the register set from the abstract state, giving the current set of root variables.

Here is a separation logic predicate that specifies that the concrete state $(\mathbb{M}, \mathbb{R})$ contains the representation of the abstract state $(\mathbb{S}, R)$ and, in a different part of the

memory, a pair at address x:

$\mathbb{M} \vdash \mathit{repr}(\mathbb{S}, \mathbb{R}, R) \ * \ x \mapsto -, -$

4.3.2 Minor interface parameters

There are a number of minor parameters to the collector interface that can be tweaked as needed. For simplicity, I fix these for the remainder of my development. Unlike *repr*, the definition of these parameters must be known by both client and implementer. For instance, the register root is the main root register. This register contains the object that will be read from or written to, and contains the only object that will survive garbage collection. It is defined to be r8. Another special register is gcInfo. This register contains a pointer to data needed by the garbage collector, and thus must be preserved by the mutator (which is reflected in the properties of *repr* given in the next section). I define this to be r16.

presReg is a register predicate that holds on registers that are preserved by the barriers. This is defined as $\mathit{presReg}(r) ::= r \in \mathit{calleeSaved}$, where *calleeSaved* is the set of callee saved registers with the standard MIPS calling conventions, which is defined as

$\mathit{calleeSaved} ::= \{\mathsf{r31}, \mathsf{r16}, \mathsf{r17}, \mathsf{r18}, \mathsf{r19}, \mathsf{r20}, \mathsf{r21}, \mathsf{r22}, \mathsf{r23}\}$.

Another parameter is *NULL*, the initial field value used by the allocator. I define *NULL* to be 1 instead of the more traditional 0, as 0 is a valid address in my memory model.

The notion of an *atomic* value must also be defined. An atomic value is a value that is not an object pointer. A value v is *atomic* (*e.g.,* is not an object pointer) if the predicate $\mathit{atomic}(v)$ holds. In my work, I define atomic values to be odd values. In other words, the lowest bit of a value is 1 for atomic values, and 0 for other values.

Finally, there are a few minor interface parameters relating to the code memory

that are analogous to those described in Section 3.5, which describes my approach to ADTs in a deeply embedded setting. *gcPhiDom* is an upper bound on the domain of the code memory type of the GC, to allow other code to be safely linked in. The addresses of each of the operations in the GC must be known so the mutator can call them. These addresses are *allocLbl*, *writeLbl1*, *writeLbl2*, *readLbl1*, *readLbl2*. The first is the address of the allocator, while the second and third are the addresses of the write operations for the first and second field. Finally, there are the labels of operations to read from the first and second fields of an object. I could have a single operation for reading, and a single operation for writing, but that makes things a little more complicated.

4.3.3 Properties of the representation predicate

To make the representation predicate useful to the mutator, some of its basic properties must be given. The mutator needs to know how abstract values that are not object pointers are represented in order to be able to operate on them. For instance, if a root register contains an abstract atomic value, how does the mutator determine if that value is equal to 1? The branch instruction is defined in terms of concrete, rather than abstract, values. These properties are the analogue of *stackWeaken* described in Chapter 3. The properties described in this section are not meant to be exhaustive, but are just a set of properties that allow verifying a few examples. A collector must prove that it satisfies all of these properties in order to implement the interface.

The first property, *atomRegEq*, states that the concrete and abstract representations of atomic values in registers are identical. For instance, if the mutator knows that a concrete value v stored in a register is atomic (perhaps by performing a run-time test), then it knows that the abstract representation of this value is also v. This

allows the mutator to transfer data to and from the abstract world of the garbage collector.

More formally, this property is written:

$$\begin{aligned}&\forall \mathbb{M}, \mathbb{A}, \mathbb{R}, R.\ \forall r \in R.\\ &\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R) \rightarrow \mathit{atomic}(\mathbb{R}(r)) \vee \mathit{atomic}(\mathbb{A}(r)) \rightarrow\\ &\quad \mathbb{R}(r) = \mathbb{A}(r)\end{aligned}$$

To break this down a bit, this first requires that the concrete state $(\mathbb{M}, \mathbb{R})$ is the representation of some abstract state $(\mathbb{A}, R)$. It also requires that either the concrete $(\mathbb{R}(r))$ or abstract $(\mathbb{A}(r))$ value of r is atomic. If all of that holds, then the concrete and abstract values of r are equal.

The next basic property *setNonRootOk* says that registers that are neither roots nor gcInfo can be changed without affecting the representation. This allows the mutator to maintain its own unmanaged data, and captures the informal notion that the only registers overseen by the GC are the roots and gcInfo.

More formally, this is written as:

$$\begin{aligned}&\forall \mathbb{M}, \mathbb{A}, \mathbb{R}, R, r, v.\\ &\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R) \rightarrow r \notin R \rightarrow r \neq \mathsf{gcInfo} \rightarrow\\ &\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}\{r \rightsquigarrow v\}, R)\end{aligned}$$

In other words, if the concrete state $(\mathbb{M}, \mathbb{R})$ is the representation of an abstract state $(\mathbb{A}, R)$, and r is a register that is neither in the root set nor equal to gcInfo, then the concrete state $(\mathbb{M}, \mathbb{R}\{r \rightsquigarrow v\})$, which is produced by setting r to some value v, is still a representation of the abstract state $(\mathbb{A}, R)$.

The next useful property is *rootCopyOk*, which states that root registers can be

copied, and that the register the value is copied to will become a root. This is safe to do because the collectors I verify do not maintain per-root information. If they did, root copying would have to be an operation provided by the collector instead of a property of the representation. This property is useful when the mutator needs to move values into or from particular registers, such as those used for function arguments or return values.

More formally, this is:

$$\begin{aligned}
&\forall \mathbb{M}, \mathbb{A}, \mathbb{R}, R, r_s, r_d. \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R) \rightarrow r_s \in R \rightarrow r_d \neq \mathsf{gcInfo} \rightarrow r_d \neq \mathsf{rzero} \rightarrow \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}\{r_d \rightsquigarrow \mathbb{A}(r_s)\}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s)\}, \{r_d\} \cup R)
\end{aligned}$$

First, the concrete state $(\mathbb{M}, \mathbb{R})$ must contain the representation of some abstract state $(\mathbb{A}, R)$. Next, there must be a source register r_s that is currently a root, and a destination register r_d that is not gcInfo (as writing to r_d would overwrite important GC information) or rzero (because the value of this register is always 0, so writing to it does not do anything). If all of the preceding facts hold, then setting the value of the register r_d in the original concrete state to the value of register r_s results in a concrete state $(\mathbb{M}, \mathbb{R}\{r_d \rightsquigarrow \mathbb{R}(r_s)\})$ that is a representation of the abstract state $(\mathbb{A}\{r_d \rightsquigarrow \mathbb{A}(r_s)\}, \{r_d\} \cup R)$. This new abstract state is the original abstract state with the value of register r_d set to the abstract value of register r_s, and with register r_d added as a root (note that it may have been a root before).

Another property that takes advantage of the fact that the collector does not maintain per-root information is *rootWeaken*. This property allows the mutator to reuse root registers for unmanaged values. This property is written more formally in

the following way:

$$\begin{aligned}
&\forall \mathbb{M}, \mathbb{A}, \mathbb{R}, R, R'. \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R) \rightarrow R' \subseteq R \rightarrow \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R')
\end{aligned}$$

In other words, if a concrete state $(\mathbb{M}, \mathbb{R})$ contains the representation of an abstract state $(\mathbb{A}, R)$ then that concrete state also contains the representation of $(\mathbb{A}, R')$, for any root set R' that is a subset of R.

The last property of *repr* that is required is *addAtomRoot*, which again takes advantage of the fact that per-root GC information is not needed, to allow the mutator to convert a non-root register containing an atomic value into a root register. This is written more formally in the following way:

$$\begin{aligned}
&\forall \mathbb{M}, \mathbb{A}, \mathbb{R}, R, r. \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}, \mathbb{R}, R) \rightarrow r \neq \mathsf{gcInfo} \rightarrow r \neq \mathsf{rzero} \rightarrow \mathit{atomic}(\mathbb{R}(r)) \rightarrow \\
&\quad \mathbb{M} \vdash \mathit{repr}(\mathbb{A}\{r \rightsquigarrow \mathbb{R}(r)\}, \mathbb{R}, \{r\} \cup R)
\end{aligned}$$

As usual, this property requires that the concrete state $(\mathbb{M}, \mathbb{R})$ contains a representation of an abstract state $(\mathbb{A}, R)$. Next, the register r that the mutator is going to promote to a root must not be gcInfo or rzero, and must contain, in the concrete state, an atomic value. r is restricted because roots can be overwritten, and gcInfo and rzero are "read only" registers, from the perspective of the mutator. The concrete value of r must be atomic because the property will write the *concrete* value of r into the *abstract* state, and the concrete and abstract representations of values are guaranteed to be equal only for atomic values. Once all of these conditions have been satisfied, then the property says that the original concrete state $(\mathbb{M}, \mathbb{R})$ contains a representation of the abstract state $(\mathbb{A}\{r \rightsquigarrow \mathbb{R}(r)\}, \{r\} \cup R)$.

As a side note, the value of rzero is always zero. This means that with my

definition of atomic (odd values), rzero will never be atomic, so *addAtomRoot* is a little redundant. But it would be reasonable to change the definition of atomic to allow 0 to be atomic (if 0 is not allowed to be a valid memory address), so I leave things as they are.

4.4 Operation specifications

The three basic operations of a garbage collector are reading from an object, writing to an object, and allocating a new object. In a standard tracing collector, most of the GC work will be done during allocation. Depending on the collector and the needs of the mutator, other operations may be needed. For instance, in a Brooks incremental copying collector [Brooks 1984], object pointer equality is different than physical equality, and thus must be implemented by the collector. As described in Section 2.3, the specification of an instruction block has two parts: a precondition (which must hold to safely execute the block) and a guarantee (which specifies the relation between the current state and the state in which the current procedure returns).

4.4.1 GC step

My specifications of the three basic operations allow garbage collection to occur. I do this to make the interface as flexible as possible. For instance, in a tracing collector such as a mark-sweep collector, collection may happen during allocation, if there is no more space. In a reference counting collector, collection may happen during a write, if the write causes the number of references to an object to drop to zero. In an incremental collector (such as the Baker GC) or a generational collector, some collection work will occur during a read or write.

$$
\begin{array}{l}
\mathit{gcStep}(\mathbb{A}, \mathbb{A}', R) ::= \\
\quad (\forall \mathit{objs}, \mathbb{M}. \\
\qquad \mathbb{A} \vdash (\mathit{minObjHp}(\{\mathbb{A}(r) \mid r \in R\}, \mathit{objs}) \wedge \mathit{eq}\ \mathbb{M}) * \mathtt{true} \rightarrow \\
\qquad \mathbb{A}' \vdash \mathit{eq}\ \mathbb{M} * \mathtt{true}) \wedge \\
\quad (\forall r \in R.\ \mathbb{A}(r) = \mathbb{A}'(r))
\end{array}
$$

Figure 4.4: Basic GC step

In the specification, the abstract behavior of garbage collection is embodied by a binary state predicate *gcStep*. As with the minor parameters described in the previous section, the definition of *gcStep* must be known to both collector and mutator. Giving it a particular name succinctly defines how the interface can be changed. This predicate is written $\mathit{gcStep}(\mathbb{A}, \mathbb{A}', R)$ where $(\mathbb{A}, R)$ and $(\mathbb{A}', R)$ are the abstract states before and after collection. This embodies the requirement that the collector preserve the set of roots.

The *gcStep* can depend somewhat on the collector. For a non-moving collector, *gcStep* states that all objects reachable from the root are preserved. This GC step is defined in Figure 4.4. The first part holds for any set of objects *objs* and memory $\mathbb{M}$. This predicate also requires that *objs* is the set of objects in the initial state $\mathbb{A}$ that are reachable from the values in the root set R and that $\mathbb{M}$ is the actual part of the abstract memory that contains these reachable objects. Given all of that, the collector guarantees that after it runs (resulting in state $\mathbb{A}'$) that the abstract memory will contain everything in $\mathbb{M}$. It also guarantees (in the last line) that none of the registers in R will be changed.

The memory predicate $\mathit{minObjHp}(V, \mathit{objs})$ will be defined precisely in Chapter 6, but informally it holds if the memory contains the objects *objs* reachable from the set of root values V and nothing else. This predicate thus describes exactly the part of the memory that the collector must preserve. The set of values $\{\mathbb{A}(r) \mid r \in R\}$ is

$$
\begin{array}{l}
\mathit{gcStep}(\mathbb{A}, \mathbb{A}', R) ::= \\
\quad \forall \mathit{objs}, \mathbb{M}. \\
\qquad \mathbb{A} \vdash (\mathit{minObjHp}(\{\mathbb{A}(r) \mid r \in R\}, \mathit{objs}) \wedge \mathit{eq}\ \mathbb{M}) * \texttt{true} \rightarrow \\
\qquad \exists \mathbb{M}'.\ \mathbb{A}' \vdash \mathit{eq}\ \mathbb{M}' * \texttt{true}\ \wedge\ \mathit{isoState}_R((\mathbb{M}, \mathit{rfileOf}(\mathbb{A})), (\mathbb{M}', \mathit{rfileOf}(\mathbb{A}')))
\end{array}
$$

Figure 4.5: Copying GC step

the set of values that the registers in R have in state $\mathbb{A}$.

For a moving collector, *gcStep* is similar, except that instead of guaranteeing that the reachable portions of the initial and final states are identical, it guarantees that they are *isomorphic*. The GC step I use for copying collectors is given in Figure 4.5. As I said, it does not guarantee that the final state contains an exact copy of the original reachable objects, or that the root registers are identical. Instead, they are isomorphic. Again, I defer a precise definition of isomorphic states until Chapter 6 (in Figure 6.7), but the basic idea is that $\mathit{isoState}_R(\mathbb{S}, \mathbb{S}')$ holds if there exists an isomorphism ϕ from the objects in $\mathbb{S}$ to the objects in $\mathbb{S}'$ such that all of the roots of $\mathbb{S}'$ are the roots of $\mathbb{S}$ forwarded according to ϕ, and such that the objects in $\mathbb{S}'$ are those of $\mathbb{S}$, copied and forwarded according to ϕ. I believe that a moving collector could have the same *gcStep* as a non-moving collector, because the interface does not expose the relative ordering of abstract object pointers, but I have not yet demonstrated this with a verified collector.

More exotic notions of garbage collector behavior are also possible. In a lazy language such as Haskell [Jones 2003], the heap may contain *thunks* in addition to values. A thunk is a computation that has been delayed. While running, a user program may evaluate a thunk and replace it with the computed value. The garbage collector, because it must traverse the memory, may find opportunities to reduce some of these thunks. I could allow this in the interface by changing the *gcStep* so that the collector will guarantee that field values will be *equivalent* instead of

$$
\begin{array}{l}
\mathit{readPre}_k(\mathbb{S}) ::= \\
\quad \exists \mathbb{A}. \quad \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, \{\mathsf{root}\}) * \mathtt{true} \wedge \\
\qquad\quad \mathbb{A} \vdash \mathbb{A}(\mathsf{root}) + k \mapsto - * \mathtt{true} \\
\\
\mathit{readGuar}_k(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathbb{A}, A, R. \\
\qquad \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, R) * A \rightarrow \\
\qquad \mathsf{root} \in R \ \wedge \ R \subseteq \{\mathsf{root}\} \cup \mathit{calleeSaved} \rightarrow \\
\qquad \exists \mathbb{A}'. \ \mathit{gcStep}(\mathbb{A}, \mathbb{A}', R) \wedge \\
\qquad\quad \mathbb{S}' \vdash \mathit{repr}(\mathbb{A}', \mathbb{S}', \{\mathsf{v0}\} \cup R) * A \wedge \\
\qquad\quad \mathbb{A}' \vdash \mathbb{A}'(\mathsf{root}) + k \mapsto \mathbb{A}'(\mathsf{v0}) * \mathtt{true}) \ \wedge \\
\quad \forall r. \mathit{presReg}(r) \wedge r \neq \mathsf{v0} \rightarrow \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 4.6: Read barrier specification

equal, which is to say that a thunk may be replaced by the evaluated value of that thunk. This may require a more expressive program logic capable of reasoning about embedded code pointers, such as XCAP [Ni and Shao 2006].

4.4.2 Read specification

Now that I can begin to give actual specifications of GC operations. Perhaps the most basic operation one might wish to perform on an object is to read the value of a field. At the abstract level, this operation takes a pointer to an object, and an offset indicating which field the mutator wants to read, and returns the value of that field.

The specification only allows reading from an object in the register root. There may be other registers in the root set but the exact root set is not fixed. The mutator can use any root set that is root plus some subset of the callee saved registers. This will only allow the barrier to do GC work that does not require knowing the entire root set. This is not a harsh restriction as no practical read barrier will examine the entire root set, which in general may be arbitrarily large.

The specification for the read barrier is given in Figure 4.6. The precondition and guarantee are parameterized by k, the offset of the field being read from. As objects are limited to pairs and memory is word-aligned, k will be either 0 or 4. The precondition says that it is safe to perform a read operation on the root register if the state contains the representation of some abstract state $\mathbb{A}$, where root is a root of the abstract state, and the abstract memory contains a value at an offset of k from the value stored in the abstract root. Because $\mathbb{A}(\mathsf{root}) + k$ is a pointer and $k = 0$ or $k = 4$, $\mathbb{A}(\mathsf{root})$ is a pointer, and thus not atomic. Therefore, because root is a valid root, $\mathbb{A}(\mathsf{root})$ must be a pointer to the start of an object.

The guarantee is a bit more complex. Following my standard approach, described in Section 2.7, the guarantee has two parts, one describing what happens to the memory and the other describing what happens to the register file. For the latter, the guarantee simply specifies the registers that are preserved by the call.

For the memory, the specification first describes the initial state. There are three auxiliary variables, $\mathbb{A}$, A, and R, used to relate the initial and final states. $\mathbb{A}$ is the initial abstract state, A is a memory predicate describing the part of the memory that does not contain the garbage collected heap, and R is the initial root set. The next line requires that the initial concrete memory be divided into two parts. The first part of the memory must contain a representation of the abstract state $(\mathbb{A}, R)$. The rest of the memory is described by A.

The third line of the guarantee gives requirements for the root set. First, the register root must be in the root set, as this is the register that will be read from. Next, all registers in R must either be root or callee-saved.

Finally, the specification describes the result of performing a read. There is some new abstract state $\mathbb{A}'$ that is related to the initial abstract state by the GC guarantee, because a collector, such as the Baker collector [Baker 1978], may perform some work

$$
\begin{array}{l}
\mathit{writePre}_k(\mathbb{S}) ::= \\
\quad \exists \mathbb{A}. \quad \mathbb{S} \vDash \mathit{repr}(\mathbb{A}, \mathbb{S}, \{\mathsf{root}, \mathsf{a0}\}) * \mathtt{true} \wedge \\
\qquad\quad \mathbb{A} \vDash \mathbb{A}(\mathsf{root}) + k \mapsto - * \mathtt{true}
\end{array}
$$

$$
\begin{array}{l}
\mathit{writeGuar}_k(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathbb{A}, A, R. \\
\qquad \mathbb{S} \vDash \mathit{repr}(\mathbb{A}, \mathbb{S}, R) * A \rightarrow \\
\qquad \{\mathsf{root}, \mathsf{a0}\} \subseteq R \rightarrow \\
\qquad \exists \mathbb{A}'.\ \mathit{gcStep}(\mathbb{A}\{\mathbb{A}(\mathsf{root}) + k \rightsquigarrow \mathbb{A}(\mathsf{a0})\}, \mathbb{A}', R) \wedge \\
\qquad\quad \mathbb{S}' \vDash \mathit{repr}(\mathbb{A}', \mathbb{S}', R) * A) \ \wedge \\
\quad \forall r. \mathit{presReg}(r) \rightarrow \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 4.7: Write barrier specification

during a read. Next, the specification describes the final concrete state $\mathbb{S}'$. The final concrete state contains the implementation of the final abstract state $\mathbb{A}'$. Notice that the return register of the read operation ($\mathsf{v0}$) is now a member of the root set (whether or not it was before). Finally, the guarantee specifies that the rest of the memory is not touched, which is reflected in the final A.

After this, the guarantee describes what the read does at the abstract level, which is to place the value of the field at offset k into register $\mathsf{v0}$. All of this is done in terms of the abstract state $\mathbb{A}'$, not the concrete state. The predicate specifies that the abstract memory contains, at address $\mathbb{A}'(\mathsf{root}) + 4$, the value being returned in register $\mathsf{v0}$. The *gcStep* relation ensures that this is the same object (for some definition of "same") as was passed in to read the operation.

4.4.3 Write specification

The write operation writes a value into a field of an object pointed to by the register root. The specification of the write operation, given in Figure 4.7, is fairly similar to that of the read operation. The only difference in the precondition is that register $\mathsf{a0}$

must be a root in addition to root, because the mutator must only write GC-managed values into the memory to ensure that the object heap is still well-formed.

The guarantee is also similar, with the same three auxiliary variables, and the initial concrete state must again contain the abstract state $\mathbb{A}$. The requirement for the register set R is looser, only requiring the presence of the two registers of interest, root and a0. This demonstrates what the register set requirement looks like if all write implementations of interest preserve all of the registers. To verify a collector that requiring a write barrier this would look more like the read barrier guarantee.

The line for the GC guarantee is more complex, because it specifies that the write occurs *before* collection, by updating the initial state with the abstract effect of the write barrier, which updates the memory at $\mathbb{A}(\mathsf{root}) + k$ with the value $\mathbb{A}(\mathsf{v0})$. I do this because the write may cause objects to become garbage after the write, and I want to give the collector the freedom to collect all objects that are garbage upon return from the write. As before, the guarantee specifies that the final abstract state $\mathbb{A}$ is implemented in the final concrete state $\mathbb{S}$. No additional line is needed to specify the abstract behavior of the write operation, as was needed for the read operation, as this is already done in the line describing the GC guarantee. The last line specifies the effect on the concrete register file, as before. A write does not change any registers, so no extra registers are excluded.

4.4.4 Allocator specification

The final specification I will describe is for the allocator. In most collectors, this operation will be much more complicated than a read or a write, but the interface is actually simpler, because most of the work that the collector will do is hidden from the mutator by the representation predicate. In contrast to the read and write barrier, the GC will need to examine the entire root set, so the root set must be

$$
\begin{array}{l}
\mathit{allocPre}(\mathbb{S}) ::= \\
\quad \exists \mathbb{A}.\ \ \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, \{\mathsf{root}\}) * \mathtt{true} \\
\\
\mathit{allocGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathbb{A}, A. \\
\qquad \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, \{\mathsf{root}\}) * A \rightarrow \\
\qquad \exists \mathbb{A}', \mathbb{A}'', x.\ \mathit{gcStep}(\mathbb{A}, \mathbb{A}', \{\mathsf{root}\}) \wedge \\
\qquad\quad \mathbb{A}'' = \mathbb{A}'\{x \rightsquigarrow \mathrm{NULL}\}\{x+4 \rightsquigarrow \mathrm{NULL}\}\{\mathsf{v0} \rightsquigarrow x\} \wedge \\
\qquad\quad \mathbb{S}' \vdash \mathit{repr}(\mathbb{A}'', \mathbb{S}', \{\mathsf{root}, \mathsf{v0}\}) * A \wedge \\
\qquad\quad (\forall B.\ \mathbb{A}' \vdash B \ \ \rightarrow \ \ \mathbb{A}'' \vdash B * x \mapsto \mathrm{NULL}, \mathrm{NULL})) \ \wedge \\
\quad \forall r. \mathit{presReg}(r) \rightarrow \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 4.8: Allocator specification

given exactly. The root set is the singleton set $\{\mathsf{root}\}$. The structure of both parts of the specification are similar to those of the previous specifications. The precondition requires that there is some abstract state represented in the concrete state.

Aside from the usual specification of the preservation of the concrete registers, the guarantee takes any representation of an abstract state $(\mathbb{A}, \{\mathsf{root}\})$. There are three existentially quantified variables in the part of the specification describing the return state. There are two abstract states $\mathbb{A}'$ and $\mathbb{A}''$ as well as an address x. The address x is the abstract location of the new object that has been allocated. For simplicity, the allocator only returns in the case of success, so there is no need to indicate failure. The first abstract state $\mathbb{A}'$ is the abstract state after the collection, as described by *gcStep*. As I said before, the root set for this collection is simply root. The second abstract state $\mathbb{A}''$ is the result of allocating an object at x in the abstract state $\mathbb{A}''$. At the abstract level, allocation is done by initializing the two fields of the object to NULL and setting the return register $\mathsf{v0}$ to x.

After this, the guarantee indicates that the final concrete state $\mathbb{S}'$ contains the representation of the final abstract state $(\mathbb{A}'', \{\mathsf{root}, \mathsf{v0}\})$. The root set is the old root, plus the object that has just been allocated. As before, the rest of the memory is

untouched, and is thus still described by A.

Finally, the guarantee specifies that all of the addresses of the newly allocated object are *fresh* (*i.e.,* were not present before the allocation occurred). While it would be possible to state this explicitly, it can be stated more cleanly using separation logic: any memory predicate B that holds on the abstract state $\mathbb{A}'$ also holds on the final abstract state $\mathbb{A}''$, which also contains separately a freshly allocated object at x. This contains the same information as explicitly specifying that the object x is disjoint from the domain of $\mathbb{A}'$, but is easier to use.

4.5 Top level components

As in Section 3.5, due to the deeply embedded setting, additional components are needed in the interface to allow the mutator to call the GC operations. The definitions of all of these components are, like *repr*, hidden from the mutator. All required properties of these components will be explicitly exported.

The first such component is $\mathbb{C}_{gc}$ which is the code memory that contains the entire implementation of all of the operations, including any subroutines they may invoke. The next component is ϕ_{gc}, which is the code memory type for $\mathbb{C}_{gc}$. The third component is a proof that $\phi_{gc} \vdash \mathbb{C}_{gc} : \phi_{gc}$ holds. In other words, the code memory $\mathbb{C}_{gc}$ correctly implements the specifications in ϕ_{gc}, and does not call on any external routines to do so.

The final set of components specify parts of ϕ_{gc} so that it can be used. The first requirement is that the domain of ϕ_{GC} is a subset of *gcPhiDom*, to allow linking with the mutator. Next, each operation's code label is mapped to the correct specification

in ϕ_{gc}, to allow the mutator to verify calls to the operations:

$$\phi_{gc}(\mathit{allocLbl}) = (\mathit{allocPre}, \mathit{allocGuar})$$
$$\phi_{gc}(\mathit{readLbl1}) = (\mathit{readPre}_0, \mathit{readGuar}_0)$$
$$\phi_{gc}(\mathit{readLbl2}) = (\mathit{readPre}_4, \mathit{readGuar}_4)$$
$$\phi_{gc}(\mathit{writeLbl1}) = (\mathit{writePre}_0, \mathit{writeGuar}_0)$$
$$\phi_{gc}(\mathit{writeLbl2}) = (\mathit{writePre}_4, \mathit{writeGuar}_4)$$

4.6 Collector coercions

While the primary users of the garbage collector interface are the mutator and the collector proper, this interface can also be used to define garbage collector *coercions*. A coercion is a functor that takes a collector C verified to satisfy some GC interface I, along with a proof that I is stronger than some other GC interface I', and returns C, except now verified to satisfy the interface I'. This improves reuse because it allows mutators to be verified with a high level interface and GCs to be verified with a low level interface. To combine the two components with incompatible interfaces only requires showing that the GC's interface is stronger.

For instance, if a mutator is verified using a type system embedded in Hoare logic [Lin et al. 2007], it can be verified with respect to a type-preserving GC. This mutator can of course be directly combined with a collector that has been verified to be type-preserving. But using a coercion, it can be combined with a collector that has been shown to be fully heap preserving (as I have discussed in this chapter) by proving that a fully heap preserving compiler is also type preserving, for whatever type system the mutator is using.

A variety of coercions is possible, but I use a fairly direct notion of compatibility here. I only allow the interfaces to vary in terms of a few of the specific components

described in this chapter. Assume a GC that implements I and another interface I', as described in this chapter. Most of the components of I and I' must be the same. The interesting component that is allowed to vary is *gcStep*. In order to be able to show that the GC at hand implements I', the following relation must hold between the GC step components of the two interfaces:

$\forall \mathbb{A}, \mathbb{A}', R.\ \ I.gcStep(\mathbb{A}, \mathbb{A}', R) \rightarrow I'.gcStep(\mathbb{A}, \mathbb{A}', R)$

In other words, if the machine takes a GC step following I, then it also takes a GC step following I'. Again, this is nothing too radical, but this can be done very easily and naturally by taking advantage of the existing module system of Coq.

4.7 Conclusion

In this chapter, I have described all of the components of the garbage collector interface. One component is a *representation predicate* that relates an abstract object heap to a concrete state, allowing many implementation details to be hidden from the mutator. Another component of the interface is a set of properties of the representation predicate that allows some basic reasoning about the abstracted heap. I then showed how to use the representation predicate to define specifications of the three most basic garbage collector operations: reading, writing, and allocation. Finally, I specified some components of the interface relating to the actual implementation and verification of each of the operations that will allow the mutator to use the operations.

With this interface, combined with the approach to ADTs described in Chapter 3, a mutator can be verified while being parameterized over a garbage collector implementation. Each block of the mutator can assume that the specifications of ϕ_{gc} are available, and thanks to the set of equalities relating to ϕ_{gc} in the interface, verify

that the calls have the specifications I have described in the previous section of this chapter. Then the mutator is able to reason about linking its own implementation to the implementation of the collector. Later, once a GC that implements the interface is verified, the mutator and GC can be combined to produce a fully verified garbage collected program. The interface in this chapter has been formally defined in Coq as a signature.

Chapter 5

Representation Predicates

5.1 Introduction

In the previous chapter, I discussed the garbage collector interface from the perspective of the mutator. In this chapter, I examine how the collector will implement one critical part of that interface, the representation predicate. The representation predicate relates the abstract state seen by the mutator to the actual concrete state. The representation predicate for a collector must enforce all of the constraints needed by that collector to properly function, such as reserving space for auxiliary data structures, making sure that those data structures contain the correct values, and ensuring that the data stored in objects is well-formed, so the GC can examine it. The collector must verify that the implementation of each operation will preserve the representation, as defined by the specifications of operations defined in Section 4.4.1.

I discuss the representation predicates in detail for three different GC algorithms: a mark-sweep collector, the Cheney copying collector, and the Baker incremental copying collector. These collectors have all been mechanically verified to match the interface described in the previous chapter [McCreight et al. 2007]. I discuss the

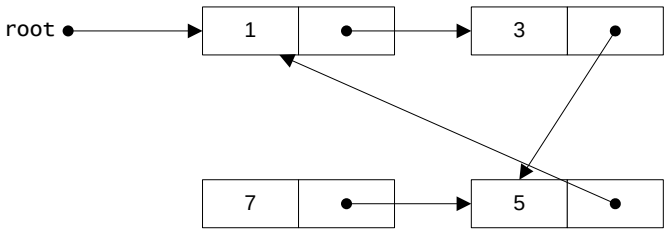

Figure 5.1: Abstract memory

actual verification of the two copying collectors later in Chapters 6 and 7.

The algorithms I discuss in this chapter are all *tracing* collectors. A tracing collector determines the set of live objects by tracing the objects that are reachable from the roots. This is in contrast to, for instance, a reference counting collector, which tracks the number of references to each object and collects an object when the number of references drops to 0. All of the collectors in this chapter use a fairly simple object layout: every object is a pair. For field values, object pointers are distinguished from other values by the lowest bit: if a field value is odd it is *atomic* (*e.g.,* not an object pointer). Otherwise, it is an object pointer. More complex objects, such as those with an arbitrary number of fields, or with another means of distinguishing atomic fields (such as headers), are certainly possible, but are left for future work. As I discussed in the previous chapter, all of the collectors I describe in this chapter store the information they need for collection in a record pointed to by the register gcInfo. As such, this register cannot be a root.

Figure 5.1 is an example of an abstract memory. I will show possible concrete representations of this abstract memory throughout the rest of the chapter.

5.2 Mark-sweep collector

The first collector invariant I will discuss is for the mark-sweep collector [Jones and Lins 1996]. I will begin by giving an overview of the mark-sweep algorithm, which has two phases. In the *mark* phase, the collector traces the set of live objects starting from the roots using a depth-first traversal. Each object has a *mark* associated with it. If an object is visited, and the mark has been set, then nothing is done. Otherwise, the mark is set, then all of the objects that the current object points to are added to a stack. When the stack is empty, the collection is complete, and all reachable objects have been marked. The simplest representation of a mark is to place the mark for an object in a header word of that object. I will be using this representation.

The *sweep* phase occurs after the mark phase has finished. In this phase, the collector examines every object in the order they appear in the object heap. If the object's mark is not set, the object is unreachable, and so it is added to a linked list containing all of the unallocated objects (the *free list*). At some point before the next mark phase, the marks of each object must be unset.

Figure 5.2 shows one possible concrete representation of the abstract state in Figure 5.1 with a mark-sweep collector. The two registers root and gcInfo point to the root of the heap and the GC information record. The first row is the object heap. The darker lines indicate object boundaries. There are a few key features of the object heap to notice. First, all of the objects are allocated contiguously, in the first row of the figure. They are contiguous so that the sweep phase can examine all of the objects. Each object has a single word header that is 0. When the collector is running, this is where it will set the mark for the object. The second-to-last object is a free object (the only free object), and thus did not show up in the abstract memory. Other than that, the actual fields of these objects look basically the same

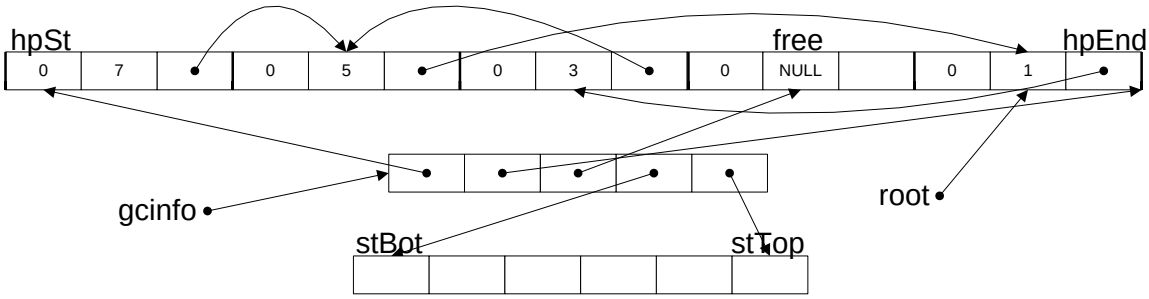

Figure 5.2: Concrete mark-sweep memory

$$
\begin{array}{lcl}
\mathit{freeList}(x, \emptyset) & ::= & !(x = \mathrm{NULL}) \\
\mathit{freeList}(x, x \cup \mathit{freeObjs}) & ::= & \exists x'.\ x \mapsto -, x' * \mathit{freeList}(x', \mathit{freeObjs} - x)
\end{array}
$$

$$\mathit{objHdrs}(S, v) ::= \forall_* x \in S.\ x - 4 \mapsto v$$

$$
\begin{array}{l}
\mathit{markSweepRepr}(\mathbb{A}, \mathbb{R}, R) ::= \\
\quad \exists \mathit{hpSt}, \mathit{hpEnd}, \mathit{free}, \mathit{stBot}, \mathit{stTop}, \mathit{objs}, \mathit{freeObjs}. \\
\qquad !(\mathsf{gcInfo} \notin R \wedge \\
\qquad\quad \mathit{objs} \cup \mathit{freeObjs} = \mathit{rangeMSObjs}(\mathit{hpSt}, \mathit{hpEnd}) \wedge \\
\qquad\quad (\forall r \in R.\ \mathbb{A}(r) = \mathbb{R}(r) \wedge \mathit{okFieldVal}(\mathit{objs}, \mathbb{A}(r)))) * \\
\qquad (\mathit{objHp}(\mathit{objs}, \mathit{objs}) \wedge \mathit{eq}\ (\mathit{memOf}(\mathbb{A}))) * \mathit{objHdrs}(\mathit{objs}, 0) * \\
\qquad \mathit{freeList}(\mathit{free}, \mathit{freeObjs}) * \mathit{objHdrs}(\mathit{freeObjs}, 0) * \\
\qquad \mathit{buffer}(\mathit{stBot}, \mathit{stTop}) * \\
\qquad \mathbb{R}(\mathsf{gcInfo}) \mapsto \mathit{hpSt}, \mathit{hpEnd}, \mathit{free}, \mathit{stBot}, \mathit{stTop}
\end{array}
$$

Figure 5.3: Mark-sweep collector representation

as they did in the abstract state.

