On Garbage and Program Logic*

Cristiano Calcagno[1 2] and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] DISI, University of Genova

Abstract. Garbage collection relieves the programmer of the burden of managing dynamically allocated memory, by providing an automatic way to reclaim unneeded storage. This eliminates or lessens program errors that arise from attempts to access disposed memory, and generally leads to simpler programs. One might therefore expect that reasoning about programs in garbage collected languages would be much easier than in languages where the programmer has more explicit control over memory. But existing program logics are based on a low level view of storage that is sensitive to the presence or absence of unreachable cells, and Reynolds has pointed out that the Hoare triples derivable in these logics are even incompatible with garbage collection. We present a semantics of program logic assertions based on a view of the heap as finite, but extensible; this is for a logical language with primitives for dereferencing pointer expressions. The essential property of the semantics is that all propositions are invariant under operations of adding or removing garbage cells; in short, they are garbage insensitive. We use the assertion language to formulate notions of partial and total correctness for a small programming language, and provide logical characterizations of two natural notions of observational equivalence between programs.

1 Introduction

Garbage collection is an essential method used to reclaim heap-allocated objects whose lifetime cannot be easily predicted at compile time. It is most strongly associated with high-level languages such as Lisp, ML and Java, where heap allocation is the norm. But it can also be used in a lower level language like C, coexisting with explicit deallocation primitives [8]. In any case, garbage collection relieves the programmer of the burden of explicitly managing dynamically allocated memory. This generally leads to simpler programs, and removes or lessens errors that result from incorrect attempts to access disposed memory, errors that are often difficult to diagnose or even reproduce.

Given the intuitive rationale for garbage collection, one would expect that reasoning about garbage collected languages (especially memory safe languages) would be much easier than for lower level languages. But existing program logics take a view of storage that is sensitive to the presence of unreachable cells, a view that is not invariant under garbage collection.

* Work partially supported by the EPSRC

1.1 A Logical Conundrum

The following problem was raised by Reynolds in [11].

Consider a statement form $x := \mathtt{cons}(E, E')$ that allocates and initializes a new cons cell and places a pointer to that cell in storage variable x. Then the following sequence of instructions creates a new cell, and then makes it garbage.

$$x := \mathtt{cons}(3, 4);\ x := z.$$

Now, ask the question: is there a pointer y to a cell in the heap where $y.1 = 3$, after these statements have been executed? From the point of view of execution the answer is that it depends, on whether a garbage collector has taken control or not. (We use $x.1$ and $x.2$ for the combination of dereferencing a pointer and accessing one of the two components of a cons cell.)

The conundrum is that program logic seems to take a particular stance, one that is incompatible with garbage collection. That is, previous logics for pointer programs would allow us to derive a Hoare triple

$$\{true\}\, x := \mathtt{cons}(3, 4);\ x := z\, \{\exists y.\, y.1 = 3\}.$$

The problem is that on termination there might not actually be a cell whose car is 3, if a garbage collector reclaims the detached cell.

It is worth looking at this example in more detail. After the statement $x := \mathtt{cons}(3, 4)$ has been executed, it has to be admitted that there is such a cell, because it cannot be garbage collected. So we would expect to have a true Hoare triple $\{true\}\, x := \mathtt{cons}(3, 4)\, \{\exists y.\, y.1 = 3\}$. But then either Hoare's of Floyd's assignment axiom gives us $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$ because x is free in neither the precondition nor the postcondition. And now the die is cast: we obtain the triple above by sequencing.

The conundrum related to the problem of *full abstraction*. That is, there are Hoare triple contexts that can distinguish programs that are observationally equivalent, if we take a standard notion of observation (such as termination). For example, one would expect the program fragment given above to be observationally equivalent to $x := z$ on its own, but if the standard assignment axiom is to be believed then the weakest precondition of $\exists y.\, y.1 = 3$ is just itself, so we only get $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$. Thus, if there are states where this precondition is false, then the logical context $\{true\} - \{\exists y.\, y.1 = 3\}$ will distinguish two "equivalent" commands, since it will not be satisfied by $x := z$. (In this discussion we have ignored the possibility of running out of memory. But a similar point can be made if we bound the number of possible pointers by strengthening the precondition to say that there is at least one cell available, and by using $x := \mathtt{cons}(7, 8);\ x := z$ in place of the single statement $x := z$.)