The second row of the diagram is the GC information record, which has 5 components. The first and second elements of this record point to the start and end of the object heap, to allow the scan to know where to start and stop. The third element is a pointer to the free list, which, as I said, has only a single element here. The last two elements of the GC information record are pointers to the start and end of the mark stack, which is the third row of the diagram. The representation describes the memory when the collector is not running, so the mark stack is unused.

Figure 5.3 shows the actual definition of the type of representation I illustrated in Figure 5.2. This is a simple representation that does not obscure a lot about the concrete state: the abstract state is a subset of the concrete state. This will work for many of the collectors I will consider.

There are two new auxiliary memory predicates I define for the mark-sweep collector. The first, $\mathit{freeList}(x, S)$, asserts that the memory contains a linked list containing pairs located at all of the addresses in S. x is the head of the list. The free list predicate is inductively defined on S. The second predicate, $\mathit{objHdrs}(S, v)$, asserts that the memory contains headers for all of the objects in S, and furthermore that those headers all have the value v.

In addition, the memory predicate $\mathit{objHp}(\mathit{objs}, \mathit{objs}')$ indicates that this portion of memory contains all of the objects in the set objs', and that these objects are non-overlapping. All of the object pointers in these objects are in the set *objs*. I define this predicate more formally in Section 6.2.

Objects are located contiguously in a range from *hpSt* to *hpEnd*. Each object is a pair, and has a single word header, so there is an object every 12 bytes in the object heap. The pointer *free* points to the head of a linked list containing the free objects, given by *freeObjs*. The rest of the objects, designated by *objs*, are considered to be allocated. The pointers *stBot* and *stTop* point to the bottom and top of the an array that will be used as a stack during the mark phase.

There are a number of *pure* constraints that do not involve the memory. These are enclosed by the ! operator, adopted from linear logic. First, gcInfo cannot be a root register, because it is needed to store auxiliary GC data. Second, the combination of the allocated and unallocated objects must be equal to the set of all objects in the range from *hpSt* to *hpEnd*, given by the set $\mathit{rangeMSObjs}(\mathit{hpSt}, \mathit{hpEnd})$, which is essentially every twelfth address from *hpSt* to *hpEnd* (one header word, plus a word

for each of two fields). It may also be necessary to require that *hpSt* and *hpEnd* are pointers, depending on the definition of this set. Next the predicate requires that every register in the root set has the same value in both the concrete and abstract states. Furthermore, each register value in the root set must either be atomic or an allocated object.

The rest of the representation predicate describes the memory. The first part of the memory is made up entirely of allocated objects. The fields of these objects must either be atomic or allocated objects. This ensures that during the mark phase that every pointer encountered is an allocated object. This part of the memory is also the same as the memory of the abstract state $\mathbb{A}$, because the allocated state is the only part of the concrete state manipulated by the GC that the mutator cares about. The predicate $eq\ \mathbb{M}\ \mathbb{M}'$ is defined to be $\mathbb{M} = \mathbb{M}'$.

The next part of the memory contains the free list, which starts at *free* and contains the objects in *freeObjs*, as described earlier in this section. Then, again as described earlier, the next part of the memory contains the headers for all of the objects, which must all be 0, which is the value indicating they are unmarked. To maintain this invariant, the sweep phase must set the header of each object to 0. Next, there is the portion of the memory containing the mark stack. The memory predicate $buffer(x, y)$, defined in Section 6.2, holds on a memory if it contains all of the addresses from x to y, excluding y. Finally, gcInfo points to a record containing the various information the collector needs.

5.3 Copying collector

A copying collector also traces through reachable objects starting from the root, but instead of marking objects it has reached, it copies them to a separate region of

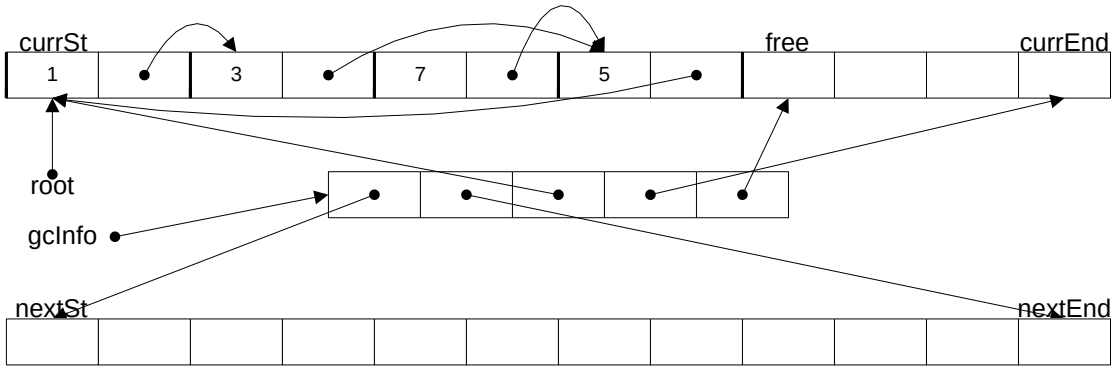

Figure 5.4: Concrete copying collector memory

$$
\begin{array}{l}
\mathit{copyRepr}(\mathbb{A}, \mathbb{R}, R) ::= \\
\quad \exists \mathit{currSt}, \mathit{currEnd}, \mathit{nextSt}, \mathit{nextEnd}, \mathit{free}, \mathit{objs}. \\
\qquad !(\mathsf{gcInfo} \notin R \wedge \\
\qquad\quad \mathit{aligned8}(\mathit{currSt}, \mathit{free}) \wedge \mathit{aligned8}(\mathit{free}, \mathit{currEnd}) \wedge \\
\qquad\quad \mathit{objs} = \mathit{rangeObjs}(\mathit{currSt}, \mathit{free}) \wedge \\
\qquad\quad (\forall r \in R.\ \mathbb{A}(r) = \mathbb{R}(r) \wedge \mathit{okFieldVal}(\mathit{objs}, \mathbb{A}(r))) \wedge \\
\qquad\quad \mathit{nextEnd} - \mathit{nextSt} = \mathit{currEnd} - \mathit{currSt} \wedge \\
\qquad\quad \mathit{nextSt} \leq \mathit{nextEnd}) * \\
\qquad (\mathit{objHp}(\mathit{objs}, \mathit{objs}) \wedge \mathit{eq}\ (\mathit{memOf}(\mathbb{A}))) * \\
\qquad \mathit{buffer}(\mathit{free}, \mathit{currEnd}) * \mathit{buffer}(\mathit{nextSt}, \mathit{nextEnd}) * \\
\qquad \mathbb{R}(\mathsf{gcInfo}) \mapsto \mathit{nextSt}, \mathit{nextEnd}, \mathit{currSt}, \mathit{currEnd}, \mathit{free}
\end{array}
$$

Figure 5.5: Copying collector representation

memory. Once all of the reachable objects have been copied, and all of their fields have been forwarded, the old region containing objects can be freed. The Cheney collector [Cheney 1970] is able to do this without any auxiliary data structures beyond the extra space being copied into. One advantage of a copying collector is that it is *compacting*: after a collection, all of the allocated objects are contiguous, which means that the free objects are also contiguous, making allocation very inexpensive.

Figure 5.4 shows what the concrete representation of the example abstract memory might look like. This is for a two-space copying collector. One *semi-space* contains allocated objects and the other will be used to copy reachable objects when

the garbage collector is run next. As before, root points to the root object and gcInfo points to a record of GC information. All of the allocated objects are contiguous, and look the same as they do in the abstract memory. As before, heavy lines indicate object boundaries. The last four blocks in that row represent the free space. The middle row of blocks is the GC information record. The first two elements of the record point to the bounds of the space being copied into. The second two elements point to the bounds of the space being copied from. The bounds of the semi-spaces are needed to do allocation, and to tell which semi-space a pointer points to, during collection. The last element of the information record points to the first available free block. The last row is the fallow semi-space, which must be the same size as the active semi-space.

In Figure 5.5 I give a representation predicate for a stop-the-world copying collector such as the Cheney collector. This is the representation predicate used by the Cheney collector described in Chapter 6, and also can be used to describe the concrete memory in Figure 5.4. The first semi-space is bounded by *currSt* and *currEnd* and contains the allocated objects. The other, bounded by *nextSt* and *nextEnd*, is where the reachable objects will be copied to when the next collection occurs. All allocated objects are contiguous, beginning at the start of the semi-space (*currSt*) and ending at the free pointer *free*. The rest of the current semi-space (from *free* to *currEnd*) is the free space, where new objects will be allocated.

There are a number of constraints that are pure (*i.e.,* do not involve the memory). First, the predicate requires that gcInfo is not a root. Next, it requires that the beginning and end of the region containing allocated objects (from *currSt* to *free*) are double-word aligned. It also requires that the beginning and end of the region containing the free space are double-word aligned (from *free* to *currEnd*). Because all objects are pairs, this ensures neither region contains fractional objects, which

simplifies reasoning. Next, *objs* is the set of currently allocated objects. The purpose of this definition is to make the representation predicate clearer and shorter. Next, the register files of the concrete and abstract state must agree, for every register in the root set. Additionally, all root registers must contain values that are well-formed with respect to the set of allocated objects. In other words, each register value must either be atomic or a member of *objs*.

The final two pure constraints ensure that collection can be done without any bounds checking. Both semi-spaces must be the same size. Additionally, the end of the destination semi-space must be greater than or equal to the start of the destination semi-space. This requirement is needed to ensure alignment after the semi-spaces are swapped, because if $nextEnd - nextSt = 0$, *nextSt* could be larger than *nextEnd*, because the machine uses arithmetic on natural numbers.

The rest of the representation predicate describes parts of the memory. First, there is the part of the memory that contains the allocated objects. This part of the memory simultaneously satisfies two constraints. First, this part of the memory contains exactly the objects in *objs*, and all of the fields that are object pointers are in the set *objs*. This enables the collector to trace the object heap without needing to verify the validity of each pointer it comes across. The second constraint that this part of the memory satisfies is that it is equal to the memory of the abstract state $\mathbb{A}$. In other words, the memory of the abstract state is the part of the concrete state containing the allocated objects, whether or not they are reachable.

The next part of the memory is a buffer containing the free objects. There is also a part of the memory that contains a buffer containing the other semi-space. Finally, the register gcInfo in the concrete state points to a record containing the values of the bounds of the semi-spaces, and a pointer to the next free object. This reduces the number of registers the collector requires. When an object is allocated, or the

semi-spaces are swapped, this record must be updated.

5.4 Incremental copying collector

An incremental collector is, as the name suggests, a garbage collector that does collection incrementally. Each time an object is allocated, a little bit of collection is done. While making a collector incremental reduces the maximum length of time the garbage collector will ever run, it also makes the representation predicate much more complex, because when the mutator is running the collection may only be partially done. In effect, the state invariant while the mutator is running is the same as the loop invariant in a stop-the-world collector. In addition, the read or write operations of the collector, which in the stop-the-world case can be implemented with a load or store operation, must be augmented to do some collection work to prevent the mutator from getting ahead of the collector. All of this has the potential to greatly complicate the mutator-collector interface, but fortunately my approach to mutator-collector interfaces described in Chapter 4 allows this complexity to be hidden behind the representation invariant. As a consequence, I am able to give the Baker collector, an incremental copying collector, the same interface as the Cheney collector, a stop-the-world copying collector.

Before I describe the representation predicate for the Baker collector, I must describe the algorithm. To a first approximation, the Baker algorithm [Baker 1978] is an incremental version of the Cheney algorithm [Cheney 1970]. The collector examines all objects that are reachable from the root, breadth-first. The first time an object is seen during this tracing, the collector copies the object to a fresh area called the *to-space*. The address of the new copy of the object is placed in the first field of the old copy of the object. This allows the collector to distinguish objects

that have been copied from those that have not, and allows the collector to find the new copy of objects. As mentioned, the main difference between the two collectors is that while the Cheney collector does not return until all reachable objects have been copied, the Baker collector may return before that happens, so there are still from-space or "stale" pointers in the heap. The collector, through the barriers, maintains the invariants that the roots will never contain stale pointers, keeping them hidden from the mutator.

Figure 5.6 gives a possible concrete representation of the same abstract memory I have been using as a running example. The top row of objects is the *from-space*, where the collector is copying objects from, and the bottom row of objects is the *to-space*, where the collector is copying objects to. As before, the heavy lines indicate objects boundaries. The middle row of blocks is the record of GC information.

The collector has started a collection, beginning at the root root. Object 1 has been copied and scanned. Scanning an object x forwards all of the fields of x, copying any objects that have not been copied yet. Objects that have been copied and scanned are known as *black objects*. The collector will not visit black objects again. In the Baker collector, all of the black objects are contiguous. Object 3 was copied when object 1 was scanned, but it has yet to be scanned itself. This is why it still points to object 5 in the from-space. Objects such as this that the collector has determined are reachable, but has yet to visit, are known as *gray objects*. Like the black objects, gray objects are contiguous in the Baker collector. After object 3 is the free space, where new objects are allocated. Objects allocated by the GC are allocated from the front, as they will be added to the set of gray objects. Objects allocated by the mutator are allocated from the back, because the mutator can only write to-space pointers into objects it allocates, so any objects it allocates will be black.

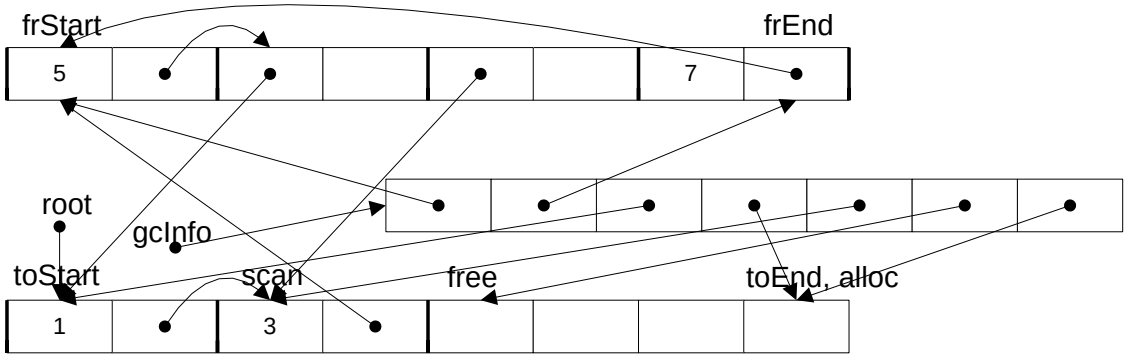

Figure 5.6: Concrete incremental copying collector memory

$$
\begin{array}{l}
\mathit{bakerConcObjHp}(W, M, G, B) ::= \\
\quad \mathit{objHp}(W \cup M, W) * \mathit{objHp}(B \cup G, B) * \mathit{objHp}(W \cup M \cup B \cup G, G) \\
\\
\mathit{bakerRepr}(\mathbb{A}, \mathbb{R}, R) ::= \\
\quad \exists \mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \phi, W, M, \mathbb{M}, \\
\quad !(\mathsf{gcInfo} \notin R \wedge \\
\qquad \text{aligned8 } [\mathit{toSt}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \mathit{toEnd}] \wedge \mathit{aligned8}(\mathit{frSt}, \mathit{frEnd}) \wedge \\
\qquad W \cup M = \mathit{rangeObjs}(\mathit{frSt}, \mathit{frEnd}) \wedge \\
\qquad M \cong_{\phi} B \cup G \wedge \\
\qquad (\forall r \in R.\ \mathbb{A}(r) = \mathbb{R}(r) \wedge \mathit{okFieldVal}(\mathit{toObjs}, \mathbb{R}(r))) \wedge \\
\qquad \mathit{memOf}(\mathbb{A}) = (\phi^* \cup \mathrm{id}_{W \cup B \cup G \cup B'}) \circ \mathbb{M}) * \\
\quad (\mathit{bakerConcObjHp}(W, M, G, B \cup B') \wedge \mathit{eq}\ \mathbb{M}) * \\
\quad \mathit{mapHp}(M, \phi) * \mathit{buffer}(\mathit{free}, \mathit{alloc}) * \\
\quad \mathbb{R}(\mathsf{gcInfo}) \mapsto \mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc} \\
\\
\quad \text{where } B = \mathit{rangeObjs}(\mathit{toSt}, \mathit{scan}), B' = \mathit{rangeObjs}(\mathit{alloc}, \mathit{toEnd}), \\
\qquad G = \mathit{rangeObjs}(\mathit{scan}, \mathit{free}),\ \text{and } \mathit{toObjs} = B \cup G \cup B'
\end{array}
$$

Figure 5.7: Baker collector representation

The forwarding pointers of objects 1 and 3 are stored in the old copies of 1 and 3, which are the second and third objects in the from-space. The first field of a copied object points to the new location. Thus, given a pointer to an old object, the collector can easily tell what the new location of the object is. Besides the old copies, the from-space also includes objects 5 and 7. Notice that object 5 still points to the old copy of object 1. These objects, which the GC has yet to determine if they are reachable or not, are known as *white objects*. Eventually, the GC will realize that object 5 must be copied, but object 7 is not reachable, and will eventually be freed. There is no space left in the from-space, because the collector is not invoked until there is no more space. The classification of objects into black, gray and white is known as the *tricolor invariant*, and is due to Dijkstra et al. [1978].

Finally, I will discuss the GC information record. The first two elements are pointers to the beginning and end of the from-space, while the third and fourth elements point to the beginning and end of the to-space. The fifth element, known as the *scan pointer* points to the first gray object. The sixth element is the *free pointer*, and points to the first free object. The final element is the *alloc pointer*, and points to the *last* free object. The free pointer is used to allocate new objects for the GC, while the alloc pointer is used to allocate new objects for the mutator, for reasons described above.

Now I will formalize these and other constraints, by giving the representation invariant for the Baker collector. This representation invariant, defined in Figure 5.7, will be used in the verification of an implementation of a Baker collector I will describe in Chapter 7.

The predicate *bakerConcObjHp* enforces the basic color constraints for the concrete object heap. If these invariants are violated, the collector will not function properly. The set of white from-space objects W have not been examined, while

the mapped from-space objects M have already been examined and copied. The set of gray to-space objects G have been copied from M, but have not had their fields updated. Finally, the set of to-space black objects B have been copied and had their fields updated. White objects can only point to from-space objects ($W \cup M$), while black objects can only point to to-space objects ($B \cup G$). This latter constraint ensures that when the set of gray objects G is empty that the entire from-space can be collected without creating any dangling pointers. And last, gray objects can point to any valid object in either space. Gray objects may point to from-space objects as a result of their initial copying and to the to-space as a result of mutator writes occurring after copying. Objects in M are not part of the concrete object heap because they have already been copied to G.

That done, I can describe the representation predicate itself. First I will describe the existentially quantified variables. *frSt* and *frEnd* are the beginning and end of the from-space, containing the old copies of the objects. The to-space is bounded by *toSt* and *toEnd*. The scan pointer is *scan*, while the free pointer is *free*. When *scan* = *free*, the collection is finished. The alloc pointer is *alloc*. ϕ is the map from the old locations of objects to their new locations. The sets of objects W and M are as I described in the previous paragraph. Finally, $\mathbb{M}$ is the portion of the concrete memory that corresponds to the abstract memory. Unlike the previous collectors described in this chapter, the abstract memory is not simply a subset of the concrete memory.

There are four more sets of objects, with definitions given at the bottom of the figure. These sets have the same basic meaning as in my description of *bakerConcObjHp*, but there is more information now. Each object set in fact covers a specific range, and the set of black objects is broken into B, which contains objects that used to be gray, and B', which contains objects that have been newly allocated during the

current collector cycle. Separating the sets B and B' is needed to enable reasoning about the copying that has been done during the collection cycle. The set B ranges from the start of the to-space to the scan pointer, while the set G ranges from the scan pointer to the free pointer. The set B' ranges from the allocation pointer to the end of the to-space. Collectively, B, B' and G form the allocated to-space objects *toObjs*. The rest of the to-space is the free space where new objects will be allocated.

For the pure constraints of the representation predicate, the predicate first requires that gcInfo is not a root, as before. Next, it requires that the various pointers of interest are double-word-aligned, to prevent fractional objects, and ensure the expected relative ordering of pointers (as $x - y = 0$ does not imply $x = y$ with natural arithmetic). We write aligned8 $[x_0, x_1, ..., x_k]$ for $\mathit{aligned8}(x_0, x_1) \wedge \mathit{aligned8}(x_1, x_2) \wedge ... \wedge \mathit{aligned8}(x_{k-1}, x_k)$. Next, the two sets of from-space objects, W and M, must together cover the entire from-space. ϕ must be an isomorphism from M to $B \cup G$, to ensure that, for instance, two from-space objects are not forwarding to a single to-space object. Notice that B' is not in the range of ϕ, as objects in B' are not copies of from-space objects. This is why the predicate must distinguish B and B'. Next, the predicate requires that the value of every root register is the same in the abstract and concrete state, and that these roots are either atomic or to-space objects. Note that from-space objects are not allowed in the root set. This is an important invariant that prevents black objects from being contaminated with stale pointers, and ensures that the root set only needs to be examined once per collection cycle.

The final "pure" constraint is the most complex and the most interesting. The heap of the abstract state $\mathit{memOf}(\mathbb{A})$ must be equal to $(\phi^* \cup \mathrm{id}_{W \cup B \cup G \cup B'}) \circ \mathbb{M}$. This relates the portion of the concrete memory containing allocated objects $\mathbb{M}$ to the abstract memory $\mathit{memOf}(\mathbb{A})$. The abstract memory is the same as the concrete

memory, except that any references to objects that have been copied are replaced with their new locations. The basic idea is to construct a map that forwards stale values (those in M) while leaving all other field values alone, then apply this map to the range of the concrete memory by composing it with the concrete memory.

Following Birkedal et al. [2004], I write ϕ^* for the extension of the map ϕ with the identity function on odd values. I write id_S for the identity function on the set S. Therefore, $\phi^* \cup \mathrm{id}_{W \cup B \cup G \cup B'}$ is a partial map that is the identity function for all atomic values as well as object pointers that have either already been forwarded (those in $B \cup G \cup B'$) or those that have not been copied (those in W). At the same time, every value in M is forwarded using the isomorphism ϕ.

It might seem to make more sense to forward the objects in W to their final destination, because this would ensure that the entire abstract state only had forwarded objects. Unfortunately, the interleaving of mutator and collector activity means that it is impossible to determine which objects in W will eventually be copied, let alone the order in which they will be copied. A white object reachable in one state might become unreachable due to a later store performed by the mutator, meaning that the object will never be copied.

Now I will describe the specifications that deal more directly with the memory. As I have done for the other collectors, I first specify the object heap. The basic color constraints are specified by *bakerConcObjHp*, described earlier. In addition to these color constraints, the predicate specifies that the portion of memory containing these allocated objects is equal to $\mathbb{M}$, which is used above to specify the abstract memory.

The next portion of the memory contains the objects that have already been copied, which must implement the isomorphism ϕ: the first field of each object $x \in M$ is $\phi(x)$. The predicate *mapHp* used to specify this is defined in Section 6.2.

The next component of the memory is the free space. This begins at the free pointer and ends at the allocation pointer. Finally, as before, the various boundary pointers are stored in a record that the register gcInfo points to.

Notice that there is no relationship specified between the sizes of the from- and to-spaces, in contrast to the Cheney collector. As a consequence, it is possible that the Baker collector will run out of space during a collection, so a dynamic check will be needed. This is done for simplicity.

5.5 Conclusion

In this chapter, I have described in detail the representation predicates for three different types of garbage collectors: a stop-the-world mark-sweep collector, a stop-the-world copying collector, and an incremental copying collector. These representation predicates allow many implementation details to be hidden from the mutator, by using them as part of the GC interface I described in the previous chapter. I believe that representation predicates can be constructed for many other types of collectors, such as reference counting collectors, other incremental copying collectors, incremental mark-sweep collectors, and generational collectors, as these collectors do not seem to require more complex memory invariants than the Baker collector. In the next two chapters, I will show how these invariants are used, as part of my verification of the Cheney and Baker copying collectors.

Chapter 6

Cheney Collector Verification

6.1 Introduction

In this chapter, I discuss the Cheney copying garbage collection algorithm and implementation and then some issues involved with its verification. To recap, a *garbage collector* [Jones and Lins 1996] is a procedure that automatically frees objects that are no longer being used by a user program (the *mutator*). In a *tracing collector*, the set of unused objects is conservatively estimated as the set of objects reachable from a certain *root set*, which is, loosely speaking, the set of values or locations the mutator can access in a single instruction. Finally, a *copying collector* is a type of tracing collector that performs collection by copying all reachable objects, while updating all object pointers in the copies to point to the new version of each object. Once this is done, all of the old objects can be freed. The Cheney collector is a *stop-the-world collector*, which means that when collection begins the mutator does not execute again until the collection is finished. In a single threaded setting this means that when the collector is invoked it does not return until it has finished collecting. In a multithreaded setting this means that all mutator threads are paused while the

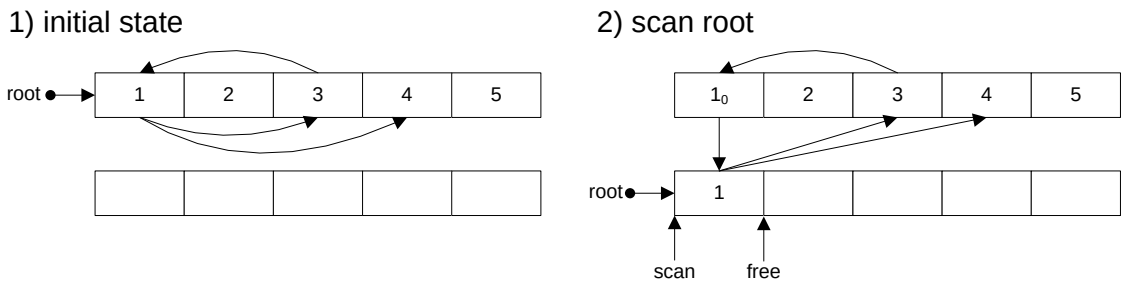

Figure 6.1: Example: Cheney collection (beginning)

collector is running.

In a scanning collector, the objects in various states of collection can be described using the *tricolor abstraction* [Dijkstra et al. 1978]. In this, each object has (conceptually) a color, either black, gray or white. Black and gray objects are not garbage. Black objects will not be examined again by the collector in the current collection cycle, while gray objects will be. Once a gray object has been visited, it becomes a black object. If the collector has not yet determined if an object is garbage or not, it is white. For a copying collector, it is also useful to talk about *mapped* objects. These are the old copies of the black and gray objects. In this chapter, I generally omit discussion of some of the fine details of the specifications regarding alignment.

One difficulty in designing a copying collector is figuring out how to store *forwarding pointers* that give the new location of a copied object. Another difficulty is avoiding copying objects that have already been copied. A *Cheney collector* [Cheney 1970] is a copying collector that avoids both of these problems by storing the forwarding pointer in the first field of the old copy of the object. In this way, the GC can examine the first field of a from-space object to determine if it is mapped or white: if the first field is a to-space pointer, it must be mapped, otherwise it must be white.

Now I will go through an example of what a Cheney collection looks like, at a

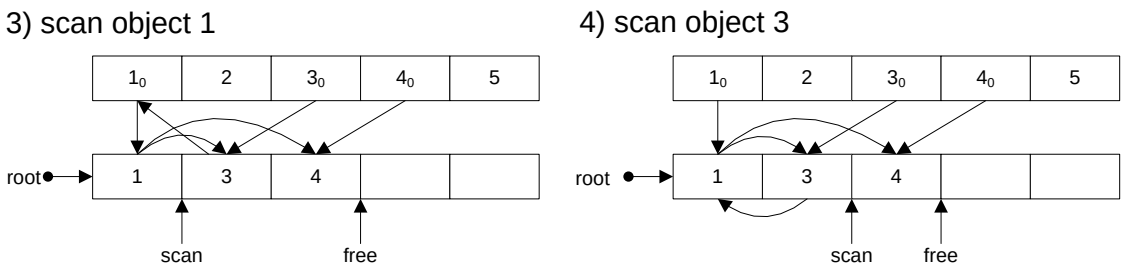

Figure 6.2: Example: Cheney collection (middle)

high level. Figure 6.1 shows the initial state and the first step. There are two semi-spaces, and the GC is going to copy objects from the top space to the bottom space. The root contains a pointer to object 1, which in turn points to objects 3 and 4, and object 3 points to object 1. Objects 2 and 5 are garbage.

To start the collection, the GC scans the root, looking for object pointers. The result of scanning the root is 2, which is the right side of Figure 6.1. The GC found that the root contained a pointer to object 1 and determined that 1 has not been copied yet. In response, it copied object 1, left a pointer to the new object 1 in the old object 1, and updated the root to the new object 1. There are two new pointers to the to-space. The scan pointer points to the first object the GC needs to scan, and the free pointer points to the next available free space. These both start pointing at the start of the to-space, but copying object 1 has caused the free pointer to be allocated. Objects between the scan pointer and free pointer are all gray objects.

The next two steps of the collector are shown in Figure 6.2. There is an object to scan, because the scan pointer is not equal to the free pointer. The GC scans object 1, looking for object pointers. It finds two, object 3 and 4. Again, the GC determines that neither has been copied yet, so it copies them to the to-space (starting at the free pointer), updating the old objects to point at the new, as before. After the GC has finished scanning object 1 it can advance the scan pointer. All objects from the

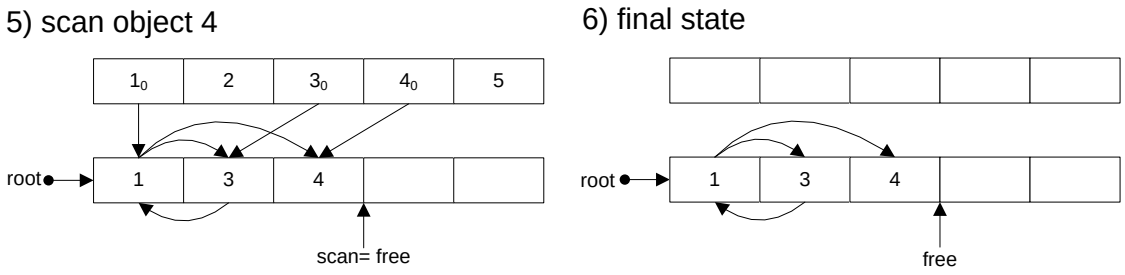

Figure 6.3: Example: Cheney collection (end)

start of the to-space to the scan pointer are black objects. The result of scanning object 1 is shown as step 3.

Again, the GC scans the object that the scan pointer points to, which is in this case 3. Scanning looks through object 3 for object pointers. It finds a pointer to object 1_0, the old copy of object 1. The collector examines object 1_0 and sees that it contains a pointer to object 1. Because object 1 is in the to-space and not the from-space, the GC can tell that object 1_0 has been copied to the location of object 1. The GC then replaces the pointer to 1_0 with a pointer to 1 and continues. The GC advances the scan pointer, but not the free pointer, because it has not copied any more objects. The result of scanning object 3 is shown as step 4.

Figure 6.3 shows the conclusion of the collection. First, the GC sees that there is still an object to be scanned, object 4, so it scans it. Object 4 does not contain any object pointers, so the GC does not have to change any values. As usual, the GC advances the scan pointer. The result of scanning object 4 is shown as step 5.

Objects 1, 3 and 4 are all black now, and contain no pointers to from-space objects. The scan pointer is equal to the free pointer, so there are no more gray objects. This means that the collector is done. New objects can be allocated starting at the same free pointer used during the collection. Removing the information from step 5 that is only needed for collection produces the picture in step 6. The GC is

$$
\begin{array}{l}
\mathit{rangeNat}(x, y) ::= \{k \mid x \leq k < y\} \\
\mathit{rangeAddr}(x, y) ::= \{x + 4k \mid 0 \leq k < (y - x)/4\} \\
\mathit{rangeObjs}(x, y) ::= \{x + 8k \mid 0 \leq k < (y - x)/8\} \\
\\
\mathit{mapHp}(S, \phi) ::= !(\forall x, y.\ \phi(x) = y \rightarrow x \in S) * \\
\quad \forall_* x \in S.\ \exists a.\ !(\phi(x) = a) * x \mapsto a, - \\
\\
\mathit{atomic}(x) ::= \mathit{odd}(x) \\
\mathit{notSpacePtr}(x, y, z) ::= \mathit{atomic}(x) \vee x \notin \mathit{rangeNat}(y, z) \\
\mathit{okFieldVal}(S, x) ::= \mathit{atomic}(x) \vee x \in S \\
\mathit{okField}(S, x) ::= \exists y.\ !(\mathit{okFieldVal}(S, y)) * x \mapsto y \\
\mathit{okObj}(S, x) ::= \mathit{okField}(S, x) * \mathit{okField}(S, x + 4) \\
\mathit{objHp}(S_0, S) ::= \forall_* x \in S.\ \mathit{okObj}(S_0, x)
\end{array}
$$

Figure 6.4: Generic specification definitions

now ready to return to the mutator.

In the remainder of the chapter, I discuss my mechanized verification of an implementation of the Cheney GC. I first describe some basic predicates that I use. I then break down the collector component by component, working my way outwards from the most basic building block: the scanning of a single field, the loop that scans all of the objects, then the whole collector (which is the loop plus some initialization code). After that, I discuss the three operations: allocation (which may invoke the collector), reading and writing. For each of these components, I give both pseudocode and assembly implementations, then the specification of interesting code blocks in that component. Finally, I give an overview of the verification that the assembly code matches the specification. All of the code, specifications and proofs described in this chapter have been mechanically checked in the Coq proof assistant.

6.2 Generic predicates and definitions

There are a number of generic definitions that I will need for the GC specifications. These are given in Figure 6.4. First are three finite sets of natural numbers. *rangeNat*(x, y) is the set of all natural numbers from x up to, but not including, y. *rangeAddr*(x, y) is the set of all word-aligned addresses in the range x to y, assuming that x is a word-aligned address. *rangeObjs*(x, y) is the set of pointers to all objects in the range x to y, assuming that all of the objects are pairs.

Next is a memory predicate to describe the part of the memory containing copied objects. *mapHp*(S, ϕ) holds on a memory if S is the domain of the finite map ϕ, and the memory contains a pair for every element of S, where the first element of the pair located at x is $\phi(x)$. In this way, the memory is a representation of ϕ: the GC can get the value of $\phi(x)$ by loading the value stored in the memory at x, if x is in S. This is the basic "typing" judgment for field values, when S is the set of valid objects.

Finally, there are predicates to describe valid objects. First I define predicates on values that may occur in an object field or a root. A value x is *atomic* (written *atomic*(x)) if x is odd. The oddness predicate is mutually inductively defined in the usual way in terms of an evenness predicate. A value x is not a pointer to the space from y to z, written *notSpacePtr*(x, y, z), if x is atomic or is not in the range from y to z. Somewhat similarly, a value x is a valid field value with respect to a set of objects S, written *okFieldVal*(S, x), if x is atomic or x is a member of S.

Now I can define some memory predicates to describe valid objects using separation logic. A memory is a valid field located at x, with respect to a set of objects S (written *okField*(S, x)), if the memory contains only a value y at location x, and y is a valid field value with respect to S. A memory is a valid object at location x with

$$\mathit{aligned8}(x, y) ::= \exists k.\ y = x + 8k$$

$$\mathit{buffer}(x, y) ::= \forall_* z \in \mathit{rangeAddr}(x, y).\ z \mapsto -$$

$$\mathit{calleeSavedOk}(\mathbb{S}, \mathbb{S}') ::= \forall r \in \mathit{calleeSaved}.\ \mathbb{S}(r) = \mathbb{S}'(r)$$

Figure 6.5: More generic specification definitions

respect to a set of objects S (written $\mathit{okObj}(S, x)$), if the memory is a valid field at x and $x + 4$, again with respect to S.

A valid object heap contains the objects in S, which point to objects in S_0, written $\mathit{objHp}(S_0, S)$, if the memory can be split into disjoint pieces such that every x in S has a valid object at x with respect to S_0. An object heap is *closed* if $\mathit{objHp}(S, S)$ holds: every field is either atomic or points to another object in the heap.

I define some other miscellaneous predicates in Figure 6.5. $\mathit{aligned8}(x, y)$ holds if the address y occurs 0 or more pairs after x. This predicate is used in various places to ensure that a slice of memory from x to y (excluding y) will not contain any fractional pairs. $\mathit{buffer}(x, y)$ is a memory predicate that holds on a memory that contains all of the addresses from x to y, not including y. $\mathit{calleeSavedOk}(\mathbb{S}, \mathbb{S}')$ holds if the values of all of the registers in *calleeSaved* (which is simply the set of standard callee saved registers for the MIPS machine) are the same in $\mathbb{S}$ and $\mathbb{S}'$.

In Figure 6.6, I give some general memory predicates for describing the copying and forwarding of objects. $\mathit{fieldRespMap}(\phi, \mathbb{M}_0, x)$ holds on a memory if the memory is a single cell at x that contains the forwarded version of the value at x in memory $\mathbb{M}_0$. The value is mapped using the function ϕ^*, which is the map ϕ extended with the identity function at atomic values, following Birkedal et al. [2004]. $\mathit{objRespMap}(\phi, \mathbb{M}_0, x)$ is similar, except that it holds on an entire object instead of a single field. These two predicates describe what happens to an object when it is

$$\mathit{fieldRespMap}(\phi, \mathbb{M}_0, x) ::= \exists v.\ !(\exists v_0.\ \mathbb{M}_0 \vdash x \mapsto v_0 * \mathtt{true} \wedge \phi^*(v_0) = v) * x \mapsto v$$
$$\mathit{objRespMap}(\phi, \mathbb{M}_0, x) ::= \mathit{fieldRespMap}(\phi, \mathbb{M}_0, x) * \mathit{fieldRespMap}(\phi, \mathbb{M}_0, x + 4)$$

$$\mathit{fieldCopied}(x, \mathbb{M}_0, y) ::= \exists v.\ !(\mathbb{M}_0 \vdash x \mapsto v * \mathtt{true}) * y \mapsto v$$
$$\mathit{objCopied}(x, \mathbb{M}_0, y) ::= \mathit{fieldCopied}(x, \mathbb{M}_0, y) * \mathit{fieldCopied}(x + 4, \mathbb{M}_0, y + 4)$$
$$\mathit{objCopiedMap}(\phi, \mathbb{M}_0, x) ::= \exists x_0.\ !(\phi(x_0) = x) * \mathit{objCopied}(x_0, \mathbb{M}_0, x)$$

$$\mathit{fieldIsFwded}(x_0, \phi, \mathbb{M}, x) ::= \exists v.\ !(\exists v_0.\ \mathbb{M} \vdash x_0 \mapsto v_0 * \mathtt{true} \wedge \phi^*(v_0) = v) * x \mapsto v$$
$$\begin{aligned}&\mathit{objIsFwded}(\phi, \phi', \mathbb{M}, x) ::= \\ &\quad \exists x_0.\ !(\phi'(x_0) = x) * \\ &\qquad \mathit{fieldIsFwded}(x_0, \phi \cup \phi', \mathbb{M}, x) * \mathit{fieldIsFwded}(x_0 + 4, \phi \cup \phi', \mathbb{M}, x + 4)\end{aligned}$$
$$\mathit{objIsFwded}_1(\phi, \mathbb{M}, x) ::= \mathit{objIsFwded}(\emptyset, \phi, \mathbb{M}, x)$$

Figure 6.6: Generic copying and forwarding predicates

scanned, when $\mathbb{M}_0$ is the memory before the scan: the fields of the object are forwarded using some map ϕ. Alternatively, they can be thought of as describing what happens to an object when it goes from being gray to being black.

fieldCopied$(x, \mathbb{M}_0, y)$ holds on a memory if it consists entirely of the value v at address y, where v is the value of the memory $\mathbb{M}_0$ at address x, and similarly for entire objects with the *objCopied* predicate. This describes what happens when an object that was located at address x in memory $\mathbb{M}_0$ is copied to address y in the current memory, or, in other words, what happens when an object goes from being white to being gray. The predicate *objCopiedMap*$(\phi, \mathbb{M}_0, x)$ is a specialized version of *objCopied* holds if the memory contains an object at x that has been copied from $\mathbb{M}_0$ at location x_0, where $\phi(x_0) = x$. This predicate describes what happens when an object is copied from the from-space to the to-space, when ϕ is the forwarding map.

fieldIsFwded is a combination of *fieldRespMap* and *fieldCopied*. The predicate *fieldIsFwded*$(x_0, \phi, \mathbb{M}, x)$ holds on a memory if that memory contains a field at x that is a forwarded version of the field at x_0 in memory $\mathbb{M}$. This describes what happens to the field of an object as it goes from being white to being black. *objIsFwded* is

$$
\begin{array}{l}
\mathit{isoState}_R(\mathbb{A}, \mathbb{A}') ::= \\
\quad \exists \mathit{objs}, \mathit{objs}', \phi. \\
\qquad \mathbb{A} \vdash \mathit{objHp}(\mathit{objs}, \mathit{objs}) \wedge \mathbb{A}' \vdash \mathit{objHp}(\mathit{objs}', \mathit{objs}') \wedge \\
\qquad \mathit{objs} \cong_\phi \mathit{objs}' \wedge \\
\qquad (\forall r \in R.\ \phi^*(\mathbb{S}(r)) = \mathbb{S}'(r)) \wedge \\
\qquad \mathbb{A}' \vdash \forall_* x \in \mathit{objs}'.\ \mathit{objIsFwded}_1(\phi, \mathbb{A}, x)
\end{array}
$$

Figure 6.7: State isomorphism

similarly a combination of *objRespMap* and *objCopiedMap*, with a twist: instead of using a single map to forward both the object location and the field values, the second argument ϕ' alone is used to forward the object location, while the combination $\phi \cup \phi'$ is used to forward the values of the fields. This allows me to reason about the movement of such forwarded objects using only ϕ'. $\mathit{objIsFwded}_1$ is a special case of *objIsFwded* where the first argument is fixed to the empty map $\emptyset$, which is useful when the fields of an object are forwarded using the same map as the object itself.

$S \cong_\phi S'$ holds if the finite map ϕ is an isomorphism from set S to set S'. For this to hold, the following must hold:

1. The domain of ϕ must be S, and the range of ϕ must be S'
2. ϕ must be *injective*: for all x, x' and y, if $\phi(x) = y$ and $\phi(x') = y$, then $x = x'$
3. ϕ must be *surjective*: for all $y \in S$, there must exist some x such that $\phi(x) = y$

I can now define what a state isomorphism is, in Figure 6.7. For a state $\mathbb{A}$ to be isomorphic to a state $\mathbb{A}'$ with respect to a set of root registers R, written $\mathit{isoState}_R(\mathbb{A}, \mathbb{A}')$, a number of things must hold. First, $\mathbb{A}$ must contain some set of objects *objs* and $\mathbb{A}'$ must contain some set of objects objs'. Next, there must be some some finite map ϕ that is an isomorphism from *objs* to objs'. ϕ is the isomorphism between the states. The final two lines of the definition check that the two parts

$$\frac{}{\mathbb{M} \vdash y \longrightarrow^* y} \quad \text{(Rch_Refl)}$$

$$\frac{\mathbb{M}(x) = x' \quad \mathbb{M} \vdash x' \longrightarrow^* y}{\mathbb{M} \vdash x \longrightarrow^* y} \quad \text{(Rch_Step1)}$$

$$\frac{\mathbb{M}(x+4) = x' \quad \mathbb{M} \vdash x' \longrightarrow^* y}{\mathbb{M} \vdash x \longrightarrow^* y} \quad \text{(Rch_Step2)}$$

$$\frac{x \in R \quad \mathbb{M} \vdash x \longrightarrow^* y}{\mathbb{M} \vdash R \longrightarrow^* y} \quad \text{(RootRch)}$$

Figure 6.8: Reachability predicates

of the state, register file and memory, respect the isomorphism. Mapping the value of each register in R in the initial state $\mathbb{A}$ using ϕ^* (which, as I have said, is equal to the identity map for odd values and ϕ for even values) should produce the value of that register in the final state $\mathbb{A}'$. Finally, the memory of the state $\mathbb{A}'$ should be such that all of the objects are objects from $\mathbb{A}$, copied and mapped according to ϕ. This ensures that all of the objects have been moved according to ϕ and had their fields updated. Thanks to the use of $*$, it also ensures that none of the objects are overlapping.

My definition of state isomorphism is similar to that of Birkedal et al. [2004], but their version of ϕ is an isomorphism between memory addresses instead of objects, and does not appear to be correct. An extended version of their paper changes their notion of state isomorphism [Torp-Smith et al. 2006], and my notion of state isomorphism looks similar to theirs, though mine is defined using separation logic predicates, which may make it easier to use.

Other predicates deals with reachability in the memory and are given in Figure 6.8. Reachability is an important notion for a tracing collector such as the Cheney collector. $\mathbb{M} \vdash x \longrightarrow^* y$ is intended to hold if a value y is reachable from a value x in memory $\mathbb{M}$. This only makes sense on values x that are valid field values

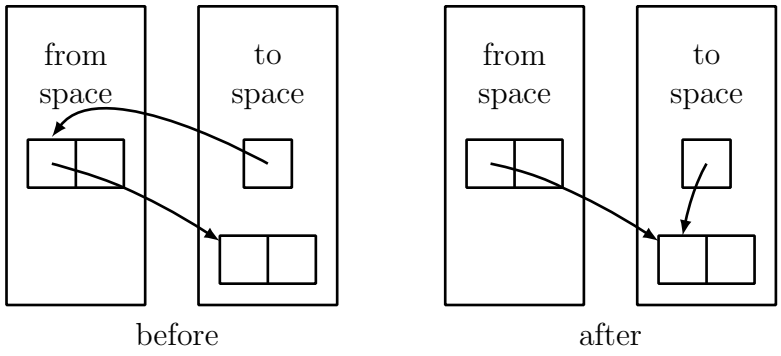

Figure 6.9: Field scanning: object already copied

for a valid object heap: the rules do not check, for instance, that x points to the beginning of an object. A value y is reachable from itself, or from a value x if the first or second field of x is some x' such that y is reachable from x'. Root reachability, written $\mathbb{M} \vdash R \longrightarrow^* x$, holds if x is reachable from some element of the set of values R in the memory $\mathbb{M}$.

6.3 Field scanning

The heart of the Cheney collector is field scanning. Field scanning updates the value of the field of an object which has previously been copied by the collector. If the field value is atomic, nothing needs to be done. Otherwise, the field contains an object pointer, which must be forwarded. How this pointer is forwarded depends on whether the object the pointer points to has already been copied.

If the first field of the object the field points to is a to-space pointer, then the object has already been copied, and that first field is the forwarding pointer for the object. This situation is illustrated in Figure 6.9. The lone square in the to-space is the field the GC is scanning.

If the first field of the object the field is pointing to is not a to-space pointer, then the object must be copied to the location of the free pointer. The new location of

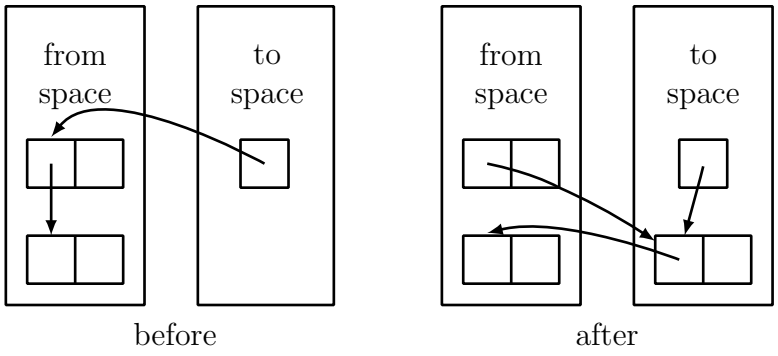

Figure 6.10: Field scanning: uncopied object

the object is stored in two places: the first field of the copied object and in the field being scanned. An example of this case is illustrated in Figure 6.10. As before, the single square is the field being scanned. The values are copied over to the new object without being updated, and thus the first field of the new to-space object points to a from-space object. When the new object is eventually scanned this pointer will be updated.

6.3.1 Implementation

Field scanning requires a few utility functions. Figure 6.11 has C-like pseudocode for these functions. This pseudocode uses three global variables that are object pointers: `free` (a pointer to the next available free object), and `toStart` and `toEnd`, which point to the beginning and end of to-space. The first utility function, `fwdObj`, copies the fields of `src` to the next available free object, and replaces the first field of `src` with the location of the new object, which is also returned. There is no bounds check, so this procedure cannot be called unless there is a free object available. `isAtom` returns true if and only if `x` is atomic (*i.e.,* is an odd value). `isNotToSpacePtr` returns true if and only if the argument `x` is not a to-space pointer.[1] A value is an

[1]This function is not the more sensible `isToSpacePtr` in a possibly misguided attempt to simplify the program's control flow.

```
void* fwdObj (void* src) {
  void* newObj = free;  // store value of free pointer
  newObj[0] = src[0];   // copy first field
  newObj[1] = src[1];   // copy second field
  *src = newObj;        // store forwarding pointer in old obj
  free += 2;
  return newObj;
}

bool isAtom (void* x) {
  return x & 1;
}

bool isNotToSpacePtr (void* x) {
  return isAtom(x) || (x < toStart) || (x >= toEnd);
}
```

Figure 6.11: Cheney utility function pseudocode

object pointer only if it is non-atomic and contained within the to-space.

The assembly implementation of these functions is given in Figure 6.12. Now I will explain the register naming scheme, and give machine registers corresponding to each name. Some register usage follows standard MIPS convention: register a0 is r4 and contains the argument to a procedure, v0 is r2 and holds the return value,[2] and

[2]In isNotToSpacePtr, it is also used as a temporary register.

```
FWDOBJ:                 ISNOTTOSPACEPTR:        RETURN:
lw rtemp,0(a0)          andi v0,a0,1            jr ra
sw rtemp,0(rfree)       bne v0, rzero, RETURN
lw rtemp,4(a0)          sltu v0,a0,rtoStart
sw rtemp,4(rfree)       bne v0, rzero, RETURN
sw rfree,0(a0)          addiu v0,a0,1
addiu v0,rfree,0        sltu v0,rtoEnd,v0
addiu rfree,rfree,8     bne v0, rzero, RETURN
jr ra                   addiu v0,rzero,0
                        jr ra
```

Figure 6.12: Cheney utility functions implementation

```
scanField (void* field) {
  fieldVal = *field;
  if (isAtom(fieldVal))
    return;
  fieldValField = *fieldVal;
  if (notToSpacePtr(fieldValField)) {
    *field = fwdObj(fieldVal);
  } else {
    *field = fieldValField;
  }
}
```

Figure 6.13: Cheney field scanning pseudocode

ra is r31 and holds the return pointer. rzero is r0, and is thus always has the value 0.

However, I generally use an unconventional register allocation scheme to avoid the need for stack allocation. Register rtemp is r6 and is used to hold temporary values. The register rfree is r7 and points to the first available free object, while rtoStart and rtoEnd (corresponding to r13 and r14) contain pointers to the start and end of to-space. Note that I do not actually implement `isAtom` as a separate function in assembly. Instead, I inline the single instruction needed to implement this procedure, a bitwise-and, where needed. Otherwise this is a straightforward translation of the pseudocode to assembly.

Now that I have defined the helper functions, I can define the actual field scanning operation. Pseudocode for `scanField` is given in Figure 6.13. `scanField` first loads the value of the field into `fieldVal`. If the field value is atomic, scanning is done. Otherwise, `fieldVal` must be an object pointer, so the procedure can load the value of the first field of that object into `fieldValField`. If the first field of the object is *not* a to-space pointer, then the object `fieldVal` has not yet been forwarded, so `scanField` forwards it using `fwdObj`, which, as described above, also stores a forwarding pointer. Otherwise the object must already have been been forwarded,

```
SCANFIELD:
lw rtemp1,0(a0)
andi rtemp,rtemp1,1
bne rtemp, rzero, RETURN
addiu rtemp0,a0,0
lw t2,0(rtemp1)
addiu t7,ra,0
addiu t4,rfree,0
addiu a0,t2,0
jal NOTTOSPACEPTR, SCANPOINTER

SCANPOINTER:
addiu rfree,t4,0
beq v0, rzero, SCANNOCOPY
addiu a0,rtemp1,0
jal FWDOBJ, SCANCOPIED

SCANCOPIED:
sw v0,0(rtemp0)
addiu ra,t7,0
jr ra

SCAN_NO_COPY:
sw t2,0(rtemp0)
addiu ra,t7,0
jr ra
```

Figure 6.14: Cheney field scanning implementation (assembly)

and furthermore the value of the first field (`fieldValField`) must be the forwarding pointer. In either case, `scanField` updates the original field once it have determined the new location of the object.

The assembly implementation of `scanField` is given in Figure 6.14. The unconventional call instruction the assembly machine uses necessitates breaking the procedure into a number of basic blocks. Registers rtemp0 (which is r24) and rtemp1 (which is r9) are additional temporary registers. The other registers either use the standard MIPS naming convention or the naming convention used by the field scanning utility functions.

6.3.2 Specification

The implementation of `scanField` is made up of a number of blocks of assembly code. For all of the blocks except `SCANFIELD`, I assign low-level specifications that are almost directly derived from the dynamic semantics of the assembly code: if a block loads from a register, that register must contain a pointer, and so on. This allows me to do all of the high-level reasoning in a single place.

For instance, see the specification for `SCAN_NO_COPY` in Figure 6.15. This code block, defined in Figure 6.14, stores the value of register t2 into the memory location

$$scanNoCopyPre(\mathbb{S}) ::= \exists v.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{rtemp0}) \mapsto v * \mathtt{true}$$

$$\begin{array}{l} scanNoCopyGuar(\mathbb{S}, \mathbb{S}') ::= \\ \quad (\forall v, A.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{rtemp0}) \mapsto v * A \rightarrow \\ \qquad \mathbb{S}' \vdash \mathbb{S}(\mathsf{rtemp0}) \mapsto \mathbb{S}(\mathsf{t2}) * A) \wedge \\ \quad \forall r.\ \mathbb{S}'(r) = \mathbb{S}\{\mathsf{ra} \rightsquigarrow \mathbb{S}(\mathsf{t7})\}(r) \end{array}$$

Figure 6.15: `SCAN_NO_COPY` specification

pointed to by register rtemp0, then copies t7 to ra and returns. The precondition simply states that memory must contain some value at the address contained in rtemp0, whereas the guarantee states that that location is overwritten with the value in t2 (while leaving the rest of memory alone), and that the register file is untouched, except for changing the value of ra to have the initial value of t7. This structure follows the specification style of Section 2.7. The specifications for `SCANPOINTER` and `SCANCOPIED` are similar.

I give the actual field scanning method `SCANFIELD` two specifications, one high-level and one low-level. The low-level specification is easier to verify with respect to the code, while the high-level specification is easier for code that calls `SCANFIELD` to use. I show that the high-level specification is weaker than the low-level specification, then verify the implementation of `scanField` against the low-level specification, implying that the implementation also matches the high-level specification. Separating these concerns simplifies the verification effort.

Valid field scanning state

The high and low level specifications are given in terms of a predicate *scanFieldStOk* that specifies certain basic relationships between the state and the various sets of colored objects. This predicate is defined in Figure 6.16. There are six arguments to this predicate. ϕ is the forwarding function that maps the old locations of objects to

$$
\begin{aligned}
&\mathit{scanFieldStOk}(\phi, W, M, F, A, \mathbb{S}) := \\
&\quad |W| \leq |F| \wedge \\
&\quad (W \cup M) \cap (\mathit{rangeNat}(\mathbb{S}(\mathsf{rtoStart}), \mathbb{S}(\mathsf{rtoEnd}))) = \emptyset \wedge \\
&\quad F = \mathit{rangeObjs}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{rtoEnd})) \wedge \\
&\quad (\forall x, y.\ \phi(x) = y \rightarrow \neg \mathit{notSpacePtr}(y, \mathbb{S}(\mathsf{rtoStart}), \mathbb{S}(\mathsf{rtoEnd}))) \wedge \\
&\quad \mathbb{S}(\mathsf{rtoStart}) \leq \mathbb{S}(\mathsf{rfree}) \leq \mathbb{S}(\mathsf{rtoEnd}) \wedge \\
&\quad \mathbb{S} \vdash \mathit{mapHp}(M, \phi) * \mathit{objHp}(W \cup M, W) * (\forall_* x \in F.\ x \mapsto -, -) * A
\end{aligned}
$$

Figure 6.16: Valid field scanning state

their new locations. ϕ is fairly unconstrained from the perspective of field scanning. The finite sets W, M and F are, respectively, the white objects, mapped objects and free objects. The memory predicate A describes the rest of memory, and $\mathbb{S}$ is the state on which the predicate must hold. Field scanning does not care about black or gray objects because it does not examine them.

Now I will break down the specification, line by line. The specification makes use of definitions given in Section 6.2. The first requirement is that the set of white objects, which have not been scanned yet, is smaller than the set of free objects left. This ensures that the GC can copy every white object without running out of space, allowing a call `fwdObj` without a bounds check. Next, the specification requires that the from-space objects (given by $W \cup M$) do not fall within the bounds of the to-space, which are stored in the registers $\mathsf{rtoStart}$ and rtoEnd. Again, this is not a very precise specification, but is enough to verify field scanning. Third, the specification requires that the set of free objects F is exactly the set of objects from $\mathbb{S}(\mathsf{rfree})$ to $\mathbb{S}(\mathsf{rtoEnd})$. In other words, the free objects make up the end of the to-space, starting at the value in rfree. This allows field scanning to safely allocate a new object at $\mathbb{S}(\mathsf{rfree})$, if F is not empty. Next, the specification requires that every value in the range of ϕ is non-atomic and falls within the bounds of the to-space. This is necessary to allow the GC to distinguish objects in W (which have a first field that is either

atomic or within the from-space) from objects in M (which have a first field that is within the to-space). Next, the specification enforces a basic ordering by requiring that the beginning of the to-space does not occur before the beginning of the free objects, which in turn does not occur before the end of the to-space.

Finally, the specification describes the shape of the memory of $\mathbb{S}$. Memory is divided into four disjoint parts. The first contains the finite map ϕ, which has a domain M. The second part contains the objects in W. Each of these objects points only to objects in M and W. The third part of memory contains the free objects in F, while the last part of memory satisfies the memory predicate A.

Precondition

The high and low level specifications share the same precondition, which is an instantiation of the *scanFieldStOk* predicate I just described, where the parameter A describing the remainder of memory specifies that register `a0` must contain a pointer to a field that is valid with respect to $W \cup M$. In other words, it must contain a pointer to a location in memory that is either atomic or a from-space pointer. The precondition is formally defined as follows:

$$\begin{aligned} &\mathit{scanFieldPre}(\mathbb{S}) := \exists \phi, W, M, F. \\ &\quad \mathit{scanFieldStOk}(\phi, W, M, F, (\mathit{okField}(W \cup M, \mathbb{S}(\texttt{a0})) * \texttt{true}), \mathbb{S}) \end{aligned}$$

Low level

The low-level guarantee has two cases: one for when an object is not copied, and one for when it is, corresponding to Figures 6.9 and 6.10, respectively. The formal specification is given in Figure 6.17. There are two different guarantees that describe the result, one for each possibility. The low-level field scanning guarantee,

$$
\begin{array}{l}
\mathit{objNotCopiedGuar}(\phi, W, M, F, A, \mathbb{S}, \mathbb{S}') ::= \\
\quad \mathbb{S}(\mathsf{rfree}) = \mathbb{S}'(\mathsf{rfree}) \wedge (\forall x \in W.\ \mathbb{S}' \vdash \mathit{objCopied}(x, \mathbb{S}, x) * \mathtt{true}) \wedge \\
\quad \mathit{scanFieldStOk}(\phi, W, M, F, (\mathit{fieldRespMap}(\phi, \mathbb{S}, \mathbb{S}(\mathsf{a0})) * A), \mathbb{S}') \\
\\
\mathit{objCopiedGuar}(\phi, W, M, F, A, \mathbb{S}, \mathbb{S}') ::= \\
\quad \mathbb{S}(\mathsf{rfree}) + 8 = \mathbb{S}'(\mathsf{rfree}) \wedge \\
\quad \exists v \in W.\ \ \mathbb{S} \vdash \mathbb{S}(\mathsf{a0}) \mapsto v * \mathtt{true} \wedge \\
\qquad (\forall x \in W - \{v\}.\ \mathbb{S}' \vdash \mathit{objCopied}(x, \mathbb{S}, x) * \mathtt{true}) \wedge \\
\qquad \mathit{scanFieldStOk}(\phi \cup \{v \rightsquigarrow \mathbb{S}(\mathsf{rfree})\}, W - \{v\}, M \cup \{v\}, F - \{\mathbb{S}(\mathsf{rfree})\}, \\
\qquad\quad (\mathbb{S}(\mathsf{a0}) \mapsto \mathbb{S}(\mathsf{rfree}) * \mathit{objCopied}(v, \mathbb{S}, \mathbb{S}(\mathsf{rfree})) * A), \mathbb{S}') \\
\\
\mathit{scanFieldGuar0}(\mathbb{S}, \mathbb{S}') := \\
\quad (\forall \phi, W, M, F, A.\ \mathit{scanFieldStOk}(\phi, W, M, F, (\mathit{okField}(W \cup M, \mathbb{S}(\mathsf{a0})) * A), \mathbb{S}) \rightarrow \\
\qquad \mathit{objNotCopiedGuar}(\phi, W, M, F, A, \mathbb{S}, \mathbb{S}') \vee \\
\qquad \mathit{objCopiedGuar}(\phi, W, M, F, A, \mathbb{S}, \mathbb{S}')) \wedge \\
\quad \forall r \notin \{\mathsf{rtemp}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{t2}, \mathsf{t4}, \mathsf{t7}, \mathsf{a0}, \mathsf{rfree}, \mathsf{v0}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 6.17: Low level scan field specification

scanFieldGuar0, allows any well-formed scan field state, where a0 points to a field that is valid with respect to the from-space objects. The final state is a disjunction of the two possible outcomes of the field scanning, plus a specification of the registers that have their values preserved.

In the first case, *objNotCopiedGuar*, no object has been copied. Instead, the field was just forwarded. The guarantee specifies that the value of the free pointer has not changed, indicating that no object has been newly allocated. Next it specifies that all of the white objects have the same values as they had before. This is crude, but it works. Finally, the specification shows that this is a well-formed scan field state. The map and the sets of white, map and free objects have not changed. The only change here is that the value in the field that a0 points to has been forwarded according to ϕ.

In the second case, *objCopiedGuar*, the field a0 points to contains an unforwarded object, so more work must be done than in the other case. First of all, the free pointer

$$
\begin{array}{l}
\mathit{scanFieldGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \phi, W, M, F, A. \\
\qquad \mathit{scanFieldStOk}(\phi, W, M, F, (\mathit{okField}(W \cup M, \mathbb{S}(\mathsf{a0})) * A), \mathbb{S}) \rightarrow \\
\qquad \exists \phi', W', M', F', G'. \\
\qquad\quad W = M' \cup W' \ \wedge \ F = G' \cup F' \ \wedge \ M' \cong_{\phi'} G' \ \wedge \\
\qquad\quad (\forall x \in G'.\ \mathbb{S}'(\mathbb{S}(\mathsf{a0})) = x) \ \wedge \\
\qquad\quad (\forall x \in W'.\ \mathbb{S}' \vdash \mathit{objCopied}(x, \mathbb{S}, x) * \mathtt{true}) \ \wedge \\
\qquad\quad \mathit{aligned8}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{rfree})) \ \wedge \\
\qquad\quad \mathit{scanFieldStOk}(\phi \cup \phi', W', M \cup M', F', \\
\qquad\qquad (\mathit{fieldRespMap}(\phi \cup \phi', \mathbb{S}, \mathbb{S}(\mathsf{a0})) * \\
\qquad\qquad (\forall_* x \in G'.\ \mathit{objCopiedMap}(\phi', \mathbb{S}, x)) * A), \mathbb{S}')) \ \wedge \\
\quad \forall r \notin \{\mathsf{rtemp}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{t2}, \mathsf{t4}, \mathsf{t7}, \mathsf{a0}, \mathsf{rfree}, \mathsf{v0}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 6.18: High level scan field specification

has been incremented by 8, because `scanField` has allocated a new object. Next, there is some value v, which is a white space object, that has been copied. This is the original value of the field being scanned, as indicated by the second line of the definition. The line after indicates, as in *objNotCopiedGuar*, that all white space objects, aside from v, have been copied. Next, the final state must be a well-formed scan field state. This time, however, many things have changed. First of all, the mapping from old objects to new objects has been extended by a map from v to $\mathbb{S}(\mathsf{rfree})$, because a new object has been allocated at the free pointer. Next, v is no longer a white space object, so it is removed from the set of white space objects and added to the set of mapped objects. $\mathbb{S}(\mathsf{rfree})$ must also be removed from the set of free objects. Finally, the field being forwarded now contains a pointer to the newly allocated copy and the newly allocated copy contains a copy of the object v.

High level

The high-level guarantee, given in Figure 6.18, takes the two cases of the low-level guarantee and combines them, by describing the behavior of the field scanning op-

eration at a more abstract level. As with the low-level guarantee, the high-level guarantee requires that the initial state is a well-formed scanning state, for some ϕ and sets of objects. In the final state, the initial set of white objects W is split into a new set of white objects W' and a new set of mapped objects M'. M' is made up of the objects that have been copied during this call to field scan (which will have either zero or one elements). Similarly, the set of free objects F is split into a new set of free objects F' and a set of gray objects G' that have been copied but not forwarded. Finally, there is a new map ϕ' that is an isomorphism from the newly mapped objects M' to the set of new gray objects G'.

The fourth line of the guarantee specifies that all components of G' are contained in the updated field. In other words, G is either empty or it contains the forwarded location of the initial field value. This will later allow me to show that all copied objects are reachable from the new root. Next, all of the objects in W' are unchanged from $\mathbb{S}$ to $\mathbb{S}'$, in the same way as in the low-level specification.

Finally, the state is again a valid scan field state. The arguments are similar to *objCopiedGuar*, although some of the details are hidden. The map in the new state is a combination of the old map ϕ and the new map ϕ'. The new set of white objects is W', while the new set of mapped objects is the union of the new and old mapped objects M and M'. The new set of free objects is F'. Finally, the initial function argument points to a forwarded version of the initial field value (having been forwarded with respect to $\phi \cup \phi'$), all of the objects in G have been copied to new locations that follow the map ϕ', and the extra part of memory, described by A, has not been altered.

6.3.3 Verification

Basic verification

Verifying `RETURN`, `SCAN_NO_COPY` and `SCANCOPIED` is fairly trivial, requiring only the most basic reasoning along the lines of the example presented in Chapter 2. `FWDOBJ` and `SCANPOINTER` are more difficult to verify, but only require manipulating separation logic predicates. Verifying `ISNOTTOSPACEPTR` is mainly a matter of checking every control path.

Verifying that *scanFieldGuar0* is stronger than *scanFieldGuar* requires considering the two cases of the former, then instantiating the various sets of objects in the latter. In the case of *objNotCopiedGuar*, no objects have been copied, so most of the sets of objects stay the same. More specifically, $M' = \emptyset$, $W' = W$, $F' = F$, $G = \emptyset$ and $\phi = \{\}$. In the case of *objCopiedGuar*, an object v has been copied, so v must be moved from the set of white objects to the set of mapped objects, and the location of the copy of v (which is $\mathbb{S}(\mathsf{rfree})$) must be added to the map and the set of two-space objects. Concretely, $M' = \{v\}$, $W' = W - \{v\}$, $F' = F - \mathbb{S}(\mathsf{rfree})$, $G = \{\mathbb{S}(\mathsf{rfree})\}$ and $\phi = \{v \rightsquigarrow \mathbb{S}(\mathsf{rfree})\}$. Once the existential variables have been instantiated the existential in each case, verifying the strength of *scanFieldGuar0* is simply a matter of going through and proving that all of the various sets match up appropriately with each other and with memory.

Verifying scanfield

To verify that `SCANFIELD` implements the low-level specification, the two code paths through the block must be considered. I will call the value of the field being scanned x.

Case. x is odd. `SCANFIELD` exits immediately. This case is fairly straightforward, and uses the *objNotCopiedGuar* case of *scanFieldGuar0*.

Case. x is even. x must either be a white or mapped object.

Subcase. x is a white object (*e.g.,* is a member of W). This is the case corresponding to Figure 6.10. Therefore, the first field of that object will not be a to-space pointer, so `NOTTOSPACEPTR` will return 1. For the call to `SCANPOINTER` to be safe, the value being scanned must point to a pair and rfree must point to an object. The former is easy to show, because it is a white object. The latter will follow from showing that F is not empty. W is not empty, because x is a member of W. This, combined with the fact that $|W| \leq |F|$, implies that F is not empty.

For the guarantee, this subcase will use the *objCopiedGuar* case of *scanFieldGuar0*. Verifying that this case holds involves a lot of manipulation to relate the initial and final states. Because the structure of the guarantee requires that it hold for *any* set of white objects, it must be shown that because x is part of *some* set of white objects in a well-formed field scanning state, it is part of *any* set of white objects that can be contained in a well-formed field scanning state. x must either be atomic or part of the set of white or mapped objects. It cannot be atomic, because it is even. It cannot be part of a set of mapped objects, because its first field is not a to-space pointer. The definition of a to-space pointer is based on the values of registers, and thus does not depend on W or M. Note that it cannot be shown that any two sets of white objects will be the same for a given state, because the set of white objects is only loosely specified. To reason about memory, the object x is split out from W, updated, then added to M, and the copy of x is shown to be properly copied. This requires using a lot of standard lemmas about the iterated separating conjunction operator.

```
void cheneyLoop () {
  while (scan != free) {
    scanField(scan);
    scanField(scan + 1);
    scan += 2;
  }
}
```

Figure 6.19: Cheney loop pseudocode

Subcase. x is a mapped object, and thus a member of M. This is the case corresponding to Figure 6.9. In this case, the object has already been copied, so all the method must do is use the forwarding pointer to update the field. It can be shown that `NOTTOSPACEPTR` must return 0, implying that `SCANPOINTER` will take the correct branch. As in the case where x is atomic, this case uses the *objNotCopiedGuar* case of *scanFieldGuar0*. Similarly to the previous case, it must be shown that x is a member of any set of mapped objects. x can never be a white object, because its first field is a to-space pointer, and the from-space and to-space are disjoint. Otherwise, the proof is mainly a matter of manipulating various separation logic predicates.

6.4 The loop

The loop of the Cheney collector scans gray objects using `scanField` until no gray objects remain. The gray objects start at the scan pointer and end at the free pointer, so there are no gray objects when these two pointers are equal.