Of course, the foregoing hinges on several points: the substitution-oriented treatment of assignment and a typical reading of the logical operators. And there are several ways that one might react to the conundrum; we discuss several of the possibilities in the conclusion. The purpose of this paper is to provide one solution, in the form of a semantics of pointer assertions that is insensitive to garbage.

1.2 A Resolution

The semantics we give is based on a view of memory as finite but extensible. What we mean by this is that at any given time the heap consists of a finite collection of cons cells in storage, but the quantifiers are arranged so as to permit arbitrary (but finite) extensions to the heap. This has echoes of the idea of "finite but potentially infinite", but the semantics is also compatible with overflow-sensitive situations where a bound is placed.

We use a possible-world semantics, where the current heap (a finite collection of cells) is the world. The most important case is the interpretation of $\exists x. P$. As usual, x here ranges over values, including pointers and even pointers not in the current heap. If such a new pointer is chosen, then the world is extended as well, to provide a binding for (at least) the new pointer. The other connectives are interpreted pointwise, without changing worlds.

After presenting the model we prove three main results. The first is garbage insensitivity: the semantics of assertions is invariant under the operations of removing or adding garbage cells.

The second and third are both full abstraction results, which establish that two programs satisfy the same Hoare triples just when they are observationally equivalent. One of these results connects total correctness specifications with an equivalence obtained from observing termination, and the other connects partial correctness specifications with an equivalence obtained from observing the presence of runtime errors (such as dereferencing *nil*) as well as termination. The first equivalence, for total correctness, conflates divergence and runtime error. Both equivalences equate the program fragments discussed in the conundrum above.

We also establish a connection between the total states model and a partial states model based on the celebrated "dense topology" semantics of classical logic [3, 9]. This connection provides some theoretical justification for the unusual semantics of quantification in the total states model.

We work with a pared down storage model and programming language for this study, where the heap consists of a collection of binary cons cells, and where there are data pointers but no code pointers or closures. Garbage insensitivity appears to extend to higher-order store, but the extension of full abstraction is not obvious. Even though it is simple, this kind of storage model has interesting structure that has only recently begun to be uncovered [11, 7].

In this paper we concentrate on garbage and the relation between specifications and observational equivalence; we do not consider program logic axioms for the individual statements of our programming language. In a future paper we will show how the approach of [11, 7] can be adapted, to provide a logic that is sound for an operational semantics with rules for garbage collection.

2 The Storage Model

The storage model we use supports pointer allocation, dereferencing and assignment, and divides the state into *stack* and *heap* components. The stack consists of associations of values to variables, and is altered by the standard assignment statement $x := E$. As is common in Hoare logic, we do not distinguish between a variable name and the l-value it denotes; this is justified because we are not considering aliasing between stack variables. The heap consists of a collection of binary cons cells, which can only be accessed via pointers; the extension to records of other sizes is straightforward.

The basic domains of the model are as follows.

$$\begin{aligned}
\texttt{Pointers} &\triangleq \{p, q, ...\} \quad \texttt{Bool} \triangleq \{\mathit{false}, \mathit{true}\} \quad \texttt{Tags} \triangleq \{a, b, ...\} \\
\texttt{Variables} &\triangleq \{x, y, ...\} \quad \texttt{Nat} \triangleq \{0, 1, ...\} \\
\texttt{Values} &\triangleq \texttt{Nat} + \texttt{Bool} + \texttt{Pointers} + \{\mathit{nil}\} \\
\texttt{Stacks} &\triangleq \texttt{Variables} \rightharpoonup_{fin} \texttt{Values} \\
\texttt{Heaps} &\triangleq \texttt{Pointers} \rightharpoonup_{fin} \texttt{Values} \times \texttt{Values} \\
\texttt{States} &\triangleq \texttt{Stacks} \times \texttt{Heaps}
\end{aligned}$$

Although there is no ordering on variables, in the programming language to be presented later allocation and deallocation of variables will obey a stack discipline. Stacks and heaps are represented by finite partial functions: we write $\mathit{dom}(s)$ or $\mathit{dom}(h)$ for the domain of definition of a stack or heap.

Notice that a state $s, h \in \texttt{Stacks} \times \texttt{Heaps}$ might have dangling pointers, pointers that are reachable but not defined in h. Since we will consider a programming language without memory disposal, total states play a central role. We use the following definitions.