6.4.1 Implementation

A pseudocode implementation of the Cheney collector loop is given in Figure 6.19. `scan` points to the first gray object, while `free` points to the first free object, which is also the next object past the last gray object. Thus when `scan` is equal to `free`

```
CHENEYLOOP:                          CHENEYLOOP2:                        CHENEYRETURN:
beq rscan, rfree, CHENEYRETURN       addiu a0,rscan,4                    addiu ra,raSave,0
addiu a0,rscan,0                     addiu rscan,rscan,8                 jr ra
jal SCANFIELD, CHENEYLOOP2           jal SCANFIELD, CHENEYLOOP
```

Figure 6.20: Cheney loop implementation

$$
\begin{aligned}
&\mathit{cheneyLoopStOk}(\phi, G, W, M, F, A, \mathbb{S}) ::= \\
&\quad G = \mathit{rangeObjs}(\mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree})) \wedge \\
&\quad \mathbb{S}(\mathsf{rtoStart}) \leq \mathbb{S}(\mathsf{rscan}) \wedge \mathit{aligned8}(\mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree})) \wedge \\
&\quad \mathit{scanFieldStOk}(\phi, W, M, F, A, \mathbb{S})
\end{aligned}
$$

Figure 6.21: Cheney loop state formation

there are no more gray objects, and collection is finished. The loop body scans the two fields of the first gray object, then moves to the next object and repeats. Each invocation of `scanField` may cause additional objects to be copied, incrementing `free`.

The actual assembly implementation for the loop for the Cheney collector is given in Figure 6.20. This is a direct translation of the pseudocode, with the addition of some register wrangling and code block splitting to accommodate the machine's unusual `jal` instruction. The register rscan (which is r11) holds the value of `scan`, while raSave (which is r3) is used to hold the value of the return pointer, and is saved before the loop.

6.4.2 Specification

The first part of defining the specification for the Cheney collector loop is to define what it means for a state to be well-formed. This predicate, defined in Figure 6.21, builds on the definition of *scanFieldStOk* by adding additional constraints. The design principle here is that each block of code should only know enough about the

$$
\begin{array}{l}
\mathit{cheneyLoopPre}(\mathbb{S}) ::= \\
\quad \exists \phi, G, W, M, F.\ \mathit{cheneyLoopStOk}(\phi, G, W, M, F, (\mathit{objHp}(W \cup M, G) * \mathtt{true}), \mathbb{S}) \\
\\
\mathit{cheneyLoopGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \phi, G, W, M, F, A. \\
\qquad \mathit{cheneyLoopStOk}(\phi, G, W, M, F, (\mathit{objHp}(W \cup M, G) * A), \mathbb{S}) \rightarrow \\
\qquad \exists \phi', W', M', F', G'. \\
\qquad\quad W = M' \cup W' \ \wedge\ F = G' \cup F' \ \wedge\ M' \cong_{\phi'} G' \ \wedge \\
\qquad\quad (\forall x' \in G'.\ \exists x \in G.\ \mathbb{S}' \vdash x \longrightarrow^* x') \ \wedge \\
\qquad\quad \mathit{aligned8}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{rfree})) \ \wedge \\
\qquad\quad \mathit{cheneyLoopStOk}(\phi \cup \phi', \emptyset, W', M \cup M', F', \\
\qquad\qquad ((\forall_* x \in G.\ \mathit{objRespMap}(\phi \cup \phi', \mathbb{S}, x)) * \\
\qquad\qquad\ (\forall_* x \in G'.\ \mathit{objIsFwded}(\phi, \phi', \mathbb{S}, x)) * A), \mathbb{S}')) \\
\mathit{land} \\
\quad (\forall r \notin \{\mathsf{ra}, \mathsf{rscan}, \mathsf{rtemp}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{t2}, \mathsf{t4}, \mathsf{t7}, \mathsf{a0}, \mathsf{rfree}, \mathsf{v0}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)) \ \wedge \\
\quad \mathbb{S}'(\mathsf{ra}) = \mathbb{S}(\mathsf{raSave})
\end{array}
$$

Figure 6.22: Cheney loop specification

state to verify that block of code, to simplify verification. There will be further constraints for the entire collector. This predicate has three parts. First, the set of gray objects must span the part of memory from the scan pointer to the free pointer, which matches up with the definition of the gray objects I discussed above. This specifies where the gray objects are and allows me to show that the set of gray objects is empty when $\mathbb{S}(\mathsf{rscan}) = \mathbb{S}(\mathsf{rfree})$. Next the scan pointer must occur after the start of to-space. This, in combination with the fact that $\mathbb{S}(\mathsf{rfree}) \leq \mathbb{S}(\mathsf{rtoEnd})$ (which is implied by *scanFieldStOk*) ensures that every gray object is in the to-space. Finally, *scanFieldStOk* holds on the current state. As a reminder, the A argument to *scanFieldStOk* is the specification for the rest of memory.

Now I can define the precondition and the guarantee of the Cheney collector loop, given in Figure 6.22. The precondition starts off simply enough, only requiring that the state is a well-formed Cheney loop state. For the additional memory specification, memory must contain all of the gray objects G, which must only refer to white and

mapped objects, reflecting the fact that gray objects are copies of white objects. The $*$`true` is there because other things are allowed to be present in memory.

The loop guarantee is not a loop invariant. Instead, it describes what will happen during all of the rest of the iterations of the loop. Thus the final state it describes will be fully garbage collected. There are two parts to the loop guarantee. The first part describes what happens to memory, while the second describes what happens to the registers. The second part is the last two lines of the guarantee. The very last line specifies that the loop restores `ra` before it returns, and the second to last line specifies the registers that are unchanged by the loop.

For memory, the guarantee applies to any well-formed Cheney loop state. The part to the left of the implication is the same as the precondition of the loop, except that the rest of memory is described by some A instead of by `true`. This will allow the "post-condition" to show that the rest of memory is unchanged. Similarly to *scanfieldGuar*, there are a number of new sets of objects after the loop is executed. ϕ' is an isomorphism from the set of objects copied during the loop (M') to the new location of these objects (given by G'). In addition, there is the set W' of white objects that are unreachable from the initial set of gray objects G. There is also the final set of free objects F'. The guarantee also formally specifies the intuition that the set of white objects W is made up of the white objects copied in the loop M' and the unreachable white objects W', and likewise for the free objects F. Next, every object copied during the loop (as I said, those objects in G') must be reachable from the initial set of gray objects G. This will eventually imply that only objects reachable from the root are copied by the collector, which means that the GC collects all garbage (for a trace-based notion of garbage).

Finally, the resulting state $\mathbb{S}'$ must be a well-formed Cheney loop state. The isomorphism for the final state is a combination of the old and new mappings (ϕ and

$$cheneyReturnPre(\mathbb{S}) ::= \text{True}$$

$$cheneyReturnGuar(\mathbb{S}, \mathbb{S}') ::= \mathbb{S}' = \mathbb{S}\{\mathsf{ra} \rightsquigarrow \mathbb{S}(\mathsf{raSave})\}$$

$$ch2Step(\mathbb{S}) ::= \mathbb{S}\{\mathsf{a0} \rightsquigarrow \mathbb{S}(\mathsf{rscan}) + 4\}\{\mathsf{rscan} \rightsquigarrow \mathbb{S}(\mathsf{rscan}) + 8\}\{\mathsf{ra} \rightsquigarrow \texttt{CHENEYLOOP}\}$$

$$\begin{aligned} &cheneyLoop2Pre(\mathbb{S}) ::= \\ &\quad scanFieldPre(ch2Step(\mathbb{S})) \wedge \\ &\quad \forall \mathbb{S}'.\ scanFieldGuar(ch2Step(\mathbb{S}), \mathbb{S}') \rightarrow cheneyLoopPre(\mathbb{S}') \end{aligned}$$

$$\begin{aligned} &cheneyLoop2Guar(\mathbb{S}, \mathbb{S}'') ::= \\ &\quad \exists \mathbb{S}'.\ scanFieldGuar(ch2Step(\mathbb{S}), \mathbb{S}') \wedge cheneyLoopGuar(\mathbb{S}', \mathbb{S}'') \end{aligned}$$

Figure 6.23: Miscellaneous Cheney loop specifications

ϕ'). The set of gray objects in the final state is the empty set $\emptyset$, because the collector does not exit the loop until the set of gray objects is empty. The set of white objects in the final state is just W', while the set of mapped objects is a combination of the domains of the old and new set of mapped objects (M and M'). The new set of free objects is F'. The remainder of memory can be described in three parts. First, there is the part of memory containing the objects in G. These objects have had all of their fields forwarded using the final mapping ($\phi \cup \phi'$). This corresponds to the part of the object heap containing G in the precondition. Next is the portion of memory containing the newly copied objects in G'. In the initial state $\mathbb{S}$, this was part of the free space, but now it contains forwarded copies of the objects in M'. The predicates for these two parts of memory are defined in Section 6.2. Finally, the initial excess portion of memory described by A is unchanged.

The rest of the specifications for the loop, given in Figure 6.23, are relatively simple. The specifications for `CHENEYRETURN` and `CHENEYLOOP2` are, like the bulk of the blocks in the field scanner, low level and defined fairly directly in terms of the dynamic semantics. `CHENEYRETURN` just restores `ra`. For `CHENEYLOOP2`, I define a

function *ch2Step* that describes the behavior of the body of the block. The precondition simply requires that the function `SCANFIELD` can be called after running the loop, and can call `CHENEYLOOP` after returning from `SCANFIELD`. The guarantee is the juxtaposition of the guarantees of these two specifications.

6.4.3 Verification

The verification of `CHENEYLOOP2` and `CHENEYRETURN` are simple, due to the low-level nature of their specifications. The verification of `CHENEYLOOP`, on the other hand, is fairly complex, taking a little more than 700 lines of Coq proofs.

As a reminder, the loop is made up of these 3 steps:

1. Leave the loop if all gray objects have been scanned.
2. Scan the first gray object.
   (a) Scan the first field.
   (b) Scan the second field.
3. Return to the top of the loop.

While I have given a guarantee for scanning a single field, I have not yet given a guarantee that describes the scanning of an entire object, which is step two. I now rectify this by defining a new guarantee *cheneyScanObjGuar*, given in Figure 6.24, that describes the behavior of scanning an entire object. This guarantee is very similar to the field scanning guarantee, as one might expect. Instead of requiring a valid field and guaranteeing that that field is properly scanned, this guarantee requires a valid object and guarantees that the object is properly scanned. In addition, it uses the higher-level *cheneyLoopStOk* instead of *scanFieldStOk*.

$$
\begin{array}{l}
\mathit{cheneyScanObjGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \phi, G, W, M, F, A. \\
\qquad \mathit{cheneyLoopStOk}(\phi, G, W, M, F, (\mathit{okObj}(W \cup M, \mathbb{S}(\mathsf{rscan})) * A), \mathbb{S}) \rightarrow \\
\qquad \exists M', W', F', G', \phi'. \\
\qquad\quad W = M' \cup W' \ \wedge \ F = G' \cup F' \ \wedge \ M' \cong_{\phi'} G' \ \wedge \\
\qquad\quad (\forall x \in G'.\ \mathbb{S}'(\mathbb{S}(\mathsf{rscan})) = x \vee \mathbb{S}'(\mathbb{S}(\mathsf{rscan}) + 4) = x) \ \wedge \\
\qquad\quad \mathit{aligned8}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{rfree})) \ \wedge \\
\qquad\quad (\forall x \in W'.\ \mathbb{S}' \vdash \mathit{objCopied}(x, \mathbb{S}, x) * \mathtt{true}) \ \wedge \\
\qquad\quad \mathit{cheneyLoopStOk}(\phi \cup \phi', G - \mathbb{S}(\mathsf{rscan}) \cup G', W', M \cup M', F', \\
\qquad\qquad (\mathit{objRespMap}(\phi \cup \phi', \mathbb{S}, \mathbb{S}(\mathsf{rscan})) * \\
\qquad\qquad\quad (\forall_* x \in G'.\ \mathit{objCopiedMap}(\phi', \mathbb{S}, x)) * A), \mathbb{S}')) \ \wedge \\
\qquad (\forall r \notin \{\mathsf{rscan}, \mathsf{ra}, \mathsf{rtemp}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{t2}, \mathsf{t4}, \mathsf{t7}, \mathsf{a0}, \mathsf{rfree}, \mathsf{v0}\}.\ \mathbb{S}(r) = \mathbb{S}(r')) \ \wedge \\
\qquad \mathbb{S}'(\mathsf{rscan}) = \mathbb{S}(\mathsf{rscan}) + 8 \wedge \mathbb{S}'(\mathsf{ra}) = \mathtt{CHENEYLOOP}
\end{array}
$$

Figure 6.24: Cheney object scanning guarantee

With this new guarantee defined, the verification of the loop can be broken into into a number of lemmas:

1. If the GC exited the loop, executing `CHENEYRETURN` will satisfy the loop guarantee.

2. If the GC did not exit the loop and the loop precondition holds, then it is safe to scan the first field of the current gray object.

3. If the GC did not exit the loop, and the loop precondition holds, and the GC scanned the first field of the current object, then it is safe to scan the second field of the current gray object.

4. If the GC did not exit the loop, then after scanning the first and second fields of the current gray object the GC will have scanned the current gray object.

5. If the GC did not exit the loop, and the loop precondition holds, and the GC scanned the current object, then it is safe to reenter the loop.

```
void cheneyEnter(void* root) {
  swap(toStart, fromStart);
  swap(toEnd, fromEnd);
  free = toStart;
  scan = toStart;
  scanField(root);
  cheneyLoop();
}
```

Figure 6.25: Cheney entry pseudocode

6. Finally, there is what can be thought of as the inductive case: if the GC did not exit the loop, and it scanned the current gray object then satisfied the loop guarantee, then the entire execution satisfies the loop guarantee. More formally, if $\mathbb{S}(\mathsf{rscan}) \neq \mathbb{S}(\mathsf{rfree})$, $\mathit{cheneyScanObjGuar}(\mathbb{S}, \mathbb{S}')$ and $\mathit{cheneyLoopGuar}(\mathbb{S}', \mathbb{S}'')$, then $\mathit{cheneyLoopGuar}(\mathbb{S}, \mathbb{S}'')$.

After proving these lemmas, the Cheney loop can be easily verified.

6.5 The collector

Finally, I combine the loop together with some initialization code to form the Cheney collector. The initialization code loads some values into registers, swaps the semi-spaces, then scans the root.

6.5.1 Implementation

The pseudocode for the Cheney entry point is given in Figure 6.25. First the procedure swaps the from- and to-spaces, then initializes the free and scan pointers. No objects have been copied, allocated or scanned, so both pointers are set to the start of the to-space. Next the procedure scans the root using the standard `scanField` method. If the root is an object pointer, scanning it will cause an object to be

```
CHENEYENTER:
// swap spaces, init rtoStart and rtoEnd
lw rtoStart,0(gcInfo)
lw rtemp,8(gcInfo)
sw rtemp,0(gcInfo)
sw rtoStart,8(gcInfo)
lw rtoEnd,4(gcInfo)
lw rtemp,12(gcInfo)
sw rtemp,4(gcInfo)
// init rfree and rscan
addiu rfree,rtoStart,0
addiu rscan,rtoStart,0
// scan the root, then enter the loop
addiu raSave,ra,0
sw root,12(gcInfo)
addiu a0,gcInfo,12
jal SCANFIELD, CHENEYLOOPHEADER

CHENEYLOOPHEADER:
lw root,12(gcInfo)
sw rtoEnd,12(gcInfo)
j CHENEYLOOP
```

Figure 6.26: Cheney entry implementation

copied, incrementing `free`. After this, the procedure calls the Cheney loop, where all reachable objects will be scanned.

The assembly implementation, given in Figure 6.26, is more complex. The first complication is that in order to minimize the register overhead while the mutator is running the ends of each semi-space are stored in memory, at an address contained in register gcInfo, which is r23. This array contains, in order, the beginning and end of the from-space, then the beginning and end of the to-space. The initial sequence of loads and stores initializes rtoStart and rtoEnd and swaps all of the various bounds in the GC information record, except for the end of the to-space. The record element that contains the end of the to-space is left uninitialized to allow it to be used to scan the root. After the loads and stores, the free and scan pointers are initialized, as in the pseudocode. Next, the value of ra is saved and copy the root from register root (r8) to memory (where the end of the to-space is normally stored), then scan the root and go to the loop header. The loop header cleans up after the initialization

$$
\begin{array}{l}
\mathit{cheneyLoopHeaderPre}(\mathbb{S}) ::= \\
\quad \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 12 \mapsto - * (\lambda \mathbb{M}.\ \mathit{cheneyLoopPre}((\mathbb{M}, \mathbb{S}))) \\
\\
\mathit{cheneyLoopHeaderGuar}(\mathbb{S}, \mathbb{S}'') ::= \\
\quad (\forall x, A.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 12 \mapsto x * A \rightarrow \\
\qquad \exists \mathbb{S}'.\ \mathbb{S}' \vdash \mathbb{S}(\mathsf{gcInfo}) + 12 \mapsto \mathbb{S}(\mathsf{rtoEnd}) * A \wedge \\
\qquad\quad (\forall r.\ \mathbb{S}'(r) = (\mathbb{S}\{\mathsf{root} \rightsquigarrow x\})(r)) \wedge \\
\qquad\quad \mathit{cheneyLoopGuar}(\mathbb{S}', \mathbb{S}'')) \wedge \\
\quad (\forall r \notin \{\mathsf{root}, \mathsf{ra}, \mathsf{rscan}, \mathsf{rtemp}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{t2}, \mathsf{t4}, \mathsf{t7}, \mathsf{a0}, \mathsf{rfree}, \mathsf{v0}\}. \\
\qquad \mathbb{S}(r) = \mathbb{S}''(r)) \wedge \\
\quad \mathbb{S}''(\mathsf{ra}) = \mathbb{S}(\mathsf{raSave})
\end{array}
$$

Figure 6.27: Cheney loop header specifications

code by loading the root back into the root register and saving the new end of the to-space to memory. After that, it enters the loop, which is described in Section 6.4.

6.5.2 Specification

The specification for `CHENEYLOOPHEADER` is given in Figure 6.27. This is yet another spec derived fairly directly from the implementation. For the loop header block to be safely called, memory must contain a value at an offset of 12 from gcInfo, and the remainder of memory must satisfy the loop precondition. As far as the behavior of this block goes, it loads the value at offset 12 from gcInfo into root, stores the value of rtoEnd, then goes into the loop. The state right before the loop is some $\mathbb{S}'$.

The specification of the Cheney collector entry is defined in terms of a predicate *cheneyStOk* that describes a well-formed Cheney collector state. This predicate is given in Figure 6.28. This is simpler than the state specifications I have shown so far (used inside of the collector loop) because it does not have to deal with a partially copied heap. This is beginning to look more like the representation predicate for the Cheney collector described in Section 5.3.

$$
\begin{array}{l}
\mathit{cheneyStOk}(\mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathit{free}, \mathit{root}, \mathit{gcInfo}, \mathbb{M}) ::= \\
\quad !(\mathit{aligned8}(\mathit{toSt}, \mathit{free}) \wedge \mathit{aligned8}(\mathit{free}, \mathit{toEnd}) \wedge \\
\qquad \mathit{okFieldVal}(\mathit{objs}, \mathit{root}) \wedge \\
\qquad \mathit{frEnd} - \mathit{frSt} = \mathit{toEnd} - \mathit{toSt} \wedge \mathit{frSt} \leq \mathit{frEnd}) * \\
\quad (\mathit{objHp}(\mathit{objs}, \mathit{objs}) \wedge \mathit{eq}\ \mathbb{M}) * \mathit{buffer}(\mathit{free}, \mathit{toEnd}) * \\
\quad \mathit{buffer}(\mathit{frSt}, \mathit{frEnd}) * \mathit{gcInfo} \mapsto \mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd} \\
\\
\quad \text{where } \mathit{objs} = \mathit{rangeObjs}(\mathit{toSt}, \mathit{free})
\end{array}
$$

$$
\mathit{okObjHp}(V, \mathit{objs}) ::= \ !(\forall v \in V.\ \mathit{okFieldVal}(\mathit{objs}, v)) \ * \ \mathit{objHp}(\mathit{objs}, \mathit{objs})
$$

$$
\mathit{minObjHp}(V, \mathit{objs}) ::= \quad (\lambda \mathbb{M}.\ \forall x \in \mathit{objs}.\ \mathbb{M} \vdash V \longrightarrow^* x) \ \wedge \ \mathit{okObjHp}(V, \mathit{objs})
$$

$$
\begin{array}{ll}
\phi^*(x) = x & \text{if } x \text{ is odd} \\
\phi^*(x) = \phi(x) & \text{otherwise}
\end{array}
$$

Figure 6.28: Auxiliary Cheney definitions

The first four arguments are the bounds of the semi-spaces. The names are given with respect to the end of the previous collection: the from-space is empty and the to-space contains the allocated objects. *free* is the value of the free pointer, while *root* is the value of the root. *gcInfo* is a pointer to the location in memory that the auxiliary GC data is stored. Finally, the argument $\mathbb{M}$ is the object heap containing the allocated objects. This will be useful for reasoning about the behavior of the collector. The set *objs* (defined to be $\mathit{rangeObjs}(\mathit{toSt}, \mathit{free})$) is the set of allocated objects, which starts at the beginning of the to-space and ends at the free pointer.

First I will describe the "pure" parts of this predicate that do not involve memory. In order to safely perform a Cheney collection on a state, the start of the to-space must be object-aligned with the start of the free space, and the free space with the end of the to-space. These requirements prevent fractional objects. The root must be a valid field value, either atomic or an allocated object. Additionally, the from-space and the to-space must be the same size, and the end of the from space must be after

the beginning. These two requirements ensure there will be enough space to copy every object.

Now I will describe the memory part of this predicate. The first part of memory contains the objects. This must be a valid object heap, and all object pointers must point to objects in this object heap. This part of memory must also be equal to $\mathbb{M}$. (The $\wedge$ in this portion of the specification is a separation logic conjunction, not a conventional one: $A \wedge B$ holds on a memory if that memory satisfies both A and B.) After that, the memory must contain the unallocated portion of the to-space, ranging from the free pointer to the end of the to-space. Next, memory must contain the fallow semi-space and *gcInfo* must point to a list of the various bounds.

There are also a few simpler definitions defined in Figure 6.28. Memory is closed with respect to a set of values V and contains objects *objs*, written $okObjHp(V, objs)$, if all of the object pointers in the object heap point to objects in the heap, and all of the values in V are valid object pointers with respect to *objs*. In other words, starting from the set of values V and tracing object pointers will not escape the heap. $minObjHp(V, objs)$ holds if memory contains exactly the set of objects reachable from V. For this to be true, all objects in *objs* must be reachable from the set of root values V and the object heap must be closed with respect to V and *objs*. Finally, the map ϕ^*, taken from Birkedal et al. [2004], maps old heap values to new heap values when ϕ is the map from the old locations of objects to their new locations. This is the identity mapping for odd (*i.e.,* atomic) values and the same as ϕ for all other values.

I can now define the top level specification of the Cheney collector, given in Figure 6.29. For the precondition, the state must be a well-formed Cheney state. The space bounds are stored in memory (as *cheneyStOk* requires), while the free pointer, root and GC information pointer must be in their respective registers. As

$$
\begin{array}{l}
\mathit{cheneyEnterPre}(\mathbb{S}) ::= \exists \mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathbb{M}. \\
\quad \mathbb{S} \vdash \mathit{cheneyStOk}(\mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{root}), \mathbb{S}(\mathsf{gcInfo}), \mathbb{M}) * \mathtt{true} \\
\\
\mathit{cheneyEnterGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathit{toSt}, \mathit{toEnd}, \mathit{frSt}, \mathit{frEnd}, \mathbb{M}, A. \\
\qquad \mathbb{S} \vdash \mathit{cheneyStOk}(\mathit{toSt}, \mathit{toEnd}, \mathit{frSt}, \mathit{frEnd}, \mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{root}), \mathbb{S}(\mathsf{gcInfo}), \mathbb{M}) * \\
\qquad\quad A \rightarrow \\
\qquad \exists \phi, \mathit{objs}, \mathbb{M}'. \\
\qquad\quad \mathbb{S}' \vdash \mathit{cheneyStOk}(\mathit{frSt}, \mathit{frEnd}, \mathit{toSt}, \mathit{toEnd}, \mathbb{S}'(\mathsf{rfree}), \mathbb{S}'(\mathsf{root}), \mathbb{S}'(\mathsf{gcInfo}), \mathbb{M}') * \\
\qquad\qquad A \wedge \\
\qquad\quad \mathbb{M} \vdash \mathit{minObjHp}(\{\mathbb{S}(\mathsf{root})\}, \mathit{objs}) * \mathtt{true} \wedge \\
\qquad\quad \mathit{objs} \cong_{\phi} \mathit{rangeObjs}(\mathit{toSt}, \mathbb{S}'(\mathsf{rfree})) \wedge \\
\qquad\quad \phi^*(\mathbb{S}(\mathsf{root})) = \mathbb{S}'(\mathsf{root}) \wedge \\
\qquad\quad \mathbb{M}' \vdash (\forall_* x \in \mathit{rangeObjs}(\mathit{toSt}, \mathbb{S}'(\mathsf{rfree})).\ \mathit{objIsFwded}_1(\phi, \mathbb{M}, x))) \wedge \\
\quad \mathit{calleeSavedOk}(\mathbb{S}, \mathbb{S}') \wedge \\
\quad \mathbb{S}(\mathsf{raSave2}) = \mathbb{S}'(\mathsf{raSave2})
\end{array}
$$

Figure 6.29: Cheney entry specifications

usual, any additional data is allowed in memory.

The guarantee has six auxiliary variables. The first four are the bounds of the semi-spaces. Notice that unlike the precondition, these are ordered from the perspective of the upcoming collection: allocated objects are in the from-space, and will be copied to the to-space. $\mathbb{M}$ is the portion of memory containing the initial set of allocated objects, and as usual A describes the portion of memory that the collector does not know or care about. As in the precondition, the state must initially be a well-formed Cheney collector state, with the free pointer, root and GC information pointers stored in registers.

After the collection has finished, three existential variables are needed to describe the final state. ϕ is the isomorphism from the initial set of reachable objects to the final set of reachable objects. *objs* is the set of reachable objects in the initial state. Notice that this is not part of the precondition. This is because the collector calculates this set, so the caller of the collector does not need to. Finally, $\mathbb{M}'$ is the

part of the final state that contains allocated objects.

Now I will describe the specification of the final state. First, the collector ensures that the final state is a well-formed Cheney collector state. This is important, because it allows the mutator to call the collector again later. Birkedal et al. [2004] do not establish this in their Cheney verification. Notice that the bounds of the from-space are now those of the to-space and vice versa, precisely as one would expect. Other than that, this establishes that the new values of the free, root and GC information pointers are still stored in the appropriate register, and that the new object heap is represented by $\mathbb{M}'$. This line also shows that the collector has not touched the rest of memory, which can still be described by A.

The next line specifies that *objs* is actually the set of reachable objects in the initial state, using the minimum object heap predicate defined in Figure 6.28. The line after that establishes that ϕ is an isomorphism from the initial set of reachable objects to the final set of allocated objects. While this is primarily used to establish that the collector is safe, it also demonstrates that the garbage collector does not copy any unreachable objects.

The next two lines establish that the collection respected the isomorphism ϕ, using some of the definitions from Figure 6.28. First, the root must be forwarded via the mapping ϕ^*. Next, all of the allocated objects in the to-space must be forwarded versions of objects in $\mathbb{M}$, the initial allocated block of memory, using the object forwarding predicate defined in Figure 6.6.

The final two lines of the specification establish that the collector preserves all of the callee saved registers, along with register raSave2, which is register r25.

6.5.3 Verification

Verification of the collector begins simply enough, by stepping through the various loads, stores and additions that comprise the initialization portion of the allocator. As these memory operations involve constant offsets from the GC information pointer, they are easily shown to be safe.

Valid scanField state

After the initialization, it must be safe to call `scanField`. For this to hold, memory must be a valid `scanField` state. This field scanning state has an empty mapping from old objects to new objects, because no objects have been copied.

First, it must be shown that the number of allocated objects in the from-space is less than or equal to the number of objects that can be stored in the to-space, which follows from the fact that the two semi-spaces are the same size.

Next, it must be established that the set of allocated objects (which lie in the range from *frSt* to the initial free pointer) is disjoint from the set of natural numbers starting from *toSt* and ending just before *toEnd*. To do this, I first establish that memory contains two disjoint buffers (one from *frSt* to the initial free pointer, and one for the to-space) and call this proof H_0. This is easy to establish using separation logic. Now I break into a number of subcases. First, either *toSt* is a pointer or the to-space is empty. In the latter case, the current subgoal holds immediately. Similarly, either *frSt* is a pointer or the set of allocated objects is empty. As before, in the latter case, disjointedness holds immediately. Both *toSt* and *frSt* must then be valid pointers. Using this fact and H_0, it can be established that the set of allocated objects is disjoint from the set of *addresses* in the to-space. Finally, I use a lemma that shows that if two sets of valid contiguous addresses are disjoint, then one of the sets of valid addresses is disjoint with the *entire range* of the other set, thus showing

this subgoal.

The last part of establishing a valid `scanField` state is showing that memory is well-formed. This is pretty direct. The trickiest part is showing that memory contains a well-formed object map (see Figure 6.4), but the mapping is currently empty, so the empty memory satisfies this.

Into the loop

Given the well-formed `scanField` state, it is trivial to show that it is safe to call `scanField`. The next task is to reason about the state after the GC has returned from `scanField`, when it calls `cheneyLoopHeader`. The specification of that state is derived by combining the `scanFieldOk` already established and the guarantee for the `scanField` procedure, then applying the standard suite of state simplification tactics. After doing that, there must be some (possibly empty) initial set of gray objects, and a mapping ϕ from the initial copy of these gray objects to their current location.

Before I can establish that the loop precondition is satisfied, I need to show some lemmas. First, memory contains all of the initial objects and, separately, the various garbage collector information pointers. This implies a lemma *infoNotInObjs*, which states that none of the addresses the GC writes to during the initialization section are part of any of the objects in memory. Also, the addresses the GC writes to during initialization must be valid pointers.

Now it can be shown that it is safe to enter the loop. To do this, the various sets and maps must be selected. The gray (G), white (W), mapped (M) and free objects (F) are those returned by the call to `scanField`. The initial set of black objects is empty, as no object have been scanned yet. Again some register simplifying is done.

It must be shown that G is the set of contiguous objects ranging from the scan

pointer to the new free pointer. This is true because the to-space is $G \cup F$, F is the set of objects from the free pointer to the end of the to-space, and G and F are disjoint, and a few minor requirements for the relevant pointers. The rest of the side conditions are trivially true, leaving only the need to show that memory is well-formed. The only difficulty there is showing that the gray objects G are well-formed with respect to the from-space objects $W \cup M$. Each object in G is a copy of a from-space object in the state before the call to `scanField`. Using *infoNotInObjs* it can shown that the objects in G are also copies of from-space objects in the initial state, which are well-formed with respect to $W \cup M$, so this part of the precondition holds.

Through the loop

I have now shown that it is safe to enter the loop. The next goal is to show that composing the behavior seen so far with the guarantee of the loop implies that the entire function satisfies the loop entry guarantee.

The first task is to establish that the four semi-space bounds of the already-shown precondition are the same as those of any well-formed Cheney state, because they are stored at specific offsets in memory from gcInfo. Once this is done, it can be established that the state before the loop entry is in fact a well-formed Cheney state. This proof is actually almost the same as the proof that it was safe to enter the loop. The main difference is that the remainder of memory is described by A instead of by `true`. This essentially requires reasoning from the beginning of the procedure and showing what happens when the initialization code and `scanField` are executed. A better program logic might avoid this duplication.

After this is done, there is a predicate describing the state after the rest of the program, including the loop, has executed. As usual, simplification must be per-

formed on these predicates. For instance, if a register r has not changed from the initial state $\mathbb{S}_0$ to the final state $\mathbb{S}$, any instances of the value of register r in the final state, written $\mathbb{S}(r)$, need to be replaced by $\mathbb{S}_0(r)$, to simplify reasoning.

At this point, after the completion of the loop, there are several different sets of objects: M (from-space objects copied during the scanning of the root), W (the rest of the from-space objects), M' (from-space objects copied during the loop, and a subset of W), G (to-space objects copied during root scanning), G' (to-space objects copied during root scanning and W' (from-space objects that have not been copied). G and G' must be handled separately because objects G are scanned in the loop, while objects on G' are both copied and scanned in the loop. Two additional sets of objects are F (free to-space objects after the root scanning) and F' (free to-space objects after the loop). There are also two finite maps, ϕ and ϕ', which describe the forwarding of objects during root scanning and the loop, respectively.

It must be shown that in the final state all objects copied during collection are reachable from the root in zero or more steps. There are two types of these objects. The first is the set of gray objects G created by scanning the root. The guarantee of field scanning, shown in Figure 6.18, ensures that all of these objects are reachable in a single step from the root. The second set of objects are those that were copied during the loop. The guarantee of the loop itself, shown in Figure 6.22, ensures that these objects are reachable from the root in one or more steps.

There are several sets of objects that are disjoint: W and M, M and M', G and G', G' and F', G and F, and M' and W'. Disjointedness of these sets must be proved before I do any further manipulation of the memory predicate, because disjointedness information is stored *implicitly* using separation logic predicates. A lemma that states that if $\mathbb{M} \vDash (\forall_* a \in A.\ a \mapsto - * \texttt{true}) * (\forall_* b \in B.\ b \mapsto - * \texttt{true})$, then A and B are disjoint. This lemma is shown by contradiction. If there was an x such that $x \in A$

and $x \in B$, then it would be possible to shown that $\mathbb{M} \vDash x \mapsto - * x \mapsto - * \texttt{true}$, which, as I argued in Section 2.5.2, is impossible.

That done, I extract the portion of memory containing the reachable to-space objects ($G \cup G'$) in order to show that the final state is a well-formed Cheney state. I use an *extraction lemma*, which states that if $\mathbb{M} \vDash A * B$, then there exists an $\mathbb{M}'$ such that $\mathbb{M} \vDash eq\ \mathbb{M}' * B$ and $\mathbb{M}' \vDash A$. This follows from the definition of $*$.

Once the portion of memory containing the new objects has been extracted, I can show that these objects have been copied from objects in the initial state, then had their fields forwarded, respecting the map $\phi \cup \phi'$. The objects in G have had their fields updated respecting the map by the actions of the loop. To do this, I use a sort of transitivity lemma stating that objects copied from a state $\mathbb{S}$ to a state $\mathbb{S}'$ then scanned from state $\mathbb{S}'$ to a state $\mathbb{S}''$ have also been copied then scanned from state $\mathbb{S}$ to $\mathbb{S}''$. The objects in G' have been copied and scanned from the state at the start of the loop, so I use a lemma that states that if an object is unchanged from state $\mathbb{S}$ to $\mathbb{S}'$, and copied and scanned to $\mathbb{S}''$, then it is also copied and scanned from $\mathbb{S}$ to $\mathbb{S}''$.

Next, I want to show the precise values of G and G'. G is the set of objects starting from the start of the to-space to the value of the free pointer before the program enters the loop. G' is the set of objects starting from the value of the free space pointer before the loop and ending at the value of the free space pointer at the end of the current procedure. From this, I can conclude that $G \cup G'$ is the set of objects starting from the beginning of to-space to the final value of the free pointer.

With these preliminaries out of the way, I can show the conclusion of the Cheney entry guarantee of Figure 6.29. There are three existentials that must be instantiated. The first is the mapping from old objects to new objects, which is $\phi \cup \phi'$. The second is the set of reachable objects in the from-space, which have been calculated by

running the collector. This set is $M \cup M'$. Finally, there is the memory that contains the reachable objects in the final state, which is the memory I extracted earlier that contains the objects in $G \cup G'$.

When showing that the final memory is well-formed, there are three subgoals. First, it must be shown that the final object heap is well-formed. These objects ($G \cup G'$) have been copied and forwarded using a valid isomorphism, so I can use a lemma to show that these objects must also be well-formed. Second, I need to show that there is a buffer starting at the free pointer and going to the end of the to-space. This follows from the fact that F' is the set of objects in that range, which contains a whole number of objects. Third, I must show that there is a buffer for the from-space. This is a matter of showing that the combination of the free space from the from-space along with the from-space objects, both copied and uncopied, cover the entire from-space.