- Pointer q is reachable from p in h if $q = p$ or $h(p) = \langle v_1, v_2 \rangle$ and q is reachable from v_1 or v_2 in h;
- p is reachable in s, h if there is $x \in \mathit{dom}(s)$ such that p is reachable from $s(x)$ in h;
- (s, h) is total if every reachable pointer is in the domain of h.

We will also be concerned with the presence or absence of garbage.

- (s, h) is garbage free if every $p \in \mathit{dom}(h)$ is reachable in s, h.

Garbage collection is intimately connected to the relation of heap extension, which gives us a partial order on `Heaps`.

- $g \sqsubseteq h$ indicates that the graph of heap g is a subset of the graph of heap h.

From the point of view of this model, given a state s, h the effect of a garbage collector is to shrink h by selecting $g \sqsubseteq h$ in a way that doesn't produce any (new) dangling pointers. In the best case, the collector will remove all garbage.

- $\mathit{prune}(s, h) = (s, g)$, where $g \sqsubseteq h$ is the subheap of h restricted to those pointers reachable in s, h.

Actually, a relocating collector can move heap cells around. For this the notion of isomorphism is important.

- $=_\alpha$ is equality of states modulo renaming of pointers.
  (To be explicit, if $f : \texttt{Pointers} \to \texttt{Pointers}$ is a permutation, let f^* be the induced function from values to values which is the identity on non-pointer values. Then $s, h =_\alpha s', h'$, where $s' = s; f^*$ and $h' = f^{-1}; h; (f^* \times f^*)$.)

Lemma 1. *The following hold:*

- *$prune(s, h)$ is garbage free.*
- *If (s, h) is total then so is $prune(s, h)$.*
- *The relation $=_\alpha$ preserves totality and garbage freedom: If $(s, h) =_\alpha (s', h')$ and s, h is total (garbage free) then (s', h') is total (garbage free).*

3 Expressions

We restrict our attention to expressions that are free from side effects. A programming language for transforming states will be given later in Section 6, after we have presented the assertion language.

We include standard operations of arithmetic, along with generic equality testing and pointer dereference. The syntax of expressions is given by the following grammar:

$$\begin{array}{rl} E ::= & x \\ | & 0 \mid 1 \mid E + E \mid E \times E \mid E - E \\ | & \texttt{true} \mid \texttt{not}\, E \mid E\, \texttt{and}\, E \mid E == E \\ | & \texttt{nil} \\ | & E.1 \mid E.2 \end{array}$$

To keep things simple we have avoided all issues of typing. A type system could eliminate type errors in many cases but not, without great complication, for pointer dereferencing, where `nil` or some other value is typically used as a special “pointer” that should not be dereferenced.

So, the semantics of an expression E will determine an element

$$[\![E]\!]s, h \in \texttt{Values} + \{wrong\}$$

where s, h is a total state and the domain of s includes the free variables of E. A *wrong* result indicates a type error; this is treated essentially as in a partial function semantics, where *wrong* stands for undefined.

Selected semantics definitions are as follows.

$$
\begin{array}{rll}
[\![x]\!]s,h \triangleq & s(x) & \\
[\![E_1 == E_2]\!]s,h \triangleq & \mathit{wrong} & \text{if } [\![E_1]\!]s,h = \mathit{wrong} \text{ or } [\![E_2]\!]s,h = \mathit{wrong} \\
& \mathit{true} & \text{if } \mathit{wrong} \neq [\![E_1]\!]s,h = [\![E_2]\!]s,h \\
& \mathit{false} & \text{if } \mathit{wrong} \neq [\![E_1]\!]s,h \neq [\![E_2]\!]s,h \neq \mathit{wrong} \\
[\![\mathtt{not}\ E]\!]s,h \triangleq & \mathit{wrong} & \text{if } [\![E]\!]s,h = \mathit{wrong} \\
& \mathit{false} & \text{if } [\![E]\!]s,h = \mathit{true} \\
& \mathit{true} & \text{if } [\![E]\!]s,h = \mathit{false} \\
[\![E.i]\!]s,h \triangleq & \mathit{wrong} & \text{if } [\![E]\!]s,h \notin \mathit{dom}(h) \\
& \pi_i(h(p)) & \text{if } [\![E]\!]s,h = p \in \mathit{dom}(h)
\end{array}
$$

It is not difficult to verify that the semantics of expressions is insensitive to garbage.

Lemma 2. $[\![E]\!]s,h \;=\; [\![E]\!]\mathit{prune}(s,h)$.

Such a result is much more difficult for assertions than expressions, because of quantifiers which may quantify over unreachable elements.

4 Propositions

The grammar for propositions P is as follows.

$$P ::= E = E \mid P \to P \mid \mathtt{false} \mid \exists x.\, P$$

We can define various other connectives as usual: $\neg P \triangleq P \to \mathtt{false}$; $\mathtt{true} \triangleq \neg(\mathtt{false})$; $P \vee Q \triangleq (\neg P) \to Q$; $P \wedge Q \triangleq \neg(\neg P \wedge \neg Q)$; $\forall x.\, P \triangleq \neg\exists x.\neg P$.

The assertion language looks rather sparse, but using the expressions a number of properties can be defined. For example, $E = E + 0$ says that E is a number, and $E.1 = E.1$ says that E is a pointer. In practice, one would also want atomic predicates for describing reachability properties (e.g. [1]), or a definitional mechanism for defining them. Any such predicates could be included, as long as they satisfy the properties Growth, Shrinkage and Renaming described in the next section.

The semantics is described in terms of a judgement of the form

$$s, h \models P$$

which asserts that P holds of total state s, h. We require $\mathit{free}(P) \subseteq \mathit{dom}(s)$, where $\mathit{free}(P)$ indicates the set of variables occurring free in P.

The quantifier-free fragment is straightforward.

$$
\begin{array}{ll}
s,h \models E = E' \overset{\Delta}{\iff} & [\![E]\!]s,h = [\![E']\!]s,h = v \in \mathtt{Values} \\
s,h \models \mathtt{false} & \text{never} \\
s,h \models P \to Q \overset{\Delta}{\iff} & \text{if } s,h \models P \;\text{ then }\; s,h \models Q
\end{array}
$$

The only point to note is the interpretation of equality, which requires that neither side is wrong.

Normally, one would expect the following interpretation of $\exists$.

EXISTENTIAL: FIRST TRY

$$s, h \models \exists x.P \overset{\Delta}{\iff} \exists v \in \texttt{Values}.(s \mid x \mapsto v), h \models P$$

($(s \mid x \mapsto v)$ is the stack like s except that x maps to v.) The immediate problem is that this is not even well defined: there is no guarantee that $(s \mid x \mapsto v), h$ be a total state, so the right-hand side is not well formed. We might attempt to allow partial states and maintain this interpretation, but then we run into the conundrum mentioned in the Introduction.

Our solution is to accompany the selection of a value with a change of world. That is, if v is a pointer whose value is not determined in the current heap, then we look for a bigger heap where the value is determined. And in doing so, we must also ensure that no dangling pointers are introduced by following v further into the heap. Thus, we look for an extended heap, which maintains total state status.

EXISTENTIAL: OFFICIAL VERSION

$$\begin{aligned} s, h \models \exists x.P \overset{\Delta}{\iff} & \exists v \in \texttt{Values}.\, \exists g \sqsupseteq h. \\ & (s \mid x \mapsto v), g \text{ is total and } (s \mid x \mapsto v), g \models P \end{aligned}$$

Another idea that occurs is to quantify over reachable pointers only; we discuss in the conclusion.

It is worth spelling out the induced semantics of $\forall$, that obtains from the $\neg\exists\neg$ encoding:

$$\begin{aligned} s, h \models \forall x.P \overset{\Delta}{\iff} & \forall v \in \texttt{Values}.\, \forall g \sqsupseteq h. \text{ if } (s \mid x \mapsto v), g \text{ is total} \\ & \text{then } (s \mid x \mapsto v), g \models P \end{aligned}$$

The semantics handles the example from Section 1.1 as follows. Given any heap, the semantics of $\exists$ allows a new pointer to be selected, along with a heap extension (change of possible world) which has 3 in the first component; $\exists y.\, y.1 = 3$ is always true. So, both of the Hoare triple contexts in Section 1.1 boil down to $\{true\}$–$\{true\}$. And, because of the full abstraction results in Sections 7 and 8, we know that the semantics is not *too* abstract: it makes *enough* distinctions to distinguish genuinely inequivalent commands.