The rest of the goals relate to showing that the collection is correct: that the set of objects copied from the from-space is precisely the set of reachable objects, that the old and new objects are related by an isomorphism, and that the root and the copied objects respect that isomorphism. With this in hand, only the first subgoal presents real difficulty.

For this, I must first show that the initial heap contains the objects in $M \cup M'$, and that these objects only contain pointers to other objects in $M \cup M'$. Showing this relies on a lemma that states that if a memory $\mathbb{M}$ contains a set of well-formed objects S (that contain pointers to *any* other set of objects), and there is another memory $\mathbb{M}'$ that contains a set of objects S' and is a forwarded version some memory that is a superset of $\mathbb{M}$ using an isomorphism ϕ from S to S', then the objects in $\mathbb{M}$ must only point to objects in S. This is the case because all pointers in the fields of $\mathbb{M}'$ have been forwarded using ϕ (so they must be in the range of ϕ), and the domain

of ϕ is S. For similar reasons, any object pointers in the root must be members of $M \cup M'$.

Next I must show that all objects $m \in M \cup M'$ are reachable from the root. Because there is an isomorphism from $M \cup M'$ to $G \cup G'$, for each such object m there must exist an object $g \in G \cup G'$. As I have previously shown, g must be reachable from the final root. By induction on the path from the final root to g, there must be a path from the initial root to m, because all of the gray objects are forwarded versions of the objects $M \cup M'$. I initially show that m must be reachable in the entire memory, then show that it must also be reachable in the part of the heap containing only the objects of $M \cup M'$.

After showing that the appropriate registers have been preserved, I have verified that the Cheney entry block matches its specification.

Loop header

The last remaining piece of the collector proper that must be verified is the loop header. The specification of the loop header is fairly low level, so this does not present any difficulties. First I show that the state still satisfies the loop precondition after the loop is run. The block does not do much so this is easy to do. Then I show that running the loop header then running the loop satisfies the loop header guarantee. Of course, the loop header guarantee pretty much says that the loop header is run, then the loop is run, so the verification is easily finished.

6.6 The allocator

The allocator is the first component of the full Cheney collector implementation that the mutator directly interacts with. The allocator tries to allocate an object in the

```
void* alloc (void* root) {
  if (free == toEnd) {
    cheneyEnter(root);
    if (free == toEnd) {
      while (1) {}
    }
  }

  free[0] = NULL;
  free[1] = NULL;
  newObj = free;
  free += 2;
  return newObj;

}
```

Figure 6.30: Cheney allocator pseudocode

free space. A pseudocode implementation is given in Figure 6.30. If the free space is empty, the allocator calls the collector. If even after performing a collection there is still no free space, the allocator goes into an infinite loop. A more realistic collector would request more space from the operating system in this case. Once the free space is known to be non-empty, the allocator performs allocation by incrementing the free pointer and initializing the fields of the new object, then returning the address of the new object.

Instead of having an explicit infinite loop in the case where there is no more memory, even after a collection, it would also be possible to remove the second if statement, and change the first if statement to a while loop. This would also produce an infinite loop but it would be harder to verify than the code I use because it would require proving that `cheneyEnter` can be safely called repeatedly.

```
CHENEYALLOC:
addiu raSave2,ra,0
lw rfree,16(gcInfo)
lw rtemp,12(gcInfo)
bne rfree, rtemp, CHENEYDOALLOC
jal CHENEYENTER, CHENEYPOSTGC

CHENEYPOSTGC:
lw rtemp,12(gcInfo)
bne rfree, rtemp, CHENEYDOALLOC
j INFLOOP

CHENEYDOALLOC:
addiu v0,rzero,1
sw v0,0(rfree)
sw v0,4(rfree)
addiu v0,rfree,0
addiu rfree,rfree,8
sw rfree,16(gcInfo)
addiu ra,raSave2,0
jr ra

INFLOOP:
j INFLOOP
```

Figure 6.31: Cheney allocator implementation

6.6.1 Implementation

The assembly implementation of the allocator is given in Figure 6.31. This is a fairly direct translation of the pseudocode. The main differences are that the control flow is somewhat contorted, and that instead of storing the free pointer and end of the to-space pointer in a global variable, they are stored in memory.

6.6.2 Specification

The specification for CHENEYALLOC is the allocator specification from the collector interface given in Chapter 4. The specifications for the other blocks are given in Figure 6.32 CHENEYDOALLOC can be called whenever rfree points to a pair, and there is a place to store the new value of the free pointer. When it is called, it initializes the fields of the new object to NULL, stores the new value of the free pointer, while leaving the rest of memory alone. It also stores the location of the new object in v0, updates the free pointer, and restores the return pointer.

CHENEYPOSTGC is safe to call when there is a value *toEnd* stored at an offset of 12 from the GC information pointer. If *toEnd* is not equal to the free pointer, it must be safe to call the allocator. The guarantee ensures that if the block returns then

$$cheneyDoAllocPre(\mathbb{S}) ::= \ \mathbb{S} \vdash \mathbb{S}(\mathsf{rfree}) \mapsto -, - \ * \ \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto - \ * \ \mathtt{true}$$

$$\begin{aligned}
&cheneyDoAllocGuar(\mathbb{S}, \mathbb{S}') ::= \\
&\quad (\forall A.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{rfree}) \mapsto -, - \ * \ \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto - \ * \ A \rightarrow \\
&\qquad \mathbb{S}' \vdash \mathbb{S}'(\mathsf{v0}) \mapsto \mathrm{NULL}, \mathrm{NULL} \ * \ \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto \mathbb{S}(\mathsf{rfree}) + 8 \ * \ A) \wedge \\
&\quad \forall r.\ \mathbb{S}'(r) = (\mathbb{S}\{\mathsf{v0} \rightsquigarrow \mathbb{S}(\mathsf{rfree})\}\{\mathsf{rfree} \rightsquigarrow \mathbb{S}(\mathsf{rfree}) + 8\}\{\mathsf{ra} \rightsquigarrow \mathbb{S}(\mathsf{raSave2})\})(r))
\end{aligned}$$

$$\begin{aligned}
&cheneyPostGCPre(\mathbb{S}) ::= \\
&\quad \exists toEnd.\ \ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 12 \mapsto toEnd * \mathtt{true} \wedge \\
&\qquad (\mathbb{S}(\mathsf{rfree}) \neq toEnd \rightarrow cheneyDoAllocPre(\mathbb{S}))
\end{aligned}$$

$$\begin{aligned}
&cheneyPostGCGuar(\mathbb{S}, \mathbb{S}') ::= \\
&\quad \forall toEnd.\ \ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 12 \mapsto toEnd * \mathtt{true} \rightarrow \\
&\qquad \mathbb{S}(\mathsf{rfree}) \neq toEnd \quad \wedge \quad cheneyDoAllocGuar(\mathbb{S}\{\mathsf{rtemp} \rightsquigarrow toEnd\}, \mathbb{S}')
\end{aligned}$$

$$infLoopPre(\mathbb{S}) ::= \mathrm{True}$$

$$infLoopGuar(\mathbb{S}, \mathbb{S}') ::= \mathrm{False}$$

Figure 6.32: Cheney allocator, miscellaneous specifications

the free pointer and *toEnd* are not equal, and that `CHENEYDOALLOC` has successfully run, from a state where the value of rtemp has changed.

`INFLOOP` is always safe to call, and anything can be assumed after a call to it, because it never returns.

6.6.3 Verification

Verification of the allocator requires verifying the 5 blocks of assembly. As before, the verifications of blocks besides `CHENEYALLOC` are straightforward, because I use low-level specifications that closely map their behavior. `INFLOOP` just calls itself, so it is trivial to verify. For `CHENEYPOSTGC`, there are two cases, one for each branch. If the free pointer is not equal to the end of to-space, the precondition guarantees that the allocator can safely call `CHENEYDOALLOC`. The guarantee for this case is trivially

established, as it only requires that the free pointer and the end of the to-space are not equal, and that `CHENEYDOALLOC` has been run. In the other case, where the collector has failed to free any space, the allocator simply calls the infinite loop, and thus this case is trivial to establish. The safety of `CHENEYDOALLOC` is easy to establish, as the precondition explicitly requires the presence of the memory locations accessed during the block. Showing the guarantee only requires stepping through the block in a similar way.

Now it only remains to verify the initial allocator block. This is the first block I have described that must deal with the high level garbage collector interface. While all previous blocks only had to deal with the concrete state, the allocator must also reason about the abstract state.

Once past the initial set of loads, which are readily verified, there are two cases to consider. In the first case, there is enough space to allocate a new object. In the second case, there is not.

Case: enough space to allocate

In the first case, the free space pointer is not equal to the end of the to-space. From this, it can be inferred that the free pointer is an element of the set of free space objects. This lets me show that there is a pair starting at the free pointer, which is the hard part of showing that it is safe to call `CHENEYDOALLOC`.

For the guarantee, the first step is to relate the precondition to the antecedent of the guarantee. For instance, that the free pointers are the same in both. This is established using the memory.

In the consequent, I first have to select what the abstract state after collection is, and what the concrete value of the object being allocated is. Because the allocator is not performing a collection, the abstract state after the "collection" is the same

as initial abstract state, which is some $\mathbb{A}$. The object being allocated is located at the current value of the free pointer.

After that, I show that the GC guarantee is respected. To do this, I use a lemma that states that for all $\mathbb{A}$ and R, $gcStep(\mathbb{A}, \mathbb{A}, R)$ holds, which in turn depends on a lemma that a well-formed object state is isomorphic to itself.

Now I must show that the final state is well-formed. This requires applying the guarantee of `CHENEYDOALLOC`, which in turn requires splitting out the free object from the memory of the initial state. This can be done because I have already shown that the free pointer is an element of the free block. Now there are more existentials to instantiate: the ends of the semi-spaces and the value of the free pointer. As the allocator has not done a collection, the semi-spaces stay the same, but the free space pointer has increased by 8. After this is done, there are a number of side conditions that must be checked, regarding alignment of the various pointers and the validity of the roots. These are all straightforward.

Finally for this case, I must show that memory is well-formed. The difficulty here is showing that adding the newly allocated object to the memory results in a well-formed memory equal to the extension of the abstract memory with a new object.

To show that the new memory is well-formed, I apply the object heap introduction lemma, which states that one can add a well-formed object to a well-formed object heap, assuming that both have the same set of assumptions. I must also use an object heap weakening lemma, because the initial object heap does not contain any pointers to the newly allocated object. The new object is well-formed because the allocator has initialized its fields to NULL.

To show that the new abstract memory is equal to the initial abstract memory extended with a new object, I use a lemma (proved by Chunxiao Lin [McCreight

et al. 2007]) that states that if $\mathbb{M} \vdash x \mapsto v * eq\ \mathbb{M}'$ then $\mathbb{M} \vdash eq\ (\mathbb{M}'\{x \rightsquigarrow v\})$. Applying that lemma twice solves this subgoal.

The last major part of this case is to show that the allocated object is fresh. To do this, I use a lemma (also proved by Chunxiao Lin [McCreight et al. 2007]) that states that if $\mathbb{M} \vdash A$ and x is a valid pointer not in the domain of $\mathbb{M}$, then $\mathbb{M}\{x \rightsquigarrow v\} \vdash x \mapsto v * A$. Applying this lemma twice solves the subgoal. It can be shown that neither field of the new object is already in the abstract heap because the abstract heap is part of the concrete memory, which contains (separately) both fields.

Case: not enough space to allocate

First I verify that it is safe to call the collector. This is not difficult, as I only have to show that the few register operations the allocator has performed do not interfere with the well-formedness of the state. Next I have to show that it is safe to call `CHENEYPOSTGC` after the collector has been run. To do this, I only need to show that memory contains the free pointer at the correct address, and that if the free pointer is not at the end of the to-space it points to an object, which is similar to reasoning I have already done.

Finally, I must show that the combination of the GC followed by `CHENEYPOSTGC` implies the entire allocator guarantee. To do this, I first have to step through the guarantees for the two functions that are called to get to a description of the final state.

After this, I instantiate the existentials for the abstract state and the address of the newly allocated object. The memory of the abstract state is the object heap that was the result of performing a garbage collection, while the register file of the abstract state is the register file of the final concrete state. The object being allocated

is the object at the free pointer in the final state.

The first major task in this case of the proof is to show that the garbage collector respects the GC step guarantee. First I show that the minimum set of reachable objects assumed in the GC step guarantee must be the same as the minimum set of reachable objects computed by the GC, using a lemma: every object reachable in one set must be reachable in the other, so they must be equivalent. The heap containing the reachable objects in the final state is going to be the heap containing all of the objects in the final state, because the final state contains only reachable objects. After that, the major remaining difficulty is showing that if objects in a heap $\mathbb{M}''$ are forwarded versions of objects from $\mathbb{M}$, and there exists an $\mathbb{M}'$ that is a subheap of $\mathbb{M}$ containing all of the objects in the domain of the isomorphism used to forward the objects, then $\mathbb{M}''$ contains forwarded copies of objects from $\mathbb{M}'$. In other words, if $\mathbb{M}''$ contains some objects that were forwarded from $\mathbb{M}$ using an isomorphism ϕ, and a heap $\mathbb{M}' \subseteq \mathbb{M}$ contains all of the objects in the domain of ϕ, then $\mathbb{M}''$ contains objects that were forwarded from $\mathbb{M}'$ using ϕ, because every object in $\mathbb{M}$ must have a corresponding object in $\mathbb{M}'$.

The rest of the tasks in the proof involve showing that the final state is well-formed, and that the objects are fresh. The proof of this is similar or identical to the case of the proof where the collector is not invoked, described in Section 6.6.3.

6.7 Reading and writing

The final parts of the Cheney collector implementation are the read and write barriers. The Cheney collector does not require any additional work to be done when reading or writing, so the main task is to verify that these operations preserve the high-level interface.

```
void* read (void* root, int k) {
  return root[k];
}

void write (void* root, int k, void* x) {
  root[k] = x;
}
```

Figure 6.33: Cheney read and write barrier pseudocode

$\texttt{CHENEYREAD}_k$:	$\texttt{CHENEYWRITE}_k$:
lw v0,k(root)	sw a0,k(root)
jr ra	jr ra

Figure 6.34: Cheney read and write barriers

6.7.1 Implementation

The pseudocode implementations for the read and write barriers are given in Figure 6.33. A read loads the designated field of the root, while a write stores a value to the field. In order for these operations to succeed, `root` must be an object pointer, and k must be either 0 or 1, as each object is a pair. Additionally, in the write barrier x must be a valid field value, or the collector may crash when it is next invoked.

The assembly implementations, given in Figure 6.34, are similar. Each barrier is implemented as a function call for simplicity. It would also be possible to develop a framework for reasoning about assembly macros, then implement and verify each barrier as an assembly macro. To simplify the implementation, there is one read and one write operation for each field. Thus each assembly implementation is parameterized by an offset k, which will be either 0 or 4.

6.7.2 Specification

As with the allocator, the specifications for the read and write operations are given in Chapter 4. Informally, the read requires that the initial state contains the representation of an abstract state $(\mathbb{A}, R)$ where root is a root ($\mathsf{root} \in R$) and $\mathbb{A}(\mathsf{root})$ is a valid pointer, and guarantees that the final state contains the representation of the initial abstract state transformed by the GC step guarantee, except that v0 will now be a root, containing the value of the appropriate field of root. The write requires the same thing, and that a0 is a root, so that it may be safely stored. It guarantees that the initial abstract state, transformed by the GC step guarantee and with the appropriate field of root replaced with the abstract value of a0, is represented in the final concrete state.

6.7.3 Verification

Showing the safety of both the read and the write is not hard because the abstract state is a subset of the concrete state, and the precondition implies that the register being read from or written to is a pointer. The difficulty is in showing that the abstraction has been preserved.

Read

First I have to show that the GC step relation holds. This is straightforward, as no part of the original abstract state is changed by a read. Next, I must show that the implementation of a read preserves the abstraction. The main difficulty is to show that the value that has been loaded is a valid root. This is true because the loaded value came from a field of a valid object. Finally it must be shown that the barrier has loaded the correct abstract value, which is easy to do because there is a

one-to-one correspondence between abstract and concrete values.

Write

The abstract state does not change across the GC step, because no collection work is done in the write barrier, so I use a standard lemma that proves that the GC step relation is reflexive. To show that memory is well-formed, I first prove that the root is non-atomic, by contradiction. After this is done, I can pull out the object being written to from the abstract memory, which in turn lets me show that the field being written to is in the domain of the abstract memory, which is a portion of the concrete memory. Then I relate the write to the abstract memory to the write to the concrete memory by applying a lemma which states that if $\mathbb{M} \vdash eq\ \mathbb{M}' * A$ and $x \in \mathrm{dom}(\mathbb{M}')$, then $\mathbb{M}\{x \rightsquigarrow y\} \vdash eq\ (\mathbb{M}'\{x \rightsquigarrow y\}) * A$. In other words, if a memory $\mathbb{M}$ contains a sub-memory $\mathbb{M}'$ separate from the rest of memory (described by some A), then writing to $\mathbb{M}$ will write to $\mathbb{M}'$ without disturbing the rest of memory.

Finally, I have to show that the object heap is still well-typed. This is true because the old heap was well-typed, and the value being written to the field is well-typed.

6.8 Putting it together

Once all of the operations have been verified, I must construct a module containing an entire verified Cheney collector. As part of this, I must verify the various properties described in Section 4.3.3. For the Cheney collector, these are not difficult to show. Most of the proof involves breaking down the representation predicate and splitting cases based on whether various registers are equal or not. Aside from that, I must show all the minor facts about the code memory type and the code memory, but these are not tricky given that I have already verified all of the code blocks of the

component	lines
generic GC infrastructure	1712
properties of copying predicates	1560
field scanning	1398
loop	970
enter and loop header	1222
allocator	720
read and write barriers	320
top level	456
total Cheney specific	5086

Figure 6.35: Cheney formalization line counts

collector.

6.9 Conclusion

I give the line counts for the Coq implementation of my Cheney collector verification in Figure 6.35. These line counts include white space and comments, but the bulk of the lines are proofs, which tend to be very dense and not include a lot of either white space or comments. The first two lines correspond to various lemmas about either collectors in general or copying collectors in particular and are shared in part with my other GC verifications. In each of these two categories, there are probably things that are not used except by the Cheney collector, and things that are not used by the Cheney collector. Below that, I break down the number of lines by component described in this chapter. The line counts for a component include the assembly implementation, the specification and the verification. I have included the full implementation and specification of these components in this chapter, showing that the size of each category is dominated by the proofs. The line counts could probably be reduced with improved tool support and better factoring out of lemmas. The bottom line totals up the number of lines for the Cheney collector components,

but does not include the libraries in the first two lines.

In this chapter, I have stepped through an entire assembly-level implementation of a Cheney copying collector and described the specification and verification of every basic block. I have also given an overview of how I verified that the collector satisfies the high-level interface described in Chapter 4.

Chapter 7

Baker Collector Verification

7.1 Introduction

The Baker garbage collector [Baker 1978] is an incremental variant of the Cheney collector [Cheney 1970]. An incremental collector does not perform a complete collection when it runs out of space. Instead, the collector does a little bit of work each time it allocates an object. This reduces the maximum time that will be spent in the garbage collector, which can be useful for an interactive program. In some sense, an incremental collector is one step closer to a concurrent collector, where the mutator and collector run at the same time.

While the basic mechanism of the Baker collector is the same as the Cheney collector, the invariants of the collector are more complex, because invoking the collector (usually) results in a heap that is only partially collected. The representation predicate of the Cheney collector, given in Figure 5.5, is much less complex than the representation predicate of the Baker collector, given in Figure 5.7.

In addition, ensuring that all reachable objects are copied (the basic soundness property of a garbage collector) becomes more difficult. Recall that black objects are

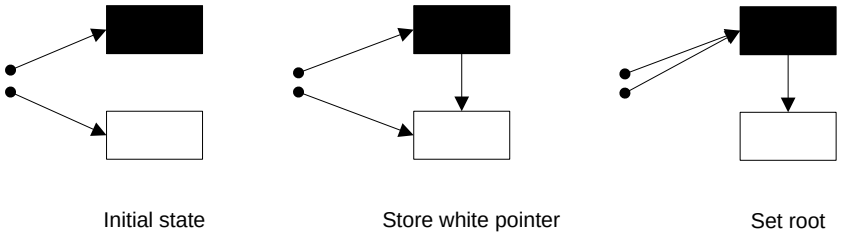

Figure 7.1: Incremental collection going awry

never visited again by the collector. Figure 7.1 shows how problems can arise in an incremental collector. In this example, there are two roots, one pointing to a black object and one pointing to a white object. The mutator executes a store instruction, storing a pointer to the white object into the black object. After that, the mutator sets both roots to point at the black object. In the final state, the white object is reachable from the roots, so it should not be collected. Unfortunately, the only path is through a black object, so the collector will never realize this and will collect it. The white object has become invisible to the collector.

Two conditions must hold for an object x to become invisible [Jones and Lins 1996]:

1. A black object contains a pointer to a white object x.

2. There is no path from a gray object to x that contains only white objects.

A sound incremental garbage collection algorithm must prevent one of these two conditions. In fact, a copying algorithm must prevent the two conditions above from holding on from-space objects that have already been copied, even though they are not considered to be white objects: when a collection is complete, a pointer to the old copy of an object is just as invalid as a pointer to an uncopied object. Those algorithms that prevent the first condition are known as *incremental update* algorithms, while those that prevent the second condition are known as *snap-shot-*

at-the-beginning algorithms.

The Baker collector is an incremental update algorithm, because it prevents black objects from containing pointers to white objects. Fortunately this type of invariant is easier to describe because it is very local. This invariant can be thought of as a standard heap typing judgment one might see in a language such as TAL [Morrisett et al. 1999], if a color is thought of as a type. A black object is well-typed if it contains no pointers that have a from-space "type". The Baker collector maintains this invariant by requiring that roots never contain from-space pointers. Because the operands of a write operation are always roots, a from-space pointer can never be written into any object, so the GC will never violate the first condition above.

That is all well and good for writes, but what about reads? When the GC reads the field of an object, it places the result into a root. The root set is allowed to contain gray objects, which may contain pointers to white objects. These two facts imply that a read operation may cause the root set to contain a from-space object, violating the invariant, which in turn can cause objects to become invisible to the GC.

To solve this problem, the Baker collector requires that some extra work is done whenever a read occurs. This is known as a *read barrier*. Before the field of an object is loaded, it must be scanned and forwarded (using the Baker GC's `scanField` procedure) to ensure that if the field contains a pointer to a from-space object it will be forwarded appropriately. Fortunately, my approach to garbage collector abstraction allows me to hide this complexity from the mutator and allows the Cheney and Baker collectors to have the exact same interface.

The memory of the Baker collector looks similar to that of the Cheney collector. Objects in the from-space are being copied to the to-space. The from-space is a mix of copied objects and uncopied white objects. Each copied object contains, in its

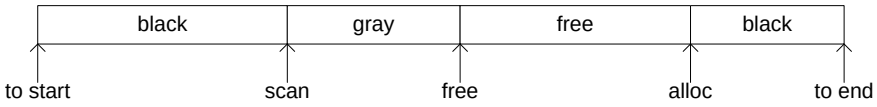

Figure 7.2: Baker to-space

first field, a pointer to its copy. The to-space looks like Figure 7.2, which is similar to what it looks like in the Cheney collector, but not quite the same. There are two blocks of contiguous black objects (objects that have been copied and scanned): one starting from the beginning of the to-space and going to the scan pointer, and one starting from the alloc pointer and going to the end of the to-space. The latter block is objects that have been allocated by the mutator during the current collection cycle. Black objects do not contain pointers to from-space objects. Gray objects range from the scan pointer to the free pointer, while free objects range from the free point to the alloc pointer. Gray objects can contain pointers to any valid from- or to-space object. Objects copied by the GC are allocated from the front of the free space, at the free pointer.

The general structure of this chapter is the same as the previous chapter, which discussed the verification of the Cheney collector. I step through the various components of the collector, starting from the innermost loop and working my way outwards, discussing the implementation, specification and verification of each component.

As in the previous chapter, everything discussed in this chapter has a machine checkable proof implemented in the Coq proof assistant. As far as I am aware, this is the first machine checkable proof of the soundness of the Baker algorithm.

```
scanField (void* field) {
  fieldVal = *field;
  if (notFromSpacePtr(fieldVal))
    return;
  fieldValField = *fieldVal;
  if (notToSpacePtr(fieldValField)) {
    *field = fwdObj(fieldValField);
  } else {
    *field = fieldValField;
  }
}
```

Figure 7.3: Baker field scanning pseudocode

7.2 Field scanning

Field scanning in the Baker collector is mostly the same as it is in the Cheney collector. Field scanning for the Cheney collector is described in Chapter 6.3. The implementation of field scanning differs from the Cheney collector in two respects. First, I do not maintain an invariant in the Baker collector strong enough to ensure that there is always enough space to copy an object. This requires the addition of a space check every time the GC copies an object. Secondly, in the Baker collector mutator writes during a collection can lead to gray objects containing to-space pointers in addition to atomic values and from-space pointers. To handle this, field scanning is modified to return immediately if any non-from-space value is encountered, instead of returning an atomic value. The pseudocode for field scanning in the Baker collector is given in Figure 7.3. `notFromSpacePtr` is analogous to `notToSpacePtr`: it returns true if the argument is atomic or does not fall within the bounds of the from-space.

The specifications are also mostly the same. A well-formed scan field state now requires that there are disjoint from- and to-spaces, but does not require that the set of white objects is smaller than the set of free objects. Also, the free space now

```
void bakerLoop () {
  // scan some objects
  i = 0;
  while (scan != free && i < scan_per_gc) {
    scanField(scan);
    scanField(scan + 1);
    scan += 2;
    i += 1;
  }

  // allocate an object
  if (free == alloc)
    while (1) {};

  alloc = alloc - 8;
  alloc[0] = NULL;
  alloc[1] = NULL;
  return alloc;
}
```

Figure 7.4: Baker loop pseudocode

ends at the register ralloc instead of at the end of the to-space. In the specification of the actual field scanning method, the field can now be in the to-space, in addition to being atomic or in the from-space. If the field is a to-space pointer, the field is not changed. The actual verification of the Baker `scanField` method is also largely the same as it was in the Cheney collector.

7.3 The loop

7.3.1 Implementation

The implementation of the main garbage collection loop (pseudocode given in Figure 7.4) is almost the same as it is in the Cheney collector. While the scan pointer and the free pointer are not equal, each object is scanned, in order. The difference

$$
\begin{aligned}
&\mathit{bakerLoopStOk}(\phi, W, M, F, A, \mathbb{S}) ::= \\
&\quad \mathbb{S}(\mathsf{rtoStart}) \leq \mathbb{S}(\mathsf{rscan}) \wedge \mathit{aligned8}(\mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree})) \wedge \\
&\quad \mathit{scanFieldStOk}(\phi, W, M, F, A, \mathbb{S})
\end{aligned}
$$

Figure 7.5: Baker loop state formation

is that there is an upper bound on the number of times the loop will execute, given by the constant `scan_per_gc`.

Another difference is that garbage collection is always performed when an object is allocated. As a consequence, after the loop an object is allocated and initialized. The code for this is similar to the Cheney collector. The difference here is that objects are allocated from the *end* of the to-space, to allow the collector to distinguish objects allocated during the current collection cycle from objects copied by the collector. The variable `alloc` points to the end of the free space, while `free` points to the beginning. If these pointers are equal, there is no more space, and the collector goes into an infinite loop.

I omit the assembly implementation here because it can be readily extrapolated from the Cheney collector assembly implementation given previously and the pseudocode given above. One thing to note is that the collector stores the new values of the various pointers into the GC info record before returning from allocation. There are three blocks in the assembly implementation of the loop. `bakerLoop` performs the loop test and the first call to scanField. `bakerLoop2` increments the variables in the loop, performs the second call to scanField, and returns to the top of the loop. `bakerExit` performs the allocation and initialization of the new object, stores the new values into the GC information record, and returns.

$$toSpace(\mathbb{S}) ::= \{k \mid \mathbb{S}(\mathsf{rtoStart}) \leq k < \mathbb{S}(\mathsf{rtoEnd})\}$$

$$\begin{array}{l} bakerLoopPre(\mathbb{S}) ::= \\ \quad \exists \phi, W, M, F. \\ \quad\quad bakerLoopStOk(\phi, W, M, F, \\ \quad\quad\quad (objHp(W \cup M \cup toSpace(\mathbb{S}), G) * \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto -, -, - * \mathtt{true}), \mathbb{S}) \\ \quad \text{where } G = rangeObjs(\mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree})) \end{array}$$

Figure 7.6: Baker loop precondition

7.3.2 Specification

While the implementation of the Baker loop looks very similar to the Cheney collector, the specification is fairly different. In the Cheney collector, the loop processes every remaining reachable object so the "post-condition" of the loop guarantee describes a fully-collected heap. In the Baker collector, only some of the remaining objects are examined, so the post-condition of the loop guarantee describes a partially-collected heap.

The Baker loop state predicate, given in Figure 7.5, extends the Baker scan field state predicate by requiring that the scan pointer is not before the beginning of the to-space, and that the scan and free pointers are object aligned. In the Cheney collector's equivalent of this predicate, I fixed the set of gray objects. I do not do this for the Baker collector because more flexibility is needed.

The Baker loop precondition, given in Figure 7.6, is almost the same as the loop precondition of the Cheney collector. The two differences are that the gray objects being scanned may now contain values in the to-space, and there is now some description of the GC info record, because it must be updated before the loop returns, to reflect the result of scanning and allocation.

The Baker loop guarantee, given in Figure 7.7, is where the real differences with the Cheney collector show up. Remember that this guarantee describes the behavior

$$objRespToMap(\phi, \mathbb{S}, x) ::= objRespMap(\phi \cup \mathrm{id}_{toSpace(\mathbb{S})}, \mathbb{S}, x)$$
$$objIsToFwded(\phi, \phi', \mathbb{S}, x) ::= objIsFwded(\phi \cup \mathrm{id}_{toSpace(\mathbb{S})}, \phi', \mathbb{S}, x)$$

$$
\begin{array}{l}
bakerLoopGuar(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \phi, G, W, M, F, A. \\
\qquad G = rangeObjs(\mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree})) \rightarrow \\
\qquad bakerLoopStOk(\phi, W, M, F, \\
\qquad\quad (objHp(W \cup M \cup toSpace(\mathbb{S}), G) * \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto -, -, - * A), \mathbb{S}) \rightarrow \\
\qquad \exists \phi', B, B', G_0, G', W', M', F'. \\
\qquad\quad W = M' \cup W' \ \wedge \ F = B' \cup G' \cup F' \cup \{\mathbb{S}'(\mathsf{ralloc})\} \wedge \ M' \cong_{\phi'} B' \cup G' \ \wedge \\
\qquad\quad G = B \cup G_0 \ \wedge \ B \cup B' = rangeObjs(\mathbb{S}(\mathsf{rscan}), \mathbb{S}'(\mathsf{rscan})) \ \wedge \\
\qquad\quad \mathbb{S}(\mathsf{rscan}) \leq \mathbb{S}'(\mathsf{rscan}) \wedge aligned8(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{rfree})) \ \wedge \\
\qquad\quad (\forall x \in W'.\ \mathbb{S}' \vdash objCopied(x, \mathbb{S}, x) * \mathtt{true}) \ \wedge \\
\qquad\quad bakerLoopStOk(\phi \cup \phi', W', M \cup M', F', \\
\qquad\qquad (\forall_* x \in B.\ objRespToMap(\phi \cup \phi', \mathbb{S}, x)) * (\forall_* x \in G_0.\ objCopied(x, \mathbb{S}, x)) * \\
\qquad\qquad (\forall_* x \in B'.\ objIsToFwded(\phi, \phi', \mathbb{S}, x)) * (\forall_* x \in G'.\ objCopiedMap(\phi', \mathbb{S}, x)) * \\
\qquad\qquad \mathbb{S}'(\mathsf{v0}) \mapsto \mathrm{NULL}, \mathrm{NULL} * \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto \mathbb{S}'(\mathsf{rscan}), \mathbb{S}'(\mathsf{rfree}), \mathbb{S}'(\mathsf{ralloc}) * A, \\
\qquad\qquad \mathbb{S}')) \ \wedge \\
\quad (\forall r \notin \{\mathsf{ralloc}, \mathsf{rscan}, \mathsf{rfree}, \mathsf{rcount}, \mathsf{a0}, \mathsf{v0}, \mathsf{ra}, \mathsf{ast}, \mathsf{rtemp0}, \mathsf{rtemp1}, \mathsf{rtemp2}, \mathsf{raSave0}\}. \\
\qquad \mathbb{S}(r) = \mathbb{S}'(r)) \ \wedge \\
\quad \mathbb{S}'(\mathsf{ra}) = \mathbb{S}(\mathsf{raSave}) \wedge \mathbb{S}'(\mathsf{ralloc}) + 8 = \mathbb{S}(\mathsf{ralloc}) \wedge \mathbb{S}'(\mathsf{v0}) = \mathbb{S}'(\mathsf{ralloc})
\end{array}
$$

Figure 7.7: Baker loop guarantee

of the remaining iterations of the loop, and thus does not concern itself with objects that have already been copied and scanned in previous iterations of the loop, or in previous invocations of the collector. The initial condition is similar (and almost the same as the precondition of the Baker loop), but the "post-condition" is different. In the Cheney collector loop, all gray objects (those that have been copied but have not had their fields updated) have their fields updated, and all white objects reachable from the gray objects are copied and have their fields updated. In the Baker collector, there are a variety of things that can happen.

As in the Cheney collector, the initial set of objects that the collector is concerned with are G (the set of objects that have previously been copied that the collector needs to scan), M (the set of from-space objects that have already been copied, and are mapped according to the partial function ϕ), W (the set of from-space objects that have not yet been copied) and F (the set of unallocated objects in the to-space).

What happens to these objects? Each object in G is either scanned (in which case it will be part of B) or not scanned (in which case it will be part of G_0). Each object in W is either determined to be reachable (in which case it will be copied and become part of M') or remains unexamined (in which case it will be part of W'). The copies of the objects in M' are either scanned (those in B') or remain unscanned (those in G'). The new and old copies of these objects are related by the isomorphism ϕ'. The initial set of objects F is split into $B' \cup G'$ (objects that are newly copied), F' (remaining free objects) and $\mathbb{S}'(\mathsf{ralloc})$ (the object allocated after the loop finishes).

There are some relationships between these sets that are not explicitly reflected in the guarantee. For instance, if G_0 is not empty, then B' must be empty, because the collector will scan all existing gray objects before it scans new ones.

There are some relationships between the various scan and allocation pointers

that are explicitly represented in the guarantee. The newly scanned objects make up the set $B \cup B'$, which must be the set of objects between the initial location of the scan pointer $\mathbb{S}(\mathsf{rscan})$ and the final location of the scan pointer $\mathbb{S}'(\mathsf{rscan})$. Furthermore, the new scan pointer is not smaller than the initial scan pointer. Also, the new free pointer is object-aligned with the old one, meaning that some whole number of objects has been allocated.

Another constraint is that all of the objects in W' (those that have not been examined by the collector) remain unchanged.

Finally, the state after the collector loop is run is still a well-formed loop state. The new mapping of from-space objects that have been copied to to-space objects is $\phi \cup \phi'$, the new set of unexamined from-space objects is W', the set of copied from-space objects is $M \cup M'$, and the new set of free objects is F'.