One property of the model is worth noting, the truth of

$$\forall y.\, \exists x.\, x.1 = y$$

This says that for any value we consider, there is some cell we can find which has y in its first component. And the same goes for the second component. This illustrates the extensible heap view taken by the semantics, where there is an extensible stock of pointers, or locations, with arbitrary possible contents, to choose from. (This formula is true in all states only if `Pointers` is infinite. With a finite collection of pointers, $\forall y$ could saturate the set of available pointers leaving none for $\exists x. x.1 = y$.)

Many specifications one uses when reasoning about pointer programs are garbage insensitive. For example: x points to a non-circular linked list; or x points to a linked list representing the sequence L. With the aid of recursive definitions (which we have avoided for theoretical simplicity in this paper) these properties can be expressed in our language, as in $nclist(x) \;\Leftrightarrow\; x = \mathtt{nil} \;\vee\; (\exists z. x.2 = z \,\wedge\, nclist(z))$.

Examples of properties that are not garbage insensitive include some that are invariant under pointer renaming (relocation)and some that are not. An example of the former is "the heap has exactly n cells", while an example of the latter is "if we add 2 to pointer p, we get pointer q". Garbage sensitive properties are useful in lower level languages.

5 Garbage Insensitivity

There are two aspects to garbage insensitivity, which we label growth and shrinkage.

Growth. If $s, g \models P$, $g \sqsubseteq h$, and s, h is total then $s, h \models P$.

Shrinkage. If $s, h \models P$, $g \sqsubseteq h$, and s, g is total then $s, g \models P$.

Growth says just that if a proposition is true then it remains true if we add extra cells to the heap. *Shrinkage* says the opposite: truth is preserved by removing cells from the heap. In both cases there is the proviso that the cells added or removed do not result in dangling pointers, as our semantics is defined only for total states.

Growth is essentially Kripke's monotonicity property for intuitionistic logic, but for one difference: the side condition on totality is a property that refers to both the stack and heap components. Conventional Kripke monotonicity would consider all larger heaps (possible worlds), without any side conditions that refer to stacks/environments.

Shrinkage looks like a backwards form of Kripke monotonicity, but is not. Because of the restriction to total states, we will only be able to go back as far as the heap obtained by pruning, which is dependent on s. (In the dense semantics presented later, which allows partial states, forwards Kripke monotonicity does hold, while backwards does not.)

Theorem 1 (Garbage Insensitivity Theorem). *All propositions satisfy Growth and Shrinkage. As a consequence, each proposition is completely determined by its meaning at garbage-free states:*

$s, h \models P$ *iff* $prune(s, h) \models P$.

The proof of the theorem relies on the

Lemma 3 (Renaming Lemma). *All propositions are invariant under pointer renaming. If* $s, h \models P$ *and* $s, h =_\alpha s', h'$ *then* $s', h' \models P$.

If we combine this with Garbage Insensitivity then we obtain a high level correspondent to the idea that propositions are invariant under the operation of a relocating garbage collector. The Renaming Lemma is true because we have provided no facilities for pointer arithmetic among the expressions we consider.

The theorem holds for any collection of atomic predicates that are closed under Growth, Shrinkage and Renaming.

Proof: [of Garbage Insensitivity] By structural induction on P.

The case of `false` is trivial. For $P \to Q$ we consider *Shrinkage*. Suppose $s, h \models P \to Q$ and consider $g \sqsubseteq h$ where s, g is total. Either $s, h \not\models P$ or $s, h \models Q$ must hold. If the former, we use the *Growth* part of the induction hypothesis for P to conclude $s, g \not\models P$, and thus $s, g \models P \to Q$. If the latter we use the *Shrinkage* induction hypothesis to conclude $s, g \models Q$, and thus $s, g \models P \to Q$. The proof of *Growth* for $P \to Q$ is symmetric.

For the *Growth* case of $\exists x. P$, if $s, g \models \exists x. P$ then there is v and $k \sqsupseteq g$ with $(s \mid x \mapsto v), k \models P$. Consider a permutation that renames the pointers in the domain of k but not g to be different from all pointers in h, while leaving pointers in g fixed. Let k' be the heap obtained from k via this permutation. Then $(s \mid x \mapsto v'), k' \models P$ by the Renaming Lemma, where v has been perhaps renamed to v' (if v is in the domain of k but not g). Let j denote the lub of h and k'. j exists: it is just the union of h and k', which have g in common but which are otherwise defined on distinct pointers. It is not difficult to see that $(s \mid x \mapsto v'), k'$ is total. (Since the permutation fixes g, no renaming is required in s.) Since $j \sqsupseteq k'$, we can apply the *Growth* induction hypothesis, to obtain $(s \mid x \mapsto v'), j \models P$. Since $j \sqsupseteq h$ we satisfied the right-hand-side of the definition of $s, h \models \exists x. P$.