Now for the complicated part of the memory specification, which describes the part of the to-space that has been altered by the collector (along with the GC information record updates). To describe what happens here, I define two variations of predicates used in the verification of the Cheney collector. $\mathit{objRespToMap}(\phi, \mathbb{S}, x)$ holds on a memory if the memory is a pair located at x, where the values of the fields are the same as in $\mathbb{S}$ updated as follows: values that are atomic or in the to-space are unchanged, while other object pointers are updated according to ϕ. This describes objects in B that were copied before the loop and were scanned during the loop. The second predicate, $\mathit{objIsToFwded}(\phi, \phi', \mathbb{S}, x)$ is similar, except that the object at x was copied from some location y such that $\phi'(y) = x$, in addition to being scanned. This describes objects in B'. Aside from that, objects in G_0 are unchanged from the loop entry, and all objects $x \in G'$ were copied unchanged from some location y such that $\phi'(y) = x$. Aside from these objects that were copied and/or scanned, an object has been initialized at $\mathbb{S}'(\mathsf{v0})$ and the scan, free and allocation pointers have been stored

$$
\begin{array}{l}
\mathit{baker2Step}(\mathbb{S}) ::= \\
\quad \mathbb{S}\{\mathsf{a0} \rightsquigarrow \mathbb{S}(\mathsf{rscan}) + 4\}\{\mathsf{rscan} \rightsquigarrow \mathbb{S}(\mathsf{rscan}) + 8\} \\
\qquad \{\mathsf{rcount} \rightsquigarrow \mathbb{S}(\mathsf{rcount}) + 1\}\{\mathsf{ra} \rightsquigarrow \texttt{BAKERLOOP}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{bakerLoop2Pre}(\mathbb{S}) ::= \\
\quad \mathit{scanFieldPre}(\mathit{baker2Step}(\mathbb{S})) \wedge \\
\quad \forall \mathbb{S}'.\ \mathit{scanFieldGuar}(\mathit{baker2Step}(\mathbb{S}), \mathbb{S}') \rightarrow \mathit{bakerLoopPre}(\mathbb{S}')
\end{array}
$$

$$
\begin{array}{l}
\mathit{bakerLoop2Guar}(\mathbb{S}, \mathbb{S}'') ::= \\
\quad \exists \mathbb{S}'.\ \mathit{scanFieldGuar}(\mathit{baker2Step}(\mathbb{S}), \mathbb{S}') \wedge \mathit{bakerLoopGuar}(\mathbb{S}', \mathbb{S}'')
\end{array}
$$

Figure 7.8: Second Baker loop block specification

into the GC information record. Otherwise, memory is unchanged, as reflected by A.

In addition to these changes to memory, the guarantee describes how various registers are, or are not, changed. Of particular note, the specification states that the allocation pointer is moved exactly enough to allocate a pair, and that the value being returned is the new value of the allocation pointer. This means that the loop does not actually need to use v0 to return a value, but I use it anyway to maintain some semblance of a standard calling convention.

The specifications for the other two blocks in the loop are simpler. As is my custom, I have designed the specifications to push most of the hard work into a single proof.

For the specification of `bakerLoop2`, given in Figure 7.8, I first define a function *baker2Step* that describes the behavior of that block, up to the call to `scanField`, which is that various registers are incremented. The precondition is then that after those register updates happen, it is safe to call the field scanning method, and that, furthermore, after the return from scanning the field (which will be some state $\mathbb{S}'$), it will be safe to go to the start of the loop. The guarantee simply states that

$$
\begin{array}{l}
\mathit{bakerExitPre}(\mathbb{S}) ::= \\
\quad \mathit{aligned8}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{ralloc})) \wedge \\
\quad \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto -, -, - * \\
\qquad (\forall_* x \in \mathit{rangeObjs}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{ralloc})).\ x \mapsto -, -) * \mathtt{true} \\
\\
\mathit{bakerExitGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall A.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto -, -, - * \\
\qquad (\forall_* x \in \mathit{rangeObjs}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{ralloc})).\ x \mapsto -, -) * A \rightarrow \\
\qquad \mathbb{S}' \vdash \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto \mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{ralloc}) * \\
\qquad\quad \mathbb{S}'(\mathsf{ralloc}) \mapsto \mathrm{NULL}, \mathrm{NULL} * \\
\qquad\quad (\forall_* x \in \mathit{rangeObjs}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{ralloc})).\ x \mapsto -, -) * A) \wedge \\
\quad (\forall r \notin \{\mathsf{ralloc}, \mathsf{v0}, \mathsf{ra}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)) \wedge \\
\quad \mathbb{S}'(\mathsf{ralloc}) + 8 = \mathbb{S}(\mathsf{ralloc}) \wedge \mathbb{S}'(\mathsf{v0}) = \mathbb{S}'(\mathsf{ralloc}) \wedge \mathbb{S}'(\mathsf{ra}) = \mathbb{S}(\mathsf{raSave}) \wedge \\
\quad \mathit{aligned8}(\mathbb{S}(\mathsf{rfree}), \mathbb{S}'(\mathsf{ralloc}))
\end{array}
$$

Figure 7.9: Baker loop exit specification

`bakerLoop2` updates some registers, scans a field, then enters the loop again.

The loop exit specification is given in Figure 7.9. The precondition requires that three fields of the GC info record are present, and that there are objects in the range from the free pointer to the allocation pointer. The exit block checks to ensure that there is free space before allocating, so the precondition does not have to require it. The guarantee says that the GC info record has been updated properly, and that one object has been removed from the free space and initialized.

7.3.3 Verification

The verification of `bakerLoop2` is very brief, as the specification is a description of exactly what happens in the block. To verify `bakerExit`, the main task is to show that if the free and allocation pointers are unequal, then there is at least one object in the free space.

For the main block, `bakerLoop`, there are two main cases, depending on whether the loop is exited or not. I will discuss each separately.

Case: loop exit

There are two ways the loop can exit: either the collection is complete, or the current slice of collection is complete. To verify both of these cases, I first prove a lemma that says that for all states $\mathbb{S}$, if *bakerLoopPre*($\mathbb{S}$) then *bakerExitPre*($\mathbb{S}$) and for all states $\mathbb{S}'$ such that *bakerExitGuar*($\mathbb{S}, \mathbb{S}'$), then *bakerLoopGuar*($\mathbb{S}, \mathbb{S}'$). In other words, if the loop precondition holds, it is safe to call `bakerExit`, and if `bakerExit` is then executed, the entire loop will satisfy its specification. This lemma can be directly applied to the case where the collector is finished, and with only a small amount of additional work (to show that setting a temp register does not break anything) to the case when the collector is only finished with the current increment.

Showing that it is safe to jump to `bakerExit` is only a matter of unfolding the definition of *bakerLoopStOk* and matching up the memory predicates.

The loop precondition does not need to hold for the guarantee portion to hold. First I must instantiate the various existential variables in the "post-condition" of the guarantee. These are trivial, as there has been no additional collection. The map ϕ' that relates objects copied during the rest of this procedure to their new copies is the empty map, as the loop is not copying anything, and the sets B and B' of objects that are newly scanned (or copied and scanned) are also empty, again because the GC is not doing any more work before exiting the loop. The set of gray objects that were left unscanned is the initial set of gray objects G. The set of new gray objects is empty. The set of from-space objects still unexamined is W, and the set M' of objects that are copied during the remainder of this call is empty. Finally, the set of free objects is the initial set of free objects F minus the object that `bakerExit` will allocate and initialize, $\mathbb{S}'(\mathsf{ralloc})$.

With these existentials selected, the various constraints on the sets can be verified.

Most of the sets are empty, so these constraints are easy to show. A lemma is needed to prove that an empty map is an isomorphism from the empty set to the empty set.

To show that objects in G and W are not changed by `bakerExit`, I extract the portion of memory that contains them before applying its guarantee. Otherwise, showing that memory is well-formed is mostly a matter of showing that various empty sets can be ignored. For part of this, I use a lemma that shows that the set of objects in a range is non-empty if the ends are object-aligned and not equal. It not entirely straightforward to show that the gray objects were copied properly, requiring reasoning about sub-memories. The rest of the proof is straightforward.

Case: back into the loop (safety)

In this case, the GC has neither finished the collection nor the current increment, so it will execute the body of the loop again. The loop invariant *bakerLoopPre* holds on the initial state, the scan and free pointers are not equal, and the count is less than the number of scans for a single invocation of the collector. I must show that it is safe to scan the first field, and that after the GC has scanned the first field it can call `bakerLoop2` and that after it has executed `bakerLoop2`, the entire call has satisfied the loop guarantee.

First I show that the scan pointer is in the set of gray objects G. This requires knowing that the scan pointer and the free pointer are not equal.

First call to scan field To show that the first call to scan field is okay, I must first select the various sets (the set of white objects, mapped objects and free objects), along with the map that relates the mapped objects to to-space objects, but these are all unchanged from the entry into the loop, as no more work has been done yet. I must also show that the field being scanned is well-formed and contains either an

atomic value, a to-space value, or a from-space pointer. The field being scanned is part of an object that is in G and all fields of objects in G are well-formed in this way, so showing this only requires the application of standard object heap lemmas.

Call to bakerLoop2 Next, I must verify that it is safe to jump to `bakerLoop2` after scanning the first field. This requires showing that it is safe to scan the second field and that after scanning the second field it is safe to jump back to the start of the loop. Before I do this, I apply the field scanning guarantee, to get a predicate describing the state after the first field has been scanned.

At the second call to `scanField`, the sets of objects have changed. The white and free sets of objects are the white and free sets of objects resulting from the first call to `scanField`, while the set of copied objects is the initial set of copied objects plus the set of objects that were copied by the call to `scanField`. Finally, the isomorphism is the initial one plus the one created by the first call. The remainder of the safety proof for the second call to `scanField` is straightforward.

To show that it is safe to return to the start of the loop, I apply the guarantee of the second call to `scanField` to a description of the state before the call. This produces a description of the state after the second call, which is also the state in which the GC returns to the loop entry.

To show it is safe to reenter the loop, values must again be selected for the various sets of objects. As before, the sets of white objects and free space objects are the same as those "returned" by the second call to `scanField`. The set of gray objects is the set of gray objects starting at the scan pointer and ending at the free pointer. The set of mapped objects is a combination of the initial set of mapped objects and the sets of objects mapped by the two calls to `scanField`. The isomorphism relating mapped objects to to-space objects is a similar combination of the three

corresponding isomorphisms.

The new scan pointer is object-aligned with the new free pointer because the initial scan and free pointers are not equal (so there must be at least one gray object), the new scan pointer is 8 more than the initial scan pointer, and because the old and new free pointers are object aligned.

Finally, I must show that the gray objects created by the first and the second calls to `scanfield` are well-formed, containing only the appropriate sorts of pointers. To do this, I first establish that the two sets of gray objects created by the two calls are the sets of objects allocated during each call. In other words, each gray set begins at the value of the free pointer before the call and ends at the value of the free pointer after the call. As it happens, each is either empty or contains a single element, but specifying them in this way avoids creating too many cases. In any event, I must reason about the various sets of gray and free objects along with their disjointedness (which is implied by *scanFieldStOk*). For instance, the start and end points of the initial free set are known, and this set is split into the gray and free sets produced by the first call to `scanField`.

Once these sets are established, the gray object heap can be split into one part for each of these three gray sets (the two created by the calls, plus the initial set minus the object the GC just scanned). It must be shown that each part is well-formed (contain only atomic values, to-space values, or from-space pointers). The newly reduced initial set of gray objects is well-formed because it was well-formed at the beginning and has not subsequently changed. For the other two, I use a lemma that states that a copy of a well-formed object is a well-formed object.

Case: back into the loop (guarantee)

The last part of verifying `bakerLoop` is to verify that it has the proper behavior. It must be shown that if rscan $\neq$ rfree and the loop scans both fields of the first gray object then executes the loop k more times, then the loop guarantee for $k+1$ executions of the loop has been satisfied. A large chunk of this proof looks like the safety proof: a description of the initial state is combined with the various guarantees of the calls the loop makes to produce a description of the final state.

Once I have finally stepped through that largely duplicated portion of the proof, I must describe the final state. First I must instantiate all of the existential variables. The isomorphism from old objects to new objects is simply the union of the isomorphisms created by the scanning of each field, plus the isomorphism from the rest of the iterations of the loop. The set of from-space objects that have been copied in the final state is similarly a union of the three sets of from-space objects that were copied. The sets of remaining unscanned from-space objects and free objects are the same as they were after the execution of the loop.

For the rest of the sets, things become more complicated. The problem is that the rest of the iterations of the loop will scan some or all of the gray objects, but it cannot be determined which of these two cases actually holds.

There are many overlapping sets of gray objects created by the guarantees of the components I am trying to combine:

G initial set of gray objects

G_1 copied by the first field scan

G_2 copied by the second field scan

G_3 copied before restarting the loop, still unscanned at end of loop

G_4 copied after restarting the loop, still unscanned at end of loop

Also, remember that in this iteration of the loop the GC has scanned the object

$\mathbb{S}(\mathsf{rscan})$. B and B' are the sets of objects scanned beginning with the next iteration of the loop, where B is those objects that were gray at the start of the next iteration of the loop and B' is the rest of the newly scanned objects, which were copied during the rest of the iterations of the loop.

First, the set of objects that, during the entire execution of the loop, were scanned, but not newly copied, is $\{\mathbb{S}(\mathsf{rscan})\} \cup (B \cap G)$. In other words, this is the object that the GC scanned during this iteration of the loop, along with any objects scanned during the rest of the loop (B) that were also gray at the initial entry into the loop (G).

Next, the set of objects that were copied and had their fields forwarded is $(B \cap (G_1 \cup G_2)) \cup B'$. In other words, this is the set of objects that were scanned but not copied during the rest of the loop, but were copied during the first iteration of the loop, along with those objects B' that were copied and scanned in the rest of the iterations of the loop.

The set of gray objects that were neither copied nor scanned during the loop is $G \cap G_3$, which is those objects that were gray upon entry into the loop, and gray upon exit from the loop. The set of objects that were copied during the entire execution of the loop but not scanned is $((G_1 \cup G_2) \cap G_3) \cup G_4$.

All that remains is to show that the final state actually satisfies all of these sets that I have selected. This involves a lot of reasoning about sets, requiring properties such as the distributivity of set intersection over set union and the associativity of set union. It also requires a lemma that says that the union of two isomorphisms is an isomorphism from the union of their domains to the unions of their ranges, if their domains and ranges are disjoint.

To show that the final state is well-formed, I must split up and then recombine in different ways many sets. For instance, B can be split into the set of initially

gray objects that were scanned after reentering the loop $((G - \{(\mathbb{S}(\mathsf{rscan}))\}) \cap B)$ and the set of objects that became gray during the first iteration of the loop that were scanned after reentering the loop $((G_1 \cup G_2) \cap B)$. I also apply various lemmas that allow me to reason about transitivity. For example, if an object is unchanged from a state $\mathbb{S}$ to a state $\mathbb{S}'$, and state $\mathbb{S}''$ contains a scanned version of the object from $\mathbb{S}'$, then it also contains a scanned version of the object in $\mathbb{S}$. Or, if an object is copied and scanned from state $\mathbb{S}$ to $\mathbb{S}'$, then unchanged from state $\mathbb{S}'$ to $\mathbb{S}''$, then it is also copied and scanned from $\mathbb{S}$ to $\mathbb{S}''$. Some of these lemmas can be reused from the verification of the Cheney collector, but some are new. In addition, I also have to reason about weakening of object forwarding: if an object has been forwarded according to map ϕ, then it has also been forwarded according to the map $\phi \cup \phi'$, if ϕ and ϕ' are disjoint in their domains and ranges.

7.4 Allocator

Next I will discuss the entry point into the Baker allocator. The actual allocation occurs after the loop, as described in the previous section. In terms of the actual instructions, the allocator blocks mostly initialize variables and perform a few checks. However, this is where I show that the concrete representation I have described so far in this chapter matches with the abstract representation described in Chapter 5.4. The Baker collector has the most complex representation of any of the collectors I have discussed. For this reason, the verification of the allocator entry block is more complex than verifying the loop, at around 4300 lines of Coq proof, versus 3000 lines to verify loop.

```
void bakerAlloc () {
  // if we have not run out of space, do some collection
  // and allocate a new object
  if (free <> alloc)
    return bakerLoop();

  // if we are out of space, but have not finished scanning, abort
  if (scan < free)
    while (1) {};

  // swap semi-spaces
  swap(frStart, toStart);
  swap(frEnd, toEnd);

  // init other pointers
  free = toStart;
  scan = toStart;
  alloc = toEnd;

  scanField(root);
  return bakerLoop();
}
```

Figure 7.10: Baker allocator pseudocode

7.4.1 Implementation

Pseudocode for the implementation is given in Figure 7.10. If there is still space left, the allocator immediately goes to the loop, which performs some collection work and allocates a new object. If not, and the previous collection has not completed, the collector gives up. It is possible to maintain an invariant between the amount of work left and the number of free objects left that would avoid this, but it also sometimes requires expanding the size of the semi-spaces (for instance, if there is not enough free space left after a collection to maintain the ratio), so I skip it to simplify things.

The actual assembly implementation is broken into two blocks. The first block, `bakerAlloc`, does most of the work. The second block, `restoreRoot` does a little bit of cleanup, then performs the final call to `bakerLoop`. Otherwise, the main difference from the pseudocode implementation is that the various pointers are stored in memory instead of in global variables, which must be restored and saved appropriately.

7.4.2 Specification

Basic state formation

Before I give the actual specifications for the two blocks in the allocator, I first need to build up some helper specifications. First, the basic formation specification for the Baker collector is given in Figure 7.11. This has a number of parameters that can be instantiated in different ways at different points in the code. $\mathbb{M}_A$ is the abstract memory and $\mathbb{M}$ is the concrete memory. *gcInfo* is a pointer to the GC information record, *root* is the root, while *frStart*, *frEnd*, *toStart* and *toEnd* are the beginning and ends of the from- and to- spaces. *scan*, *free* and *alloc* are the scan, free and allocation pointers. The scan pointer points to the first unscanned object, while the free and allocation pointers demarcate the beginning and end of the free space. x, y and z are

$$
\begin{array}{l}
\mathit{bakerStOk}'(\mathbb{M}_A, \mathbb{M}, \mathit{gcInfo}, \mathit{root}, \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \\
\quad x, y, z, A) ::= \\
\quad \exists \phi, W, M. \\
\quad\quad \text{aligned8 } [\mathit{toStart}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \mathit{toEnd}] \wedge \mathit{aligned8}(\mathit{frStart}, \mathit{frEnd}) \wedge \\
\quad\quad W \cup M = \mathit{rangeObjs}(\mathit{frStart}, \mathit{frEnd}) \wedge M \cong_\phi B \cup G \wedge \\
\quad\quad \mathit{okFieldVal}(\mathit{toObjs}, \mathit{root}) \wedge \\
\quad\quad \mathbb{M} \vdash \mathit{mapHp}(M, \phi) * \mathit{objHp}(W \cup M, W) * \\
\quad\quad\quad \mathit{objHp}(\mathit{toObjs}, B \cup B') * \mathit{objHp}(W \cup M \cup \mathit{toObjs}, G) * \\
\quad\quad\quad (\forall_* x \in \mathit{rangeObjs}(\mathit{free}, \mathit{alloc}).\ x \mapsto -, -) * \\
\quad\quad\quad \mathit{gcInfo} \mapsto \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \mathit{toEnd}, x, y, z * A \ \wedge \\
\quad\quad \mathbb{M}_A \vdash (\forall_* x \in B \cup B'.\ \mathit{objCopied}(x, \mathbb{M}, x)) * \\
\quad\quad\quad\quad (\forall_* x \in G \cup W.\ \mathit{objRespMap}(\phi^{\mathrm{id}}, \mathbb{M}, x)) \\
\\
\quad \text{where } B = \mathit{rangeObjs}(\mathit{toStart}, \mathit{scan}),\ B' = \mathit{rangeObjs}(\mathit{alloc}, \mathit{toEnd}), \\
\quad\quad G = \mathit{rangeObjs}(\mathit{scan}, \mathit{free}),\ \mathit{toObjs} = B \cup G \cup B'
\end{array}
$$

Figure 7.11: Basic Baker state well-formedness predicate

the values of the last three fields of the GC information record. While the collector is not operating, x, y and z will be the values of the scan, free and allocation pointers, respectively. Finally, as usual A describes the portion of the concrete memory not containing the collector data structure.

There are only three existential variables for this predicate. ϕ is the isomorphism describing the object copying that has already occurred. M is the set of from-space objects that have been copied already, while W is the rest of the from-space objects. There are also a few abbreviations I use for convenience. B is the set of objects that have already been copied and scanned, ranging from the beginning of the to-space to the scan pointer, while B' is the set of objects that have been allocated by the mutator during the current collection cycle and ranges from the allocation pointer to the end of the to-space. G is the set of gray objects (objects that have been copied but not scanned), and ranges from the scan pointer to the free pointer. Finally, *toObjs* is the set of allocated to-space objects, and includes all three of these sets.

First there are constraints that do not involve memory. The many heap pointers must be object aligned, to avoid having to reason about fractional objects. I write aligned8 $[x_0, x_1, ...x_n]$ as an abbreviation for $\mathit{aligned8}(x_0, x_1) \wedge \mathit{aligned8}(x_1, x_2) \wedge ... \wedge \mathit{aligned8}(x_{n-1}, x_n)$. Next, W and M must together comprise the entire from-space, and ϕ must be an isomorphism from M to $B \cup G$. The root must either be atomic or a pointer to a to-space object. Note that the root cannot contain a from-space pointer. The read barrier will maintain this invariant.

The next part of the invariant describes the concrete memory. The concrete memory contains mapped objects M which implement the finite map ϕ, uncopied from-space objects W which can only point to other from-space objects, black to-space objects $B \cup B'$ that can only point to to-space objects, and gray objects G that can point to either from- or to-space objects. Next, there is a pair for every object in the free space. Finally, the concrete memory contains a record with all of the garbage collector information and the rest of memory is described by A.

The abstract memory contains two types of objects and nothing else. First, any black objects (in $B \cup B'$) are copied exactly from the concrete memory. The other objects, in G and W, might contain from-space pointers in the concrete memory, so the invariant requires that in the abstract memory they have their fields mapped by the forwarding map. The map used to do the forwarding is ϕ^{id}, which is simply the map ϕ extended with the identity map at any object not in M. In this way, the abstract memory is a version of the concrete object heap where all obsolete pointers to M have been forwarded by ϕ.

State formation variants

Three variants of the basic Baker state formation predicate are given in Figure 7.12. All of these variants take three arguments. The first argument is the abstract state,

$$
\begin{array}{l}
\mathit{bakerStOk}(\mathbb{A}, A, \mathbb{S}) ::= \\
\quad \exists \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}. \\
\qquad \mathbb{A}(\mathsf{root}) = \mathbb{S}(\mathsf{root}) \wedge \\
\qquad \mathit{bakerStOk}'(\mathit{memOf}(\mathbb{A}), \mathit{memOf}(\mathbb{S}), \mathbb{S}(\mathsf{gcInfo}), \mathbb{S}(\mathsf{root}), \\
\qquad\quad \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \mathit{scan}, \mathit{free}, \mathit{alloc}, A)
\end{array}
$$

$$
\begin{array}{l}
\mathit{bakerStOkLoaded}(\mathbb{A}, A, \mathbb{S}) ::= \\
\quad \exists x, y, z. \\
\qquad \mathbb{A}(\mathsf{root}) = \mathbb{S}(\mathsf{root}) \wedge \\
\qquad \mathit{bakerStOk}'(\mathit{memOf}(\mathbb{A}), \mathit{memOf}(\mathbb{S}), \mathbb{S}(\mathsf{gcInfo}), \mathbb{S}(\mathsf{root}), \mathbb{S}(\mathsf{rfrStart}), \mathbb{S}(\mathsf{rfrEnd}), \\
\qquad\quad \mathbb{S}(\mathsf{rtoStart}), \mathbb{S}(\mathsf{rtoEnd}), \mathbb{S}(\mathsf{rscan}), \mathbb{S}(\mathsf{rfree}), \mathbb{S}(\mathsf{ralloc}), x, y, z, A)
\end{array}
$$

$$
\begin{array}{l}
\mathit{bakerStOkPost}(\mathbb{A}, A, \mathbb{S}) ::= \\
\quad \exists \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}. \\
\qquad \mathbb{A}(\mathsf{root}) = \mathbb{S}(\mathsf{root}) \wedge \\
\qquad x \in \mathit{rangeObjs}(\mathit{alloc}, \mathit{toEnd}) \wedge \\
\qquad \mathbb{A}(x) \notin \mathrm{dom}(\mathit{memOf}(\mathbb{A})) \ \wedge \ (\mathbb{A}(x) + 4) \notin \mathrm{dom}(\mathit{memOf}(\mathbb{A})) \wedge \\
\qquad \mathit{bakerStOk}'(\mathit{memOf}(\mathbb{A}'), \mathit{memOf}(\mathbb{S}), \mathbb{S}(\mathsf{gcInfo}), \mathbb{S}(\mathsf{root}), \mathit{frStart}, \mathit{frEnd}, \mathit{toStart}, \\
\qquad\quad \mathit{toEnd}, \mathit{scan}, \mathit{free}, \mathit{alloc}, \mathit{scan}, \mathit{free}, \mathit{alloc}, A) \\
\quad \text{where } x = \mathbb{S}(\mathsf{v0}) \text{ and } \mathbb{A}' = \mathbb{A}\{x \rightsquigarrow \mathrm{NULL}\}\{x + 4 \rightsquigarrow \mathrm{NULL}\}\{\mathsf{v0} \rightsquigarrow x\}
\end{array}
$$

Figure 7.12: Baker state well-formedness predicate variants

the second is the predicate describing the remainder of the concrete memory, and the third argument is the concrete state. All three of these variants require that the roots in the concrete and abstract states are equal, like all of the collectors I have discussed. (This is not the case in, for instance, the Brooks incremental copying collector [Brooks 1984].) Also, in all three variants the root and GC information record pointer are stored in registers, and the memories of the abstract and concrete states are passed along to *bakerStOk′*.

The predicate *bakerStOk* describes the state when the collector is not running. No additional pointers are in registers, and the values of the scan, free and allocation pointers are stored in the GC information record. The predicate *bakerStLoaded* describes the state after the GC information record has been loaded into registers. In this state, all of the relevant pointers have been loaded, and the values of the scan, free and allocation slots in the GC information record are junk values x, y and z the GC does not care about. Finally, *bakerStOkPost* describes the state after a new object has been allocated. The new object is stored in x (*i.e.,* $\mathbb{S}(\mathsf{v0})$). The new object is in the range of allocated objects, and must not exist in the initial abstract state $\mathbb{A}$. The new abstract state $\mathbb{A}'$ is the actual abstract state represented by the concrete state, and is the initial abstract state with the fields of the new object initialized and the return register updated. Otherwise, the various pointers (aside from gcInfo and root) are stored away in memory, as in *bakerStOk*.

Code block specifications

I use the basic Baker allocator state predicates to define the specifications for the code blocks. The specifications for the allocator entry point are given in Figure 7.13. This allocator specification is similar in spirit to the high-level allocator specification given in Chapter 4, but not identical. There is no fundamental reason for this

$$
\begin{array}{l}
\mathit{bakerAllocPre}(\mathbb{S}) ::= \ \exists \mathbb{M}.\ \mathit{bakerStOk}(\mathbb{M}, \mathtt{true}, \mathbb{S}) \\
\\
\mathit{bakerVirtGuar}(\mathbb{A}, \mathbb{A}') ::= \\
\quad \forall \mathit{objs}, \mathbb{M}.\ \mathbb{A} \vdash (\mathit{minObjHp}(\{\mathbb{A}(\mathsf{root})\}, \mathit{objs}) \wedge \mathit{eq}\ \mathbb{M}) * \mathtt{true} \rightarrow \\
\quad\quad \exists \mathit{objs}', \phi.\ \mathit{objs} \cong_\phi \mathit{objs}' \wedge \phi^*(\mathbb{A}(\mathsf{root})) = \mathbb{A}'(\mathsf{root}) \wedge \\
\quad\quad\quad \mathbb{A}' \vdash (\forall_* x \in \mathit{objs}'.\ \mathit{objIsFwded}_1(\phi, \mathbb{M}, x)) * \mathtt{true} \\
\\
\mathit{bakerAllocGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathbb{A}, A.\ \mathit{bakerStOk}(\mathbb{A}, A, \mathbb{S}) \rightarrow \\
\quad\quad \exists \mathbb{A}'.\ \mathit{bakerVirtGuar}(\mathbb{A}, \mathbb{A}') \wedge \mathit{bakerStOkPost}(\mathbb{A}', A, \mathbb{S}')) \wedge \\
\quad \mathit{calleeSavedOk}(\mathbb{S}, \mathbb{S}') \wedge \mathbb{S}(\mathsf{ra}) = \mathbb{S}'(\mathsf{ra})
\end{array}
$$

Figure 7.13: Baker allocator specification

discrepancy. The Baker-specific specification was developed from the bottom-up to give a comprehensible specification for the Baker collector. On the other hand, the general interface was developed from the top-down, in an attempt to describe allocation in general. In Section 7.4.5, I will describe how I prove that the specification given in this section is weaker than the top-down specification described previously.

In any event, the precondition simply requires that the state is a well-formed Baker collector state. For the guarantee, I first define an abstract guarantee. This says that for any minimal set of objects *objs* reachable from the root register contained in a memory $\mathbb{M}$, there exists some other set of objects objs' and a map ϕ such that ϕ is an isomorphism from *objs* to objs'. Furthermore, the root in $\mathbb{A}'$ is forwarded according to ϕ, and the fields of the objects in *objs* are forwarded versions of the equivalent objects in *objs*. I do not specify what happens to the rest of the abstract memory. Note that *objs*, and thus $\mathbb{M}$, is unique, but the verification of the allocator does not depend on this, so I do not prove it.

Using the abstract guarantee, I can define the concrete guarantee for the allocator. In any well-formed Baker state that represents the abstract state $\mathbb{A}$, there exists some other abstract state $\mathbb{A}'$ such that $\mathbb{A}$ and $\mathbb{A}'$ are related by the abstract Baker

$$
\begin{array}{l}
\mathit{restoreRootPre}(\mathbb{S}) ::= \\
\quad \exists \mathit{root}, \mathbb{M}.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto \mathit{root} * \mathtt{true} \wedge \\
\qquad \mathit{bakerStOkLoaded}(\mathbb{M}, \mathtt{true}, \mathbb{S}\{\mathsf{root} \rightsquigarrow \mathit{root}\}) \\
\\
\mathit{restoreRootGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad \forall \mathit{root}.\ \mathbb{S} \vdash \mathbb{S}(\mathsf{gcInfo}) + 16 \mapsto \mathit{root} * \mathtt{true} \rightarrow \\
\qquad (\exists \mathbb{M}.\ \mathit{bakerStOkLoaded}(\mathbb{M}, \mathtt{true}, \mathbb{S}')) \wedge \\
\qquad \mathit{bakerLoopGuar}(\mathbb{S}\{\mathsf{root} \rightsquigarrow \mathit{root}\}, \mathbb{S}')
\end{array}
$$

Figure 7.14: Restore root specification

guarantee, and the final concrete state represents the final abstract state.

I give the specification for `restoreRoot` in Figure 7.14. The precondition requires that the root is stored in a particular field of the GC information record, and that the state is a well-formed Baker collector state in which the pointer values have been loaded into registers. The root must be stored in the GC record because the field scanning method only scans values stored in memory, and reusing the GC record avoids some allocation. The guarantee says that the result of this block is a well-formed loaded Baker collector state, such that the initial and final states are related by the collector guarantee, once the root register has been updated. Note that the precondition is not fully present in the guarantee as is usually the case, but here it is not needed.

7.4.3 Allocator verification

The beginning of `bakerAlloc` is a series of loads and adds (not shown in the pseudocode) that are easy to verify. At the first branch, there are two cases: either there is space left or there is not.

Flip before collection

I consider the case where there is no space left first. In this case, the GC must flip the semi-spaces and scan the root before it can continue. There are two subcases here. Either the previous collection has finished or it has not. If it has not, then the GC immediately goes into an infinite loop, so that subcase is easy to verify. If it has finished, then verification can proceed, knowing that the scan pointer is greater than or equal to the free pointer. The actual flip is a series of adds and stores, so that is easy to verify.

After this, I pause to reflect the flip onto the abstract level, by first renaming the from- and to-space pointers to reflect their current status. Also, the free and scan pointers must be equal, which means there are fewer variables to worry about. I also clean up the memory predicate. For instance, I coalesce the old white and mapped objects into a single buffer, which is the new to-space. I also show that the two old sets of allocated objects, B and B', together cover the entire old to-space.

Once this is done, showing that it is safe to call `scanField` is mostly a matter of grinding through some tedious proofs. The only interesting part is selecting what the various sets of objects are. Because the GC has just flipped the semi-spaces, going into the call to `scanField` the forwarding map is empty, the set of white space objects is all of the objects of the from-space, and the set of copied objects is empty. Finally, the set of free objects is all of the objects of the to-space.

After this, I must show that it is safe to call `restoreRoot`. To do this, I have to select an abstract memory that is encoded in the concrete state. The concrete version of this abstract memory is the part of the memory that contains the gray objects generated by scanning the root, along with the remaining white space objects. To convert this concrete sub-memory to an abstract sub-memory, I map its range with

ϕ^{id}, where ϕ is the mapping of copied from-space objects to to-space objects created, as before, by scanning the root. As I have said before, ϕ^{id} is the map ϕ extended by the identity function everywhere except the domain of ϕ. Showing that the concrete state is well-formed requires applying many of the lemmas I have used before, plus a lemma that says that $\mathit{rangeObjs}(x, x)$ is equal to the empty set.

First I use a lemma to show that the sub-memory of the concrete state that contains the objects I am interested in is copied from the larger sub-memory. Then I use some lemmas to relate a memory to a version of the memory that is mapped by a function f. For instance, if $\mathbb{M} \vdash x \mapsto y$, then $\mathbb{M}' \vdash x \mapsto f\ y$, where $\mathbb{M}'$ is $\mathbb{M}$ with its range mapped by f. Similar lemmas are needed for $*$ and $\forall_*$. Using these lemmas I can easily show that the abstract memory I selected properly matches the concrete memory.

Guarantee

Finally for this case, I have to show that the guarantee is satisfied. This requires stepping through the guarantees of root scanning and the Baker loop, which is similar to what I have already done. There also is a lot of miscellaneous cleanup.

Once this is done, I work to eliminate the various intermediate states by describing the final state in terms of the initial state. For instance, while the objects scanned by the loop have merely had their fields forwarded relative to the beginning of the loop, they have in fact been copied and scanned since the entry to the function, as there were no objects in the to-space initially. Similarly, gray objects that the loop did not scan have not been changed relative to the entry into the loop, but have been copied from the from-space relative to the entry into the function. And so on for the various other sets of objects.

After the intermediate states have been cleaned up, I must show that the abstract

guarantee holds, and that the final concrete state is well-formed. For the abstract guarantee, the set $objs'$ of reachable objects in the final state is the initial set of reachable objects ($objs$) mapped by $(\phi \cup \phi')^{\mathrm{id}}$, where ϕ and ϕ' are the object mappings created by scanning the root and running the collector loop, respectively. In other words, any object that has been copied is mapped to its copy, and the rest are left alone. The mapping from the initial to the final sets of objects is $\mathrm{id}_{W'} \cup \phi \cup \phi'$, with the domain restricted to objects in the initial set of reachable objects. I show that $objs'$ is a subset of the gray, black and white objects at the end of the loop. I must also show that various combinations of the objects are disjoint, in the ways that one might expect. For instance, the set of objects that have been copied from the from-space is disjoint from the set of to-space objects. I must also prove that the prelude of the allocator that initializes the various fields does not alter the value of any of the objects.

I must also show that the final abstract memory contains the objects in $objs'$, and furthermore that these objects have been copied and forwarded according to the map $\mathrm{id}_{W'} \cup \phi \cup \phi'$. The concrete version of this memory must contain white, gray and black objects. The white objects have been left unchanged, the gray objects have been copied, and the black objects have been forwarded. First I split each of these sets (for instance, W') into parts that are reachable in the final state (for instance, $W' \cup objs'$) and those that are not (for instance, $W' - objs'$). With a few lemmas about these set operations, I can show this splitting is sound. Now I must show that mapping the fields of the reachable objects in the white, gray and black sets results in forwarded objects from the initial abstract state. Doing this requires the set of lemmas mentioned before that describe what happens when the range of a memory is mapped according to a function f. For instance, one of these lemmas states that if $\mathbb{M} \vdash x \mapsto y$, then $\mathbb{M}' \vdash x \mapsto f\ y$, where $\mathbb{M}'$ is $\mathbb{M}$ with its range mapped by f.

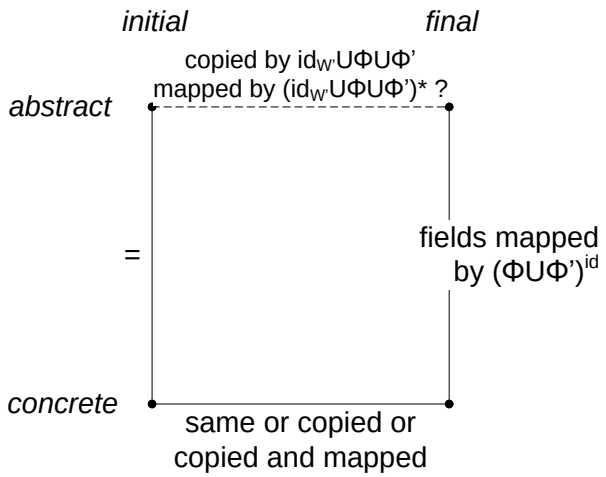

Figure 7.15: Baker commutativity

I also use a lemma that states that if $S \subseteq S'$, then $S = S' \cap S$, where S' is $objs'$ and S is the set of white, gray and black objects in the final state. After some more tedious reasoning steps, I can show that $objs'$ is equal to the union of three sets, where each of these sets is the intersection of $objs'$ with either the white, gray or black objects.

A diagram showing what I am trying to prove is given in Figure 7.15. Each dot represents one of the states being reasoned about. The top two states are abstract states, while the bottom two states are concrete. The two states on the left are the initial states, while the ones on the right are the final states. For instance, the upper left state is the initial abstract state. A solid line indicates a known relationship between two states, while the dotted line between the two top states indicates the relationship I must prove holds.

There are three cases to consider, one for each color (white, black and gray). For each of these colors, the final abstract state is the final concrete state with its fields mapped by $(\phi \cup \phi')^{\mathrm{id}}$. Also, the initial abstract and concrete states are identical, because the initial state represents the state at the beginning of a collection cycle,

so no forwarding needs to be done. The relationship between the initial and final concrete states depends on the color of the object. White objects are unchanged. Gray and black objects have been copied, following the mapping $\phi \cup \phi'$. Black objects have also had their fields forwarded, using the mapping $(\phi \cup \phi')^{\mathrm{id}}$.

The edge connecting the two top states is, as I have said, what I am trying to prove. Specifically, I am trying to show that all of the objects in the abstract state have been copied (and had their fields forwarded) according to the mapping $\mathrm{id}_{W'} \cup \phi \cup \phi'$. This is basically a commutativity diagram, so this can be solved this in the obvious way: I show that the composition of the left, bottom and right relations implies the top relation.

Consider the case of an object that is white after the collector has run (that is, an object in W'). This object is the same in the initial abstract and initial concrete states. It is also the same in the initial and final concrete states. Thus the relationship between this object in the initial abstract state and the final abstract state is equal to the relationship between this object in the final concrete state and the final abstract state.

Consider the top edge. Copying an object x in W' according to the map $\mathrm{id}_{W'} \cup \phi \cup \phi'$ does not do anything because $(\mathrm{id}_{W'} \cup \phi \cup \phi')(x) = x$. Thus for this case it only remains to show that mapping the fields of a white object with the map $(\mathrm{id}_{W'} \cup \phi \cup \phi')^*$ is the same as mapping the fields with $(\phi \cup \phi')^{\mathrm{id}}$. The typing judgment of white objects implies that there are only three subcases to consider for the value v of a field:

1. The field value v is atomic. In this case, v is odd, and thus it cannot be in the domain of $\phi \cup \phi'$. Thus with both maps v is mapped to v. Subcase holds.

2. The field value v is in W'. Again, v is not in the domain of $\phi \cup \phi'$, and thus

v will again be mapped to v by both maps. (Though for slightly different reasons.) Subcase holds.

3. The field value v is in $M \cup M'$ (the sets of objects that have been copied). In this case, v must be even and not in W', so both maps will map v to $(\phi \cup \phi')(v)$. Subcase holds.

All subcases hold, so this case holds.

The cases for black and gray objects are similar. First, compose the effects of the three sides, simplifying where possible. Then ensure that the objects are copied to the same places. Finally, ensure the fields of the objects are updated properly by considering the various possible values of the fields.

Next, I must show that the object being allocated is not already part of the abstract object heap. I can do this because I have already shown that the object being allocated is not part of the concrete object heap, and the abstract object heap is just the concrete object heap with its range mapped.

Then, I must show that the final concrete state is well-formed. In addition to the usual barrage of tedious reasoning about sets, I must show that the final object heap is well-formed. This requires showing that the colored sets produced by the collector do indeed occur in the expected ranges, a sort of reasoning I have had to do before. It also requires showing that an object with its fields initialized to NULL is well-formed, and that forwarding a well-formed object results in another well-formed object that contains object pointers in the range of the map used for forwarding, if the initial object had only pointers in the domain of the forwarding map. The final major bit of reasoning here requires showing that the gray objects are well-formed, which is true because they are copies of well-formed white-space objects.

Finally, I must show that the final abstract state is well-formed, by showing that

black objects are the same in the concrete and abstract states, and that gray and white objects are forwarded using the current mapping. Black objects are the same in both states because any pointers they contain are for to-space objects, and the map only changes from-space objects. For gray and white objects, I am basically just showing that their field values are in the domain of the mapping. I also have to reason about the addition of the newly allocated object to the abstract memory. To do this, I use a lemma that says that if $\mathbb{M} \vDash A$, $\neg\mathbb{M} \vDash x \mapsto - * \texttt{true}$ and x is a word-aligned pointer, then $\mathbb{M}\{x \rightsquigarrow v\} \vDash x \mapsto v * A$. In other words, if x is fresh in memory, then setting x to some value v will extend memory with that value, without interfering with the rest of memory.

After the usual light reasoning about the registers, this case is complete.

No need to flip

In this case, there is still free space left, so there is no need to flip spaces and start a new collection. But the GC still needs to do some more work for the current collection. This case is easier than the other case in some respects, as I do not have to reason about the flip. On the other hand, things are not starting from a clean slate, as the initial set of copied objects is not empty.

First I define the various sets of objects, and relate them to the various pointers. The to-space looks like Figure 7.2. One set of black objects begins at the beginning of the to-space and ends at the scan pointer. The set of gray objects begins at the scan pointer and ends at the free pointer. These objects together comprise the set of objects that have been copied during the current collection cycle. An isomorphism ϕ relates the set of copied from-space objects M to them. In addition, there is another set of black objects that have been allocated during the current collection cycle, which ranges from the allocation pointer to the end of the to-space. These

two sets of black objects plus the set of gray objects together comprise the set of allocated to-space objects. The root is either atomic or a member of the set of to-space objects. There are also unallocated free objects F in the to-space that range from the free pointer to the allocation pointer. The allocated to-space objects along with F comprise the entire to-space. The from-space is made up of copied objects M plus white objects W that have not been copied.

Once that is done, I can show that it is safe to enter the Baker loop, which poses no difficulties. Next, I must verify the guarantee. After a little cleanup of the hypotheses I apply the loop guarantee. After that, I can extract the concrete object heap from the larger concrete state. As in the other case, the abstract state is $(\mathbb{M}', \mathit{rfileOf}(\mathbb{S}'))$, where $\mathbb{S}'$ is the final concrete state and $\mathbb{M}'$ is the concrete object heap with its range mapped by $(\phi \cup \phi')^{\mathrm{id}}$. ϕ is the initial mapping from copied objects to to-space objects, and ϕ' is the mapping created by the execution of the Baker loop. Before showing the rest of the guarantee, I show that various sets are disjoint as one might expect. For instance, W and M are disjoint.

Next I must show that the abstract guarantee holds. The set of reachable objects in the final state is the set of reachable objects in the initial state mapped by ϕ and ϕ'. This is fairly similar to the equivalent part of the other case.

Finally I show *bakerStOkPost* (defined in Figure 7.12) holds on the final state. There are a number of subgoals here.

The first tricky subgoal is to show that each field of the newly allocated object is fresh in the abstract object heap (*i.e.,* was not previously allocated). First I establish that the newly allocated object is fresh in the *concrete* object heap. I do this by contradiction: the object being allocated must be part of the concrete memory outside of the object heap (specifically, in the free space), so it cannot be part of the concrete memory inside of the object heap. Next, the concrete and abstract object

heaps must have the same domain, so the newly allocated object cannot have been part of the initial abstract object heap either.

Next, I must show that the final state satisfies *bakerStOk′*, defined in Figure 7.11. The mapping from old to new objects is the union of the old and new mappings, while the set of white objects is the set of white objects after performing the collection. Finally, the set of mapped objects is the initial set of mapped objects plus the set of objects copied during the most recent collection. After this, there is a lot of tedious reasoning about finite sets and the alignment of objects.

In one of these subgoals, I show that the set of gray objects in the final state (that range from the final scan pointer to the final free pointer) is equivalent to the initial set of gray objects, minus the set of newly black objects, and along with the new gray objects. This lemma requires a 150 line Coq proof, reasoning about the various contiguous sets of objects (gray, black and free) in terms of splitting, coalescing, disjointedness, and the alignment of the pointers demarcating each of these sets.

Next I show that the concrete object heap contains the appropriate white, black and gray objects, and that each color of object contains references of the appropriate color (white contains white and mapped, black contains black and gray, and gray contains any of the above). This requires the usual reasoning about the well-formedness of copied or scanned well-formed objects, which should probably be factored out into lemmas.

I then use the fact that the concrete object heap is well-formed to help show that the entire concrete memory is well-formed. I must also use the various set equalities I previously proved.

Finally, I must show that the abstract memory contains copies of all of the black objects, and forwarded versions of the gray and white objects. This is true because the abstract memory is a forwarded version of the concrete object memory. Black

objects contain only to-space objects, so their fields are unchanged by the forwarding function. To prove this, I deploy my set of lemmas for reasoning about memories with their ranges mapped, as discussed in Section 7.4.3.

With that, and the simple checking of the register specifications, the verification of the main allocator block is complete.

7.4.4 Restore root verification

As I have discussed in previous sections, `restoreRoot` does not do very much: it loads the new value of the root back into the root register, then invokes the Baker collector loop. Nonetheless, the verification of the specification (given in Figure 7.14) is not trivial, though not as long as the rest of the allocator. The hard part here is showing that the state after the loop runs is a well-formed Baker state. In the end, though, this is almost identical to the proof for the case of the allocator when there is no need to flip, as described in Section 7.4.3. In fact, most of the proof is just copied and pasted directly from there. It would of course be more elegant to factor out the common proof into a lemma. This similarity is not surprising, as most of the proof is concerned with reconciling the loop's limited view of the state with the allocator's broader view of the state.

7.4.5 Specification weakening

I have now verified that the allocator matches the specification in Figure 7.13, but this is not the final specification I wish to give to the allocator, which is given in Figure 4.8. While these two specifications are at a similar level of abstraction, some of the details are different.

The first task is to show that the Baker state well-formedness predicate is equiv-

alent in strength to the Baker representation predicate that I have defined. In other words, I must show $\forall \mathbb{S}, \mathbb{A}, A.\, \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathit{rfileOf}(\mathbb{S}), \{\mathsf{root}\}) * A \leftrightarrow \mathit{bakerStOk}(\mathbb{A}, A, \mathbb{S})$.

The main difficulty of this proof is that each predicate treats the abstract memory in a different way. In *repr*, the abstract memory is a scanned version of the part of the concrete memory containing white, gray and black objects. *bakerStOk* is more fine grained, specifying that the abstract memory contains *copies* of black objects, and scanned versions of white and gray objects. So, I must show that the black objects are not affected by the forwarding map. This is the case because black objects cannot contain from-space pointers, which are the only values affected by the forwarding map. This is a bit tedious, but nothing fundamentally different from what has been done in the rest of the verification of the Baker collector.

The second task is to show that the postcondition of the allocator, *bakerStOkPost*, implies that the concrete state contains the representation of the abstract state with an additional object allocated. In other words, it must be shown that

$$\forall \mathbb{S}, \mathbb{A}, A.\ \mathit{bakerStOkPost}(\mathbb{A}, A, \mathbb{S}) \rightarrow \mathbb{S} \vdash \mathit{repr}(\mathbb{A}', \mathit{rfileOf}(\mathbb{S}), \{\mathsf{v0}, \mathsf{root}\}) * A$$

where $\mathbb{A}' = \mathbb{A}\{x \rightsquigarrow \mathrm{NULL}\}\{x + 4 \rightsquigarrow \mathrm{NULL}\}\{\mathsf{v0} \rightsquigarrow x\}$ and $x = \mathbb{S}(\mathsf{v0})$. To prove this, I use the equivalence between *bakerStOk* and *repr* I have already established, and show that I can add a root that contains a to-space pointer.

Combining the previous two facts, I can show that the precondition of the Baker allocator I have already verified is equivalent to the precondition of the allocation operation from the GC interface.

Next, I must show that the Baker allocator guarantee implies the GC interface's allocator guarantee. To do this, I first show that *bakerVirtGuar* implies the interface's GC guarantee. I must show that the reachable portion of a state is isomorphic to the

forwarded version of the reachable portion of the state, when the state is forwarded using an isomorphism. I do this by using the various lemmas about forwarded states I have previously used.

Once I know that the abstract guarantee is stronger, I can show that the actual guarantee is stronger. To do this, I have to apply the various lemmas I have just discussed, and prove that the newly allocated object is fresh.

After all of that, I use the SCAP instruction sequence weakening lemma to show that the allocation operation satisfies the specification given in the GC interface.

7.5 Read barrier

In this section I discuss the implementation and verification of the read barrier for the Baker collector. In the Cheney collector, a read is just a read. Verification then is purely a matter of showing the read operation respects and maintains the abstraction.

The Baker collector allows the mutator to run before the collector has finished. As a result, the invariants of the object heap are more fragile. As I discussed in Section 7.1, soundness of the Baker collector requires that all roots are either atomic or to-space pointers. However, objects that root values point to may contain stale from-space values, so to maintain this invariant the barrier must scan the field being read from before returning its value. Fortunately this can be implemented with the existing field scanning operation described in Section 7.2.

7.5.1 Implementation

The pseudocode implementation for the Baker collector read barrier is given in Figure 7.16. The barrier takes two arguments. x is the object the mutator is reading

```
void bakerRead (x, k) {
  scanField(&x[k]);
  return x[k];
}
```

Figure 7.16: Baker read barrier psuedocode

```
BAKERREAD(k) :
// load the registers we need for scanField
lw rfrStart,0(gcInfo)
lw rfrEnd,4(gcInfo)
lw rtoStart,8(gcInfo)
lw rtoEnd,12(gcInfo)
lw rfree,20(gcInfo)
lw ralloc,24(gcInfo)

// call scanField on the field we are reading
addiu t3,ra,0
addiu a0,root,k
addiu t6,a0,0
jal SCANFIELD, BAKERREADRET

BAKERREADRET:
// load the scanned value and return
lw v0,0(t6)
sw rfree,20(gcInfo)
addiu ra,t3,0
jr ra
```

Figure 7.17: Baker read barrier assembly implementation

from and k is the field offset. k is either 0 or 1, as objects are pairs. To execute the read barrier, the field the barrier is going to read from is scanned using the same field scanning method used in the collector. After this is done, the field will not contain a from-space value, and the barrier can return value of the field.

As usual, the real assembly implementation, given in Figure 7.17, contains more annoying details. The implementation is parameterized by a field offset `k`, which will either be 0 or 4. There is one read barrier for each possible offset. This makes things a little easier, but is not necessary.

The implementation first initializes the various registers that the field scanning method needs by loading their values from the garbage collector information record. After this is done, it prepares for a call to `SCANFIELD` by saving the return pointer ra to register t3 (as part of my rather ill-advised manual interprocedural register allocation

$$
\begin{array}{l}
\mathit{readPre}_k(\mathbb{S}) ::= \\
\quad \exists \mathbb{A}.\ \ \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, \{\mathsf{root}\}) * \mathtt{true} \wedge \\
\qquad\quad \mathbb{A} \vdash \mathbb{A}(\mathsf{root}) + k \mapsto - * \mathtt{true} \\
\\
\mathit{readGuar}_k(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall \mathbb{A}, A, R. \\
\qquad \mathbb{S} \vdash \mathit{repr}(\mathbb{A}, \mathbb{S}, R) * A \rightarrow \\
\qquad \mathsf{root} \in R \ \wedge\ R \subseteq \{\mathsf{root}\} \cup \mathit{calleeSaved} \rightarrow \\
\qquad \exists \mathbb{A}'.\ \mathit{gcStep}(\mathbb{A}, \mathbb{A}', R) \wedge \\
\qquad\quad \mathbb{S}' \vdash \mathit{repr}(\mathbb{A}', \mathbb{S}', \{\mathsf{v0}\} \cup R) * A \wedge \\
\qquad\quad \mathbb{A}' \vdash \mathbb{A}'(\mathsf{root}) + k \mapsto \mathbb{A}'(\mathsf{v0}) * \mathtt{true}) \ \wedge \\
\quad \forall r. \mathit{presReg}(r) \wedge r \neq \mathsf{v0} \rightarrow \mathbb{S}(r) = \mathbb{S}'(r)
\end{array}
$$

Figure 7.18: Baker read barrier specification

to avoid spilling registers), calculating the address of the field being scanned, then saving the pointer to this field to register t6 so it will be available after execution returns from field scanning.

The second code block BAKERREADRET loads the new value of the field into the return register v0, stores the new value of the free pointer (this is the only GC information record value that field scanning might change), restores the return register, then returns.

7.5.2 Specification

The specification of BAKERREAD is the standard specification of the read barrier, described in Section 4.4.2. I reproduce it here in Figure 7.18 for convenience. For the precondition, the basic idea is that the concrete state $\mathbb{S}$ contains the representation of the abstract state $\mathbb{A}$. In the abstract state, the field the mutator is reading from must be a valid pointer. The guarantee takes any abstract state that has a representation in the current state with root register set R and returns a concrete state that represents a new abstract state $\mathbb{A}'$. $\mathbb{A}'$ is related to the original abstract state by *gcStep*, which,

$$
\begin{array}{l}
\mathit{bakerReadRetPre}(\mathbb{S}) ::= \mathbb{S} \vdash\ \mathbb{S}(\mathsf{t6}) \mapsto -\ \ *\ \ \mathbb{S}(\mathsf{gcInfo}) + 20 \mapsto -\ \ *\ \ \mathtt{true} \\
\\
\mathit{bakerReadRetGuar}(\mathbb{S}, \mathbb{S}') ::= \\
\quad (\forall A, x, y. \\
\qquad \mathbb{S} \vDash \mathbb{S}(\mathsf{t6}) \mapsto x\ \ *\ \ \mathbb{S}(\mathsf{gcInfo}) + 20 \mapsto y\ \ *\ \ A\ \ \rightarrow \\
\qquad \mathbb{S} \vDash \mathbb{S}(\mathsf{t6}) \mapsto x\ \ *\ \ \mathbb{S}(\mathsf{gcInfo}) + 20 \mapsto \mathbb{S}(\mathsf{rfree})\ \ *\ \ A\ \ \wedge \\
\qquad \mathbb{S}'(\mathsf{v0}) = x)\ \ \wedge \\
\quad (\forall r \notin \{\mathsf{v0}, \mathsf{ra}\}.\ \mathbb{S}(r) = \mathbb{S}'(r)) \wedge \mathbb{S}'(\mathsf{ra}) = \mathbb{S}(\mathsf{t3})
\end{array}
$$

Figure 7.19: Baker read return specification

for copying collectors such as the Baker GC, states that all reachable objects are preserved, though they are moved around by some isomorphism. (The GC step for copying collectors is described in Section 4.4.1.) Furthermore, $\mathbb{A}'$ also contains, in register $\mathsf{v0}$, the value of the field the barrier is loading from.

In the mutator, the representation predicate *repr* is abstract, to ensure modularity. In the collector itself, it is concrete. The representation predicate for the Baker collector is described in Section 5.4. The main trick in the representation predicate is that any from-space values that have already been copied get forwarded whenever they appear in object fields. This is a variation on the basic Baker state well-formedness predicate *bakerLoopStOk′* defined in Figure 7.11.

The specification for the Baker read return block is given in Figure 7.19. This is not very interesting, and simply gives a low-level description of what happens in the block.

7.5.3 Verification

First the block must be stepped through. Showing that the block itself is safe to execute is direct, as it is loading from addresses that the precondition explicitly states exist.

Before I get into the difficult part of verification, I derive some basic facts from the representation predicate. For instance, the root is an object pointer, so it must be either a black or gray pointer. Also, the abstract and concrete roots have the same value. I also show that the to-space is made entirely up of black objects, gray objects and free objects.

Now I am ready to show that it is safe to call `SCANFIELD`. To do this I must derive some more facts from the precondition, such as that the from- and to-spaces are disjoint. I can show that the field being scanned is valid because it is the field of an object that is either black or gray, and consequently will either be atomic or contain a valid free- or to-space pointer.

After this is done, I must show the guarantee is valid. This begins in the way that many of these proofs do with a lot of tedious reasoning about sets and memory. I am dealing with a different abstract state than in the precondition, so I must reprove many of the same basic things. I also define an offset k' which is simply the offset of the other field. For example, if the barrier is reading from the first field, k is 0 and k' is then 4. This will allow me to combine the two cases for the two possible values of k as much as possible.

Next I show that the state before `SCANFIELD` is a well-formed field scanning state. I do this slightly differently than I did when I were showing that it is safe to call `SCANFIELD`. I separate the field the barrier is scanning from the rest of the to-space, which allows me to show that the entire to-space, excluding the field being scanned, is unchanged by field scanning. This must be done because eventually I will need to show that the initial and final states are related by an isomorphism. For the initial set of to-space objects, this isomorphism will be the identity function. To show that this extraction is okay, I must consider a number of cases depending on whether the root is black or gray. I can, however, reflect all of these cases in a single final result.

For instance, if the set of black objects is B, then the set of black objects that is unchanged by the field scanning operation is $B - \mathbb{S}(\mathsf{root})$. If the root is black, this is B without the root. If the root is not black, this is just B. After this is done, I can apply the guarantee of `SCANFIELD`.

After the field has been scanned it must contain a forwarded value. Additionally, some objects may have been copied, and thus moved from the set of white objects to the set of mapped objects. All of the to-space objects, aside from the one being read from, are unchanged. From the state after the call, I extract (almost) the new concrete object heap, which is the part of the concrete memory that contains the remaining white objects, along with all of the allocated to-space objects. I do not extract quite the full concrete object heap because I do not include the field that the barrier has just scanned. This will be read by the read return block, so it must be left alone for the moment. Then I can apply the read return block guarantee, then add the scanned field back into the concrete state.

After this, there is some tedious reasoning to show that the initial set of to-space objects (aside from the root) and the final set of white objects are not changed. The abstract object heap is the concrete object heap, as described in the previous paragraph, with stale from-space pointers forwarded.

I then prove that the new value of the field being read from is a valid to-space value, because it is the result of applying the forwarding map, which has a range that does not include from-space objects.

At this point, there are three major subgoals remaining. First, I must show that the abstract state respects the GC guarantee. Next, I must show that the concrete state contains a representation of the abstract state. Finally, I must show that the abstract return register contains the value of the abstract object the barrier is reading from. I verify the first two of these three subgoals using lemmas. The

final subgoal, regarding registers, is simple and the same as my various prior proofs involving registers, so I will not discuss it further.

Respect the abstract GC guarantee

The proof that the abstract states respect the guarantee picks up where I left off. I do not reproduce the lemma here because merely stating the lemma takes about 60 lines. The proof itself is around 1400 lines of Coq proof script. This proof is very similar to the parts of the verification of the Cheney collector that deal with verifying that the collection respects the GC guarantee.

The read barrier is simpler than the collector, because less is happening. There are only two changes. First, one field of the root object may change. Second, some white objects may be copied, becoming gray. Everything else is unchanged. The isomorphism relating the old and new objects is the identity, except for the white objects that are newly copied. In this case, the isomorphism is the map generated by the field scanning.

Representation okay

For this lemma, I have to show that the final concrete memory contains a representation of the final abstract state, and that the abstract value returned by the read barrier is indeed the value of the kth field of the object the barrier is reading from.

The proof of this lemma opens by showing a variety of basic properties, such as that various sets of objects are disjoint in the expected way, that the gray objects cover the entire space from the scan pointer to the free pointer, and that the root must be a to-space object.

One part of this proof is showing that the new concrete memory is "well-colored" after the field the barrier is reading from is scanned (and consequently has its value

```
void write (void* root, int k, void* x) {
  root[k] = x;
}
```

Figure 7.20: Baker write barrier pseudocode

updated): white objects can contain only pointers to from-space objects, black objects can contain only pointers to to-space objects, and gray objects can contain pointers to either. The root object is always a to-space object (which can contain any valid to-space pointer), and scanning a field always results in a black field, which is valid in any to-space object. The other interesting case is the objects that have been copied from the to-space. For these objects, I make use of a lemma I have used many times before that says that copying an object does not change the colors of the objects it contains.

To show that the barrier has read the right abstract value, I first show that the value being returned is the same value v of the kth field of the root object in the concrete state. Also, the concrete memory must be related to the abstract memory by a forwarding function ϕ. Because the forwarding function only changes from-space values, and because v must be a to-space value, $\phi(v) = v$. Therefore v is also the value of the kth field of the root in the abstract state, so the barrier is returning the right value.

I have shown that all of the cases hold, so I have successfully verified the read barrier.

7.6 Write barrier

Now I will describe the write barrier for the Baker collector. In the Baker collector, roots cannot contain from-space pointers (thanks to the read barrier). Therefore,

any storing of a pointer into an object will be writing a to-space pointer into the field of a to-space object. This is perfectly valid to do, so the write barrier does not need to do any additional work.

As a consequence, the Baker write barrier is, like the Cheney write barrier, just a store instruction. I repeat the pseudocode implementation of the barrier in Figure 7.20. The assembly implementation (which is a single instruction) is the same as the Cheney collector write barrier given in Figure 6.34.

The specification of the write barrier is that standard specification given in Chapter 4. Now all that remains is to verify that the implementation matches the specification. In essence, this is formally checking the belief that the write barrier does not have to do any additional work to preserve the heap abstraction for the mutator.

7.6.1 Verification

First I show that the precondition of the write barrier implies that the kth field of the object the barrier is writing to is a valid memory location in the concrete state. This is enough to show that the write barrier is safe.

With that done, I must show that the write barrier satisfies its guarantee. The abstract state of the final state is the initial abstract state with the value of the kth field set to be the abstract value the barrier is writing to that field. The guarantee includes a collector step, but the write barrier does not do anything, so I use a lemma to show that doing nothing is also a collector step.

7.7 Putting it all together

As with the Cheney collector, there is some additional work required even after I have verified all of the basic blocks in the program. First, I must show the properties

component	lines
Baker specific lemmas	3162
field scanning	2032
collector loop	3266
allocator	4258
read and write barriers	3176
top level	1039
total	16933

Figure 7.21: Baker line counts

of the representation predicate as described in Section 4.3.3. These proofs are all very similar to those in the Cheney collector, because the properties deal with roots, and roots are represented the same way in both collectors.

7.8 Conclusion

In this chapter, I have discussed my machine-checked proof of the soundness of an implementation of the Baker garbage collector algorithm. As far as I am aware, this is the first machine-checkable proof of the Baker collector. Line counts for the various components of the Coq implementation appear in Figure 7.21. These line counts include white space, comments, assembly implementations, specifications, and proofs, but the overwhelming majority of the lines are proofs. Not included are the lemmas that are shared with the Cheney collector proofs, which are less than 1800 lines in total. The compiled proof files for the Baker collector are about 11 megabytes large, though that measure might not be very meaningful.

The incremental nature of this algorithm greatly complicates the verification, even though at a glance the algorithm is fairly similar to the Cheney collector. The memory is generally in a complicated partially copied state, and requires a read barrier. Despite this, I am able to give my Baker collector implementation the

same interface as the Cheney collector. Thus to the mutator the two algorithms are indistinguishable. This demonstrates the ability of my approach to garbage collector interfaces to abstract away details of the implementation.

Chapter 8

Tools and Tactics

8.1 Introduction

The verification of the collectors discussed in Chapters 6 and 7 requires complex reasoning. Without powerful tactics for dealing with machine semantics and separation logic, these proofs would be impractical. In this chapter, I discuss some of the techniques I developed to simplify the verification of the collectors, which I have implemented in the Coq proof assistant [Coq Development Team 2007b]. Without these tactics, the verification of the garbage collectors I have described would be impractical.

In Coq, theorems are proved interactively. The system shows the user the current goal and the current set of hypotheses. The user then selects a tactic to be executed. This transforms the goal and/or hypotheses, hopefully progressing towards completion.

At their simplest, tactics correspond to introduction or elimination rules in logic. For instance, if the goal is $A \wedge B$, the tactic `constructor` applies the introduction rule for conjunction, producing two subgoals A and B. Similarly, if there is a hypothesis

`H` that is a proof of $C \wedge D$, then the tactic `elim H` will produce two new hypotheses, one a proof of C and the other a proof of D.

On top of these basic tactics is the tactic language L_{tac} that is a simple untyped imperative language (as the core tactics are fundamentally imperative) with features such as control flow structures, sophisticated pattern matching of goals and hypotheses, exceptions and function definitions. L_{tac} enables implementing sophisticated tactics out of more primitive ones.

All of the tactics described in this chapter are part of the full Coq implementation available online [McCreight 2008].

8.2 Machine semantics

As described in Chapter 2, I reason about assembly programs using the program logic WeakSCAP. With WeakSCAP, the first step in verifying that a program block matches a particular specification is relating the entry state to the exit states. At the same time, the individual instructions must be safe to execute. Once the final state in the block is defined in terms of the initial state, reasoning can be done regarding the guarantee. For instance, if I have an instruction sequence $c; c'; c''; \mathsf{jr\ ra}$ that I wish to show has the specification (p, g), I must show that for all initial states $\mathbb{S}$ such that $p\ \mathbb{S}$ holds, there is some $\mathbb{S}'$ such that $\text{cstep}_{c''}(\text{cstep}_{c'}(\text{cstep}_{c}(\mathbb{S}))) = \mathbb{S}'$, and that furthermore $g\ \mathbb{S}\ \mathbb{S}'$ holds.

8.2.1 Command step simplifications

The first task is to reduce the series of csteps. To this end, I prove a series of lemmas for reasoning about command steps for the various instructions. For every instruction except except lw and sw, these lemmas are fairly trivial, because these operations

are always safe and are derived fairly directly from the operational semantics:

$$\text{cstep}_{\mathsf{addu}\ r_d,r_s,r_t}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) + \mathbb{S}(r_t)\}$$

$$\text{cstep}_{\mathsf{addiu}\ r_d,r_s,w}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) + w\}$$

$$\text{cstep}_{\mathsf{subu}\ r_d,r_s,r_t}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r_s) - \mathbb{S}(r_t)\}$$

$$\text{cstep}_{\mathsf{sltu}\ r_d,r_s,r_t}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \text{if } \mathbb{S}(r_s) < \mathbb{S}(r_t) \text{ then 1 else 0}\}$$

$$\text{cstep}_{\mathsf{andi}\ r_d,r_s,1}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \text{if } isOdd(\mathbb{S}(r_s)) \text{ then 1 else 0}\}$$

There are also two specialized versions of the rule for addiu when one of the operands is equal to 0:

$$\text{cstep}_{\mathsf{addiu}\ r_d,r,0}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow \mathbb{S}(r)\}$$

$$\text{cstep}_{\mathsf{addiu}\ r_d,\mathsf{r0},w}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow w\}$$

For load and store, I want to relate the high-level separation logic description of memory to the low-level dynamic semantics (and later, vice versa). These are a slightly higher level wrapper around the separation logic properties given in Section 2.5.3.

For load instructions, the rule is:

$$\text{If } \mathbb{S} \vdash (\mathbb{S}(r_s) + w) \mapsto w' \ * \ \texttt{true}, \text{ then } \text{cstep}_{\mathsf{lw}\ r_d,w(r_s)}(\mathbb{S}) = \mathbb{S}\{r_d \rightsquigarrow w'\}$$

In other words, if I can demonstrate using separation logic that the memory of the current state $\mathbb{S}$ contains a value w' at the address being loaded from $(\mathbb{S}(r_s) + w$), then carrying out the load instruction is the same as setting r_d to w'. Memory can contain anything else, signified by `true`.

The rule for store instructions is similar:

$$\text{If } \mathbb{S} \vdash (\mathbb{S}(r_s) + w) \mapsto - \ * \ \texttt{true}, \text{ then } \text{cstep}_{\mathsf{sw}\ r_s,w(r_d)}(\mathbb{S}) = \mathbb{S}\{\mathbb{S}(r_s) + w \rightsquigarrow \mathbb{S}(r_s)\}$$

If I can demonstrate using separation logic that the memory of the current state $\mathbb{S}$ contains some value at address $\mathbb{S}(r_s) + w$, then carrying out the store instruction is the same as setting the value of that address to $\mathbb{S}(r_s)$.

The actual rules I use in the implementation are slightly different, because they add a level of indirection. For instance, the precondition of the full load rule is $\mathbb{S} \vdash x \mapsto w' * \texttt{true}$, with an additional condition that $x = \mathbb{S}(r_s) + w$. This allows more flexibility when applying the rewriting rule. For instance, if I were attempting to load from an offset of 4, and had a proof that $\mathbb{S} \vdash \mathbb{S}(r_s) + 2 + 2 \mapsto 7 * \texttt{true}$, I could apply the rewriting lemma directly using this proof, then show that $\mathbb{S}(r_s)+2+2 = \mathbb{S}(r_s)+4$ as a side condition instead of manipulating the existing proof. The tactic attempts to automatically solve such side goals, so in the simple case there is no extra work.

8.2.2 State update simplifications

In the course of verifying an instruction sequence, I must reason about the values of register and memory locations after a series of commands have been executed. To simplify these expressions, I have a set of rewriting rules I use with Coq's `autorewrite` tactic, that repeatedly invokes a set of rewriting rules until no more change is made. The simplifying tactic is called `stSimpl`.

Here is a list of the major rules used, along with side conditions and an explanation of the rule:

$\mathbb{S}(\mathsf{r0}) = 0$		$\mathsf{r0}$ is always 0
$(\mathbb{S}\{l \rightsquigarrow w\})(r) = \mathbb{S}(r)$		changing mem. does not affect the reg. file
$(\mathbb{S}\{r \rightsquigarrow w\})(w) = \mathbb{S}(w)$		changing the reg. file does not affect mem.
$(\mathbb{S}\{r \rightsquigarrow w\})(r) = w$	if $r \neq \mathsf{r0}$	get a value just set
$(\mathbb{S}\{r \rightsquigarrow w\})(r') = \mathbb{S}(r')$	if $r \neq r'$	ignore the setting of other registers

Now I will give an example of how and why `stSimpl` is used. Say I have the following assembly block:

```
addu r1,r2,r3;
addu r4,r5,r6;
jr ra
```

I want to reason about the values of each register after executing this block, in terms of the state I had when entering the block. Assuming the initial state is $\mathbb{S}$, using the standard command step reductions from Section 8.2.1 the state after the first step will be $\mathbb{S}\{\mathsf{r1} \mapsto \mathbb{S}(\mathsf{r2}) + \mathbb{S}(\mathsf{r3})\}$, so the final state $\mathbb{S}'$ is $\mathbb{S}\{\mathsf{r1} \mapsto \mathbb{S}(\mathsf{r2}) + \mathbb{S}(\mathsf{r3})\}\{\mathsf{r4} \mapsto (\mathbb{S}\{\mathsf{r1} \mapsto \mathbb{S}(\mathsf{r2}) + \mathbb{S}(\mathsf{r3})\})(\mathsf{r5}) + (\mathbb{S}\{\mathsf{r1} \mapsto \mathbb{S}(\mathsf{r2}) + \mathbb{S}(\mathsf{r3})\})(\mathsf{r6})\}$. This is quite a mess! With more instructions in a block, things get even worse.