The *Shrinkage* case of $\exists x. P$ follows immediately from the clause for $\exists$ and the transitivity of $\sqsubseteq$.

The case of the equality predicates is immediate from Lemma 2. ■

Interestingly, the validity of the usual law of $\exists$ elimination

$$\frac{Q \wedge P \models R \quad Q \models \exists x. P}{Q \models R} \ (x \text{ not free in } Q \text{ or } R)$$

relies on Garbage Insensitivity. (Here, $\models$ refers to the semantic consequence relation between propositions, where $P \models Q$ if any state satisfying P also satisfies Q.) The proof of validity of this law requires a variable weakening lemma, which involves a change of world as well as an additional variable.

Lemma 4 (Variable Weakening Lemma). *If* $s, h \models P$, $g \sqsupseteq h$, x *is not free in* P, *and* $(s \mid x \mapsto v), g$ *is total, then* $(s \mid x \mapsto v), g \models P$.

The proof of this lemma uses only the *Growth* part of Garbage Insensitivity, but *Growth* itself relies on *Shrinkage* for propositions of the form $P \to Q$

The following version of $\exists$ introduction also holds:

$$\frac{Q \models P[E/x] \quad Q \models E = E}{Q \models \exists x.\, P}$$

Recall that our interpretation of equality is like in a partial function logic, where $E = F$ can hold only when neither side is *wrong*. Thus, the premise $E = E$ is not trivially true: it ensures that E denotes a genuine value.

6 A Programming Language

We define a small programming language for altering stacks and heaps.

$$\begin{aligned} C ::= {} & x := E \mid E.i := E \mid x := \texttt{cons}(E_1, E_2) \\ \mid {} & C; C \mid \texttt{while}\ E\ \texttt{do}\ C\ \texttt{od} \mid \texttt{local}\ x\ \texttt{in}\ C\ \texttt{end} \end{aligned}$$

The binding form `local` x `in` C `end` declares a new stack variable, which is de-allocated on block exit. This is why we refer to the s component of the semantics as the stack.

The commands are interpreted using a relation $\rightsquigarrow$ on configurations. Configurations include terminal configurations s, h as well as triples C, s, h. Also, in order to treat `local` we introduce a new command `dealloc`(x); this command form is used only in the operational semantics, and is not part of the language. In other sections metavariable C will refer exclusively to the unextended language, while here it includes `dealloc`.

The $\rightsquigarrow$ relation is specified by the following rules.

$$\frac{[\![E]\!]s, h = v}{x := E, s, h \rightsquigarrow (s \mid x \mapsto v), h} \qquad \frac{[\![F]\!]s, h = v \quad [\![E]\!]s, h = p \in dom(h)}{E.i := F, s, h \rightsquigarrow s, (h \mid p.i \mapsto v)}$$

$$\frac{p \notin dom(h) \quad p \in \texttt{Pointers} \quad [\![E_1]\!]s, h = v_1 \in \texttt{Values} \quad [\![E_2]\!]s, h = v_2 \in \texttt{Values}}{x := \texttt{cons}(E_1, E_2), s, h \rightsquigarrow (s \mid x \mapsto p), (h \mid p \mapsto \langle v_1, v_2 \rangle)}$$

$$\frac{C_1, s, h \rightsquigarrow C_1', s', h'}{(C_1; C_2), s, h \rightsquigarrow (C_1'; C_2), s', h'} \qquad \frac{C_1, s, h \rightsquigarrow s', h'}{(C_1; C_2), s, h \rightsquigarrow C_2, s', h'}$$

$$\frac{[\![E]\!]s, h = false}{\texttt{while}\ E\ \texttt{do}\ C\ \texttt{od}, s, h \rightsquigarrow s, h}$$

$$\frac{[\![E]\!]s, h = true \quad (C; \texttt{while}\ E\ \texttt{do}\ C\ \texttt{od}), s, h \rightsquigarrow K}{\texttt{while}\ E\ \texttt{do}\ C\ \texttt{od}, s, h \rightsquigarrow K}$$

$$\frac{}{\texttt{local}\ x\ \texttt{in}\ C\ \texttt{end}, s, h \rightsquigarrow C; \texttt{dealloc}(x), (s \mid x \mapsto nil), h}$$

$$\frac{}{\texttt{dealloc}(x), s, h \rightsquigarrow s - x, h}$$

In the last rule $s - x$ is the stack like s but undefined on x. $(h \mid p.i \mapsto v)$ is the heap like h except that the i'th component of the $h(p)$ cell is v.

We say that

- "C, s, h is stuck" in case there is no configuration K such that $C, s, h \rightsquigarrow K$;
- "C, s, h goes wrong" when there exists a configuration K such that $C, s, h \rightsquigarrow^* K$ and K is stuck;
- "C, s, h terminates normally" just if there is s, h such that $C, s, h \rightsquigarrow^* s', h'$.

It is possible to prove a number of properties about this operational semantics, such as that normal termination, going wrong and divergence are mutually exclusive, invariance under pointer renaming, and a kind of determinacy, where the prunes of any two output states gotten from the same initial state must be isomorphic.

One can also entertain an extension of the operational semantics with rules for garbage collection or even relocation; e.g. $s, h \rightsquigarrow s, g$ when $g \sqsubseteq h$ and s, g is total. Such extensions do not affect the results that follow.

7 Partial Correctness

If P and Q are propositions and C is a command then we have the specification form $\{P\}\, C\, \{Q\}$.

> PARTIAL CORRECTNESS
> We say that $\{P\}\, C\, \{Q\}$ is true just if for all s, h, if $s, h \models P$ then
> - C, s, h doesn't go wrong, and
> - if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models Q$.

We have arranged the definition of partial correctness so that well specified programs don't go wrong. That is, if $\{P\}C\{Q\}$ holds then executing C in a state satisfying P never leads to a runtime error.

Using partial correctness assertions it is possible to distinguish between programs that differ in when they go wrong, but that agree on all outputs states when they are reached. As an extreme example, consider a command *diverge* that always diverges without getting stuck (e.g. `while true do` $x := x$ `od`) and another command $x :=$ `nil`; $x.1 := 17$ that always gets stuck. The former satisfies the triple $\{true\} - \{true\}$ while the latter does not. This indicates that the notion of observational equivalence appropriate to partial correctness must distinguish divergence and stuckness.

These considerations lead to a notion $\cong_{pc}$ of observational equivalence based on observing both normal termination and stuckness. In formulating observational equivalence, we will only consider the execution of closed commands, ones where all variables have been bound by `local`.

> We define $C \cong_{pc} C'$ to hold just if
> for all closing contexts $G[\cdot]$,
> - $G[C], (), ()$ goes wrong just if $G[C'], (), ()$ does, and
> - $G[C], (), ()$ terminates normally just if $G[C'], (), ()$ does.

Theorem 2 (Full Abstraction, Partial Correctness). *$C \cong_{pc} C'$ iff C and C' satisfy the same partial correctness assertions.*

The proof goes by first relating $\cong_{pc}$ to a function $[\![C]\!]^{\sim}_{pc}$ determined by a command, which ignores garbage and renaming of locations, and then relating this relation to partial correctness. First, we define

$$[\![C]\!]_{pc} \subseteq \mathtt{TStates} \times \mathtt{TStates}_*$$

where TStates is the set of total states, and $\mathtt{TStates}_* \stackrel{\Delta}{=} \mathtt{TStates} \cup \{wrong\}$.

$$((s,h),(s',h')) \in [\![C]\!]_{pc} \stackrel{\Delta}{\Longleftrightarrow} C,s,h \rightsquigarrow^* s',h'$$
$$((s,h),wrong) \in [\![C]\!]_{pc} \stackrel{\Delta}{\Longleftrightarrow} C,s,h \text{ goes wrong}$$

The equivalence relation $\sim \subseteq \mathtt{TStates} \times \mathtt{TStates}$ is defined as

$$(s,h) \sim (s',h') \quad \stackrel{\Delta}{\Longleftrightarrow} \quad prune(s,h) =_\alpha prune(s',h')$$

where $=_\alpha$ is equality modulo renaming