But `stSimpl` is able to immediately show that $\mathbb{S}'(\mathsf{r31}) = \mathbb{S}(\mathsf{r31})$, because r31 was not changed during execution of the block, and that $\mathbb{S}'(\mathsf{r4})$ is equal to $\mathbb{S}(\mathsf{r5}) + \mathbb{S}(\mathsf{r6})$, because although r4 is changed from the initial value, the operands of the sum that it is set to are not changed during the block.

8.3 Finite sets

My garbage collector specifications are defined in terms of various finite sets of objects, so the proofs involve the extensive use of finite sets. I represent finite sets as lists: if a value is in the list, it is in the set. A value can occur multiple times. The basic idea is taken from the ListSet library of Coq version 8.0 [Coq Development Team 2007a], though they do not apply it consistently. My library also has many more operations and predicates than theirs, though I probably am missing some they have. Coq version 8.1 [Coq Development Team 2007b] has a more extensive finite set library, but it was not available when I began my collector work.

I implement a variety of operations on sets, including membership testing, addition, removal, union, difference, intersection, cardinality, map, "multi-map" (each set

element is mapped to a set of elements), and filtering. These operations have all of the introduction and elimination lemmas one would expect. For instance, the introduction rule for `set_remove` says that if $a \in S$ and $a \neq b$, then $a \in$ `set_remove` b s. The elimination rule says that if $a \in$ `set_remove` b s, then $a \in S$ and $a \neq b$. This style of lemma is taken from the Coq ListSet library.

I also implement canonization for my set representation. Two sets are *equivalent* if they contain the same elements. The canonical forms of two sets that are equivalent are structurally equal. Canonization is implemented with a duplicate-removing insertion sort of the set. The main (and perhaps only) use of this is to allow me to define the iterated separating conjunction (see Section 2.6) as a fix point on a canonical set instead of as an inductive definition, to simplify proofs.

Set cardinality is needed to verify the Cheney collector, because I maintain the invariant that the number of free objects is never smaller than the number of uncopied from-space objects.

In addition, I implement a number of set predicates, including equivalence (both sets contain exactly the same elements), disjointedness, membership, subsets, and isomorphisms from one set to another.

I use set isomorphisms extensively when reasoning about copying collectors. In fact, my goal is to verify that the collector maintains an isomorphism between the sets of reachable objects in the initial and final states. I have lemmas about, for instance, the unions of isomorphisms and the transitivity of isomorphisms.

For set equivalence, I have various rules about symmetry and distribution over many of the operations.

I make extensive use of the Coq setoid rewriting mechanism to reason about set equivalence. Setoid rewriting allows the user to define their own notions of equality, prove that some operations respect that equivalence, then do "rewriting" using the

custom notion of equality as if it were the standard notion of equality in Coq.

8.4 Separation logic

I implement a separation logic library using a weak embedding. The weak embedding allows me to easily extend the logic with new predicates as needed. I have a variety of tactics for manipulating proofs in this embedded logic. This is needed because the separation logic predicates I need to describe the garbage collectors can get very large.

I use a standard series of steps to prove a large separation logic goal:

1. The hypothesis and goal are each simplified.
2. Hypothesis and/or goal are reordered and regrouped.
3. Corresponding parts of hypothesis and goal are matched to produce smaller subgoals.
4. In each subgoal, basic lemmas further break down goal and/or hypothesis.
5. If not solved, go back to step 1.

I will now describe the tactics used in each of these phases.

8.4.1 Simplification

There are two simplification tactics, `linLogIntro` and `linLogElim`. The primary goal is to render the goal (in the case of `linLogIntro`) or the hypothesis (in the case of `linLogElim`) into a "right normal form" (RNF) with respect to the separating conjunction operator $*$. In other words, it will transform things into proofs of the form $A * B * C * D$. ($*$ is right associative.) This makes further manipulations easier.

In addition, these tactics combine all instances of the trivial separation logic predicate `true` and eliminate all instances of the empty separation logic predicate `emp`, including those of the form $\forall_* x \in \emptyset.\ P$. They also extract parts that do not involve memory, which are indicated by the separation logic operators ! and $\exists$. In a hypothesis, these are split into separate hypotheses, while in the goal, these are split into separate goals. Finally, `linLogElim` solves the goal immediately when the hypothesis contains something of the form $x \mapsto y * x \mapsto z * A$.

Here is an example of simplification. Consider the hypothesis:

$\mathbb{M} \vDash \exists x : \mathit{Nat}.\ !(x > 0) * (x \mapsto - * (\texttt{emp} * \texttt{true} * A)) * \texttt{true} * \texttt{true}$

This would be simplified into four hypotheses: $x : \mathit{Nat}$, $y : \mathit{Nat}$, a proof that $x > 0$, and finally

$\mathbb{M} \vDash x \mapsto y * A * \texttt{true}$

Recall that $x \mapsto -$ is an abbreviation for $\exists y.\ x \mapsto y$, so the simplification will eliminate that existential.

8.4.2 Matching

I describe matching before reordering and regrouping, even though it is done last, because it gives context to those phases. The tactic `sconjMatch` attempts to match an RNF goal to an RNF hypothesis, breaking the problem into subgoals by matching the respective sub-predicates. For example, if the goal is $\mathbb{M} \vDash A * B * C$, and there is a hypothesis of the form $\mathbb{M} \vDash A' * B' * C'$, then the tactic `sconjMatch` will produce three subgoals:

1. $\forall \mathbb{M}.\ \mathbb{M} \vDash A \to \mathbb{M} \vDash A'$
2. $\forall \mathbb{M}.\ \mathbb{M} \vDash B \to \mathbb{M} \vDash B'$
3. $\forall \mathbb{M}.\ \mathbb{M} \vDash C \to \mathbb{M} \vDash C'$

Any trivial subgoals are solved. For instance, if $C = C'$, or $C' = \texttt{true}$, then only the first two subgoals would be generated. This tactic relies on a lemma that says that if $A \Rightarrow A'$ and $B \Rightarrow B'$, then $(A * B) \Rightarrow (A' * B')$.

8.4.3 Reordering

As I said in the previous section, if I have a hypothesis $\mathbb{M} \vdash A * B * C$ and a goal $\mathbb{M} \vdash D * E * F$, then the tactic will match A to D, B to E and C to F. However, I do not want this if the part of memory described by A is not the same as the part described by D. In this case, I need to reorder either the goal or the hypothesis, so that the parts match up appropriately. If A, B and C correspond to E, F and D, respectively, then I need to transform the hypothesis to $\mathbb{M} \vdash C * A * B$ for the matching to work correctly.

How do I do this? The underlying method is to apply a series of associativity and commutativity lemmas to the hypothesis. I can get from $\mathbb{M} \vdash A * B * C$ to $\mathbb{M} \vdash (A * B) * C$ by associativity, then to the final goal of $\mathbb{M} \vdash C * A * B$ by commutativity.

I initially applied these basic lemmas by hand, but in practice, memory predicates have many parts (sometimes 8 or more) and these manipulations become tedious. The next refinement of reordering was to prove a series of "lifting" lemmas. `sconjLift2`, for instance, will transform a proof like $\mathbb{M} \vdash A*B*C$ into $\mathbb{M} \vdash B*A*C$ in a single step, by moving the second component of the predicate to the front while leaving the rest of it alone. Using this approach, I perform the example transformation to $\mathbb{M} \vdash C * A * B$ in a single step using `sconjLift3of3`. This works fairly well, and in fact is the approach I used during most of the development of my GC proofs.

This lifting-based approach is useful when there are only a few changes to be made, but becomes cumbersome when I need to make many changes. For instance,

if I were attempting to transform $\mathbb{M} \vdash A * B * C$ into $\mathbb{M} \vdash C * B * A$, this would require 2 steps: lifting the third element, then lifting the third element again. To be able to do any reordering in a single step, a more complex method is needed.

To handle this situation, I developed a third and final refinement of the reordering tactics to allow arbitrary reordering in a single step. The desired ordering is indicated by a list of natural numbers. The kth element of the list indicates the desired final position of the kth element of the separating conjunction. For example, the list $[3, 2, 1]$ would transform $\mathbb{M} \vdash A * B * C$ into $\mathbb{M} \vdash C * B * A$. No attempt is made at error checking: if $[3, 2, 1]$ was used to permute $\mathbb{M} \vdash A * B$, then the tactic might fail or produce bizarre output.

The implementation of this tactic is heavyweight, but it works. The intuition is that a proof that one list is a permutation of another can be interpreted as a series of steps to transform one list into the other. I can define a sorting function and prove that a list is a permutation of its sorted form. This proof is then used to generate a permutation proof to drive the basic tactic.

Extensions It would be more efficient to derive the reordering directly, instead of via a proof term. Another improvement would be adding error checking, as the sorted version of the supplied list should have the form $[1, 2, 3, ..., k-1, k]$. If it does not, the result of applying the tactic will not make much sense.

8.4.4 Reassociation

In addition to reordering, one may wish to change how the predicates are associated. If I am trying to prove $\mathbb{M} \vdash A * B * C$ given $\mathbb{M} \vdash D * E$, it may be the case that the part of memory described by D corresponds to the part described by $A * B$. To ensure that things are matched up appropriately, I must transform the goal into

$\mathbb{M} \vdash (A * B) * C$. This will give me subgoals $A * B \Rightarrow D$ and $C \Rightarrow E$.

As before, it is possible to do these transformations manually, and I did, in much of my GC development. However, it is nice to have a way to quickly and declaratively change the associations in predicates. I proceed much as I did for reordering: create a Coq term that describes the desired transformation and then write a tactic that analyzes the Coq term to carry it out. However, reassociation is simpler because it does not need an intermediate derivation.

Instead, it works on inductively defined binary trees. A tree t is either a leaf containing a natural number k or a node containing two subtrees:

$$t ::= \texttt{leaf}(k) \mid \texttt{node}(t, t)$$

The values in the nodes are ignored by the reassociation tactic, but will be useful later when I combine reassociation with reordering.

A reassociation tactic takes a binary tree t and transforms the RNF goal or hypothesis to match the shape of the tree. The leaves in the tree correspond to the base memory predicates, while the nodes correspond to $*$ operators. In addition, the tactic takes an additional argument `isLeft` which is true only if the tactic has followed a left branch to get to the current node. In the leaf case, nothing needs to be done. In a node case, there are two subtrees, t_1 and t_2. First, the tactic is invoked on t_1, with `isLeft` set to `true`, because t_1 is a left branch. This recursive call ensures that the goal is of the form $\mathbb{M} \vdash A_1 * A_2$, where A_1 corresponds to t_1. Next, the tactic is recursively invoked on the right subtree t_2, with `isLeft` unchanged from the current invocation, because this is a right branch.

Finally, the predicates must be grouped together. When the tactic returns, the first memory predicate must correspond to $\texttt{node}(t_1, t_2)$. If `isLeft` is `true`, then

```
Ltac assocConjG' T isLeft :=
  match eval compute in T with
    | leaf _ => idtac
    | node ?T1 ?T2 =>
      assocConjG' T1 true;
      eapply sconjImplR;
        [idtac |
          let h := fresh "h" in
          let Hp := fresh "Hp" in
            (intros h Hp; assocConjG' T2 isLeft; apply Hp)];
      match eval compute in isLeft with
        | true => apply sconjAssocL
        | false => idtac
      end
  end.

Ltac assocConjG T := assocConjG' T false.
```

Figure 8.1: Association tactic

the tactic has followed a left branch. This means that the current goal is of the form $\mathbb{M} \vdash A_1 * A_2 * B$, where A_1 corresponds to t_1 and A_2 corresponds to t_2, and B has not yet been examined, because it includes parts of the right children of parents of the current node. Applying the association lemma transforms the goal to $\mathbb{M} \vdash (A_1 * A_2) * B$, and it is safe to return. If `isLeft` is `false`, then this is a finished rightmost subtree, so no more work remains to be done. Thus the goal looks like $\mathbb{M} \vdash A_1 * A_2$, and the tactic can return without doing any further work. The Coq definition of this tactic is given in Figure 8.1. As before, there is a similar tactic for hypotheses.

8.4.5 Reordering and reassociation

Finally, both reordering and reassociation can be carried out with a single tactic `aspConjG` (or the similar `aspConj` for hypotheses). The combination is described by

a binary tree with natural numbers at the leaves. The leaves give the ordering for the sub-predicates, while the tree gives their final shape. This tactic first flattens the tree into a list, then uses the reordering tactic. Next, it uses the initial tree to reassociate.

I use a couple of Coq's facilities to make writing these trees simpler. First, I declare that `leaf` is a *coercion* from natural numbers to binary trees. Thus if I write 4 when a tree is expected, Coq will automatically treat it as if I had written `leaf`(4). I also declare a *notation*, so that `[|`t_1, t_2`|]` is treated as though I had written `node(`t_1, t_2`)`.

Putting all of these things together, I can give an example of the use of these tactics. If the goal is $\mathbb{M} \vDash A * B * C * D * E * F * G$ then the tactic

```
aspConjG ([|7, [|6, [|[|[|5, 4|], 3|], [|1, 2|]|]|]|])
```

automatically transforms the goal to $\mathbb{M} \vDash G * F * ((E * D) * C) * A * B$. There is room for improvement in the syntax.

8.5 Related work

There is a lot of existing work on mechanizing separation logic and developing tools for automated reasoning about programs using separation logic. In this section, I give an overview of some of this work.

Weber [2004] describes a shallow embedding of separation logic in the theorem prover Isabelle/HOL, and uses this to define three separate Hoare logics for a C-like language, and prove soundness and relative completeness of the program logics. The paper also describes an example verification of in-place list reversal. It is not clear

what degree of automation is achieved.

Smallfoot [Berdine et al. 2005] is a tool for automatically verifying C-like programs using separation logic. It favors automation over expressiveness. Smallfoot can reason about data structures, but it is limited to a couple of types of linked lists and trees, and requires quantifier-free assertions. For this reason, it is not sufficient to reason about garbage collectors, which tend to have complex data structures, but adopting some of their techniques to my more general purpose tool set might greatly simplify some portions of the proofs. An OCaml implementation is available [Smallfoot Development Team 2005], but the underlying theory does not appear to be mechanized.

Marti et al. [2006] describe the formal verification of a heap manager in Coq using separation logic. They also use a shallow embedding of separation logic. They have an automated tactic for doing separation logic proofs that they describe as being similar in power to Berdine et al. [2005] "without inductively defined datatypes". This automated tactic is certified in Coq, and is described in greater detail in Marti and Affeldt [2007]. Using this tactic reduces one proof to a third of its size compared to their manual proofs. However, this automated tactic has similar limitations of expressiveness as Smallfoot, so manual proofs are still required when verifying programs with complex memory invariants. Their manual proofs about separation logic predicates appear to be lower level than mine, breaking down separating conjunctions and reasoning about the disjointedness of memories. By contrast, I rarely need to do such reasoning outside of the proof of basic properties such as the commutativity and associativity of $*$.

Appel [2006] also develops a set of Coq tactics for verifying programs using separation logic. This work shares with my approach an emphasis on simplifying manual proofs, rather than attempting to develop fully automated proving for a limited frag-

ment of separation logic. This work is more automated than mine, able to match up separation logic hypotheses with goals that differ only in the application of associative and commutative rules. Such an addition would be useful for my system. This approach to reasoning resembles traditional Hoare logic more closely than mine: at each step, the goal is a precondition, a program, and a post-condition. By contrast, in my approach, the current state is described by a hypothesis, and a goal involves the evaluation of the current program along with the post-condition. My goal is to pull as much reasoning out into the broader Coq environment as possible, with the aim of simplifying reasoning.

8.6 Conclusion

In this chapter, I have given an overview of the tools and techniques I have developed to verify programs in Coq using separation logic. I believe that without these tools my task would have been more difficult or impossible. The complex specifications required by garbage collectors have led me to emphasize streamlining manual proofs, rather than focusing on the fully automated solving of simple goals. At the same time, some parts of my proofs do not require this level of complex reasoning, and adopting methods from related work would likely further ease verification.

Chapter 9

Conclusion

A variety of languages such as Java and ML use strong type systems or other means of static verification to ensure that programs are well-behaved. But one does not truly know that the implementation of such a program is safe unless one knows that the runtime system has certain safety properties. One of the most complex parts of a runtime system is the garbage collector. While there is extensive existing work on the verification of garbage collectors, it generally either does not deal with implementations, or does not allow the verification of properties stronger than memory safety.

In this dissertation, I have developed an expressive, formalized interface for reasoning about mutator-garbage collector interaction and demonstrated that this interface hides enough implementation details to be used with more than one collector, including a collector with a read barrier. The central idea of the interface is to treat the entire garbage collected heap as an abstract data type. This approach only hides implementation details that do not directly affect the mutator, such as the layout of auxiliary GC data structures. Other implementation details, such as the representation of the root set, cannot be hidden in this way. I discussed the realization of this

interface for mark-sweep, copying and incremental copying collectors.

My mechanized verification of implementations of Cheney and Baker garbage collectors demonstrates that this is practical. As discussed, this is the first mechanized verification of the Baker GC that I am aware of. I also discussed how I verified that the implementations of these two collectors properly implement my ADT-based interface. By enabling the verification of both mutator and collector within a single formal system, the safety of an entire garbage collected program can be verified.

I also gave an overview of some of the tools that made my verification efforts possible. These include the development of the WeakSCAP program logic, a weakest-precondition based variation of an existing program logic SCAP, and a large number of tactics developed to reason about machine semantics and separation logic.

9.1 Related work

In this section, I will discuss work related to the big-picture focus of the dissertation, which is garbage collector verification. I have previously discussed work related to other aspects of my dissertation. Chapter 2 discusses the large body of work I have drawn on for reasoning about programs and memory. In Section 3.6 I discussed work related to my approach to reasoning about ADTs, and in Section 8.5 I discussed work related to my tools for reasoning about separation logic.

McCreight et al. [2007] discuss much of the same work described in this thesis, although in less detail. It also describes how my approach can be used to verify a mark-sweep collector. Lin et al. [2007] apply the approach described in this dissertation to a mutator verified with typed assembly language, combined with a conservative mark-sweep garbage collector.

There is a lot of prior work on verifying garbage collector algorithms [such as

Gries 1977; 1978, Dijkstra et al. 1978, Hudak 1982, Ben-Ari 1984, Gonthier 1996]. Some of this prior work even uses machine-checked proofs [such as Russinoff 1994, Jackson 1998, Havelund 1999, Nieto and Esparza 2000, Burdy 2001]. Prior work differs from my work primarily in that they focus on algorithms rather than implementations. As a consequence, their machine models are less realistic than mine, and often do not include as many details. The drawback of this is that errors may arise during the translation of an algorithm into an implementation that can actually be used. Their work also does not seem to generally emphasize the separate verification of the mutator and collector, often modeling the mutator as a process that non-deterministically permutes memory. In addition, their work generally focuses on reasoning about a single algorithm, instead of a variety of algorithms as I attempt.

My work might be improved by drawing on their work, as verifying a GC implementation might be more efficient if it is done by relating a verified GC algorithm to an implementation. This might allow the implementation to be changed without requiring a lot of changes to the proof. In contrast, my work as described here is essentially verifying the algorithm at the same time as the implementation.

On the other hand, their work on verifying garbage collector algorithms has been successfully applied to verifying liveness properties and concurrent algorithms, which is not possible in my present framework. Liveness properties include termination and showing that a collector eventually collects all garbage. I can show that a collector collects all garbage with stop-the-world collectors, but not incremental collectors.

Another approach, taken by Vechev et al. [2006], is to start with a very high level collector and apply a series of correctness-preserving transformations to produce a more optimized collector algorithm. While they are in effect reasoning about a number of different algorithms, all of the collectors they deal with are concurrent mark-sweep collectors. Similar work by Vechev et al. [2007] uses model checking as

part of a system that is able to automatically check the correctness and efficiency of over a million variations of a single concurrent mark-sweep algorithm. Their work deals with more algorithms than mine, but the variation between these algorithms is much more limited.

Morrisett et al. [1995] discuss a high level semantics of garbage collection, which is similar to my work in that it involves reasoning about interference between the mutator and the garbage collector. They study more radical notions of garbage, including type-based approaches.

My work builds on that of Birkedal et al. [2004], which discusses the verification of a Cheney copying collector implemented in a C-like language using separation logic. They also use a heap isomorphism to describe copying, based on Calcagno et al. [2003]. However, they do not show how to verify the mutator in the presence of the garbage collector, nor do they discuss a high-level collector interface. They also do not show how to reason about a read barrier, and their proofs do not appear to be machine-checked.

Along with the work focusing on the verification of collectors, there is also some work that focuses on reasoning about a program in the presence of garbage collection. The main shortcoming of their work relative to mine is that they do not present systems capable of reasoning about both mutator and collector. On the other hand, my mutator interfaces could likely be improved by taking ideas from their work. Calcagno et al. [2003] present a program logic that is able to hide collection entirely from mutator-side reasoning, by making it impossible for specifications to refer to unreachable objects. Hunter and Krishnamurthi [2003] extend a formal model of Java with garbage collection and show that doing so is sound. Vanderwaart and Crary [2003] present a type system that can describe the collector interface for a precise GC.

There also is extensive prior work that focuses on using a single type system to reason about both the mutator and collector [Wang and Appel 2001, Monnier et al. 2001, Monnier and Shao 2002, Hawblitzel et al. 2007]. By using a type system to verify the collector, most of their work is thus able to give a well-defined interface between the mutator and collector: the type of the collector. However, with the notable exception of Hawblitzel et al. [2007], each type system seems to be fairly specialized to a particular collector and does not give a high level interface independent of a particular algorithm. Type systems usually excel at this sort of abstraction so it would likely not be difficult to add. A more difficult problem is that their work is fairly restricted in the notions of safety that it is able to reason about. Verifying that a GC adheres to a stronger property than memory safety would likely need a more complex type system, which would in turn require a new proof of safety.

Hawblitzel et al. [2007] use the type-based approach closest to my work. They use a linear logic based type system to mechanically verify the safety of a Cheney copying collector. Their system blurs the line between a type system and a Hoare logic because it includes explicit proofs. Their collector works with more complex objects that store header information. However, they have an unnecessary bounds check inside of their collector that I am able to show can be safely eliminated. They may be able to remove this check, but it is still not going to be possible for them to prove a safety condition as strong as showing the initial and final heaps are isomorphic. They have implemented a type checker for their system in OCaml, but the soundness of their type system is only proved on paper.

Fluet and Wang [2004] implement a copying collector in Cyclone, applying an approach similar to Wang and Appel [2001] based on region types. This collector is part of a Scheme interpreter, which is not entirely verified, apparently due to the runtime system of Cyclone itself. The bulk of this runtime is apparently an

implementation of `malloc`.

There have been a number of papers that have used Hoare-like logic with separation logic to verify implementations of `malloc` and `free` [such as Yu et al. 2004, Marti et al. 2006].

Finally, my heap interface is somewhat inspired by the rely-guarantee method for reasoning about concurrent programs [Jones 1983].

9.2 Future work

The current work can be extended in a number of directions. In fact, I have already begun to pursue some and have verified a Cheney copying collector implemented in the C-like language Cminor [Leroy 2006] that uses 32-bit machine arithmetic. This collector also supports an arbitrary number of object fields, as well as static typing, using object headers. I have also developed an improved program logic and verification infrastructure. However, I have not yet applied the ADT-based interface to this collector. This work is not described here as it is outside the scope of the current dissertation.

9.2.1 Improved machine model

The machine model can be made more realistic. The main change that would make the machine model more realistic would be the use of 32-bit modular machine arithmetic. This does not introduce any fundamental difficulty for my approach, but reasoning about modular arithmetic is more difficult, as Coq does not have any built-in tools for reasoning about it.

One fundamental aspect of my collectors that differs from practice is that they are implemented in assembly, instead of a higher level language such as C. Working

at a higher level will make programming the collector easier, but given my current techniques verifying the collector is much harder than programming the collector, so this is only a minor advantage of working at a higher level. A more important advantage to verifying collectors written at a higher level is that a collector could be verified once at a high level then compiled to a variety of targets. One disadvantage is that if a GC written in C is verified and then compiled with an unverified compiler, the resulting binary is unverified. It may be possible to solve this problem by using a fully verified collector such as CompCert [Leroy 2006]. The main obstacle to working at a higher level is that a new program logic will be needed. Fortunately, Appel and Blazy [2007] describe a program logic for Cminor, one of the intermediate languages compiled by the CompCert verified compiler, that uses separation logic. I could apply this work to collector verification.

9.2.2 More realistic collectors

There are a number of ways the collector implementations could be improved to better support mutator programs. These include support for static typing, objects and root sets of arbitrary size, and more efficient algorithms.

One aspect of my verified compilers that prevents them from being used with modern functional languages is that the object heap is *dynamically typed*: pointerness is on a *per-value* basis. In languages such as SML and Haskell are *statically typed*, so pointerness is on a *per-location* basis: the first field of a particular object, for instance, will always be a pointer, or will always not be a pointer. Static typing does not require reserving field bits for the GC, allowing unboxed floating point numbers.

One common way to track this kind of pointer information is to add a header to each object that indicates whether each field is a pointer or not. This change would require modifying both the verification of the collectors as well as the interface, as

one of the most fundamental aspects of a GC-mutator interface is specifying what values are object pointers.

Another limitation of my GCs is that all objects are pairs, which obviously is not very general. I would like to support an arbitrary number of fields per object. Again, I can add information about the length of each object to some kind of per-object header. This would require extending the interface as well as the specifications of each collector, but should not cause any fundamental difficulty. Adding support for an arbitrary number of fields might actually reduce the size of proofs, because right now there are many places where the reasoning about the first and second fields are each handled as a special case, often with a lot of cutting-and-pasting.

My current GC interface and implementations only allow a single value in the root set. This restriction makes implementing even simple mutators difficult. In a real implementation of a garbage collected language the mutator has many roots, often stored on a stack. To support garbage collection, the mutator must properly maintain metadata to allow the GC to examine the entire root set and, in the case of a precise collector, specify which stack values are object pointers. As the mutator must explicitly maintain these structures they cannot be abstracted away from the mutator. In fact, it would make more sense to hide the details of these mutator-maintained data structures from the collector. Vanderwaart and Crary [2003] have given a type system for reasoning about precise GC information, but do not address the GC's side of the interface.

Finally, I am not aware of production systems that actually use Cheney or Baker collectors, so I would like to apply my approach to verify different GC algorithms. Generational collectors [Appel 1989] are widely used for functional languages and require a write barrier. These should not be any more difficult than the Baker collector, which also requires a barrier. Concurrent collectors also would be a natural

next step. Verifying a concurrent collector would be challenging, as they are even more fine grained than an incremental collector. Verifying a concurrent collector would require a new program logic capable of reasoning about such programs and would likely require a more sophisticated GC-mutator interface.

9.2.3 Improved program reasoning

My large-scale verification efforts have provided me with a lot of practical experience with reasoning about programs. While I was able to verify collectors, there were a number of aspects of the effort that were more tedious than they needed to be. As I have mentioned (and shown in my example in Section 2.8), WeakSCAP produces a verification condition with a lot of redundancy. This redundancy does not present any theoretical difficulty, but results in a lot of unnecessary duplication of effort. By constructing a new improved program logic I could eliminate this.

Additionally, my tools for reasoning about separation logic could be improved. While I do not expect to be able to fully automate verification, adopting techniques from tools such as Smallfoot [Berdine et al. 2005] should help to eliminate a lot of tedium. Finally, while reviewing my proof of the Baker collector, I was stunned by how many lines were devoted to fairly basic reasoning about finite sets. This is an area where it should be possible to greatly improve automation without a lot of effort, perhaps by leveraging the new finite set library in the latest version of Coq.

9.2.4 Other uses of ADTs for system level interfaces

A more speculative direction for my work is to apply my approach to reasoning about mutator-collector interfaces to interfaces used in systems programming. There are many resources and abstractions in operating systems, from virtual memory to file

systems to hypervisors such as Xen [Barham et al. 2003]. Allowing user programs that leverage these services to be verified with respect to a high-level interface, while still allowing a foundational proof that encompasses the entire system, will result in more reliable systems. As with language runtime systems, if an operating system is buggy, no real guarantee can be made about anything that runs on top of it.

Bibliography

Andrew W. Appel. Tactics for separation logic. `http://www.cs.princeton.edu/~appel/papers/septacs.pdf`, January 2006.

Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989. ISSN 0038-0644.

Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proc. 30th ACM Symposium on Principles of Programming Languages*, pages 285–298, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-628-5.

Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978. ISSN 0001-0782.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proc. of the Nineteenth ACM Symposium on Operating Systems*

Principles, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, 1984. ISSN 0164-0925.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *4th Formal Methods for Components and Objects*, pages 115–137, 2005.

Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pages 220–231, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-729-X.

Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988. URL `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proc. of the 1984 ACM Symp. on LISP and Functional Prog.*, pages 256–262, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-142-3.

L. Burdy. B vs. Coq to prove a garbage collector. In R. J. Boulton and P. B. Jackson, editors, *14th Int'l Conference on Theorem Proving in Higher Order Logics: Supplemental Proc.*, pages 85–97, September 2001. Report EDI–INF–RR–0046, Division of Informatics, University of Edinburgh.

Cristiano Calcagno, Peter O'Hearn, and Richard Bornat. Program logic and equiva-

lence in the presence of garbage collection. *Theoretical Comp. Sci.*, 298(3):557–581, 2003. ISSN 0304-3975.

C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970. ISSN 0001-0782.

Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl.*, pages 54–65, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250742.

Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proc. 2000 ACM Conf. on Prog. Lang. Design and Impl.*, pages 95–107, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349315.

Coq Development Team. The Coq proof assistant reference manual. Coq release v8.0pl4, January 2007a.

Coq Development Team. The Coq proof assistant reference manual. Coq release v8.1pl3, December 2007b.

Karl Crary and Joseph Vanderwaart. An expressive, scalable type theory for certified code. Technical Report CMU-CS-01-113, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, May 2001.

Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978. ISSN 0001-0782.

Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 70–83, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.

Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular verif. of assembly code with stack-based control abstractions. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, June 2006.

Matthew Fluet and Daniel Wang. Implementation and performance evaluation of a safe runtime system in cyclone. In *Informal Proc. of the SPACE 2004 Workshop*, January 2004.

Georges Gonthier. Verifying the safety of a practical concurrent garbage collector. In R. Alur and T. Henzinger, editors, *Computer Aided Verification CAV'96*, Lecture Notes in Computer Science, New Brunswick, NJ, 1996. Springer-Verlag.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996. ISBN 0-13-370875-6.

David Gries. An exercise in proving parallel programs correct. *Communications of the ACM*, 20(12):921–930, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359897.359903.

David Gries. Corrigendum. *Communications of the ACM*, 21(12):1048, 1978.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

Klaus Havelund. Mechanical verification of a garbage collector. In *FMPPTA'99*, 1999. URL `http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund/Publications/gc-fmppta99.ps`.

Chris Hawblitzel, Heng Huang, Lea Wittie, and Juan Chen. A garbage-collecting typed assembly language. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 41–52, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: http://doi.acm.org/10.1145/1190315.1190323.

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language (2nd Edition)*. Addison-Wesley Professional, 2006. ISBN 0321334434.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, October 1969.

Paul Hudak. *Object and task reclamation in distributed applicative processing systems*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 1982.

Rob Hunter and Shriram Krishnamurthi. A model of garbage collection for oo languages. In *Tenth Int'l Workshop on Foundations of Object-Oriented Lang. (FOOL10)*, 2003.

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

C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. ISSN 0164-0925.

Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. ISBN 0–471–94148–4. URL `http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html`. With a chapter on Distributed Garbage Collection by R. Lins.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10 (3):269–303, 2000. ISSN 0956-7968.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL `http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf`.

Chunxiao Lin, Andrew McCreight, Zhong Shao, Yiyun Chen, and Yu Guo. Foundational typed assembly language with certified garbage collection. In *TASE '07: Proc. of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 326–338, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2856-2.

Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation

logic. In *9th JSSST Workshop on Programming and Programming Languages (PPL 2007)*, 2007.

Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *ICFEM 2006: Eighth International Conference on Formal Engineering Methods*, 2006.

Andrew McCreight. The mechanized verification of garbage collector implementations: Coq implementation. `http://flint.cs.yale.edu/flint/publications/mccreight-thesis.html`, May 2008.

Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl.*, pages 468–479, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

Stefan Monnier and Zhong Shao. Typed regions. Technical Report YALEU/DCS/TR-1242, Dept. of Comp. Sci., Yale University, New Haven, CT, October 2002.

Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proc. 2001 ACM Conf. on Prog. Lang. Design and Impl.*, pages 81–91, New York, 2001. ACM Press.

Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA '95: Proc. of the 7th Int'l Conference on Functional Prog.*

Lang. and Comp. Architecture, pages 66–77, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-719-7.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. ISSN 0164-0925.

Aleksander Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable ADTs in Hoare type theory. In *Proc. 2007 European Symposium on Programming*, 2007.

George Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. 1998 ACM Conf. on Prog. Lang. Design and Impl.*, pages 333–344, New York, 1998.

George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press. ISBN 0-89791-853-3.

Zhaozhong Ni and Zhong Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, January 2006.

Leonor Prensa Nieto and Javier Esparza. Verifying single and multi-mutator garbage collectors with Owicki-Gries in Isabelle/HOL. In *MFCS '00: Proc. of the 25th Int'l Symp. on Mathematical Foundations of Comp. Sci.*, pages 619–628, London, UK, 2000. Springer-Verlag. ISBN 3-540-67901-4.

Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proc. 31st ACM Symposium on Principles of Programming*

Languages, pages 268–280, New York, NY, USA, 2004. ACM. ISBN 1-58113-729-X.

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proc. 32nd ACM Symposium on Principles of Programming Languages*, pages 247–258, New York, NY, USA, 2005. ACM.

Lawrence C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-56543-X.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proc. of the 17th Annual IEEE Symp. on Logic in Comp. Sci.*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.

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

Carsten Schürmann. *Automating the meta theory of deductive systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2000. Chair-Frank Pfenning.

Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 217–232. ACM Press, January 2002.

Smallfoot Development Team. Smallfoot. `http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/`, December 2005.

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201700735.

Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann, 2006. ISBN 0120884216.

Noah Torp-Smith, Lars Birkedal, and John C. Reynolds. Local reasoning about a copying garbage collector. *ACM Transactions on Programming Languages and Systems*, 2006. To appear.

Joseph C. Vanderwaart and Karl Crary. A typed interface for garbage collection. In *Proc. 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 109–122, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-649-8.

Martin T. Vechev, Eran Yahav, and David F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. In *Proc. 2006 ACM Conf. on Prog. Lang. Design and Impl.*, pages 341–353, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4.

Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzky. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *Proc. 2007 ACM Conf. on Prog. Lang. Design and Impl.*, pages 456–467, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250787.

Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*. Springer Verlag, 1993.

Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proc. 28th ACM Symposium on Principles of Programming Languages*, pages 166–178, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-336-7.

Tjark Weber. Towards mechanized program verification with separation logic. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004. ISBN 3-540-23024-6.

Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2004. ISBN 3-540-23017-3.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

Hongseok Yang. An example of local reasoning in BI pointer logic: The Schorr-Waite graph marking algorithm. In *Proc. of the SPACE 2001 Workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, pages 41–68, 2001. URL `http://www.dcs.qmul.ac.uk/~hyang/paper/SchorrWaite.ps`.

Dachuan Yu, Nadeem A. Hamid, and Zhong Shao. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming*, 50(1-3): 101–127, March 2004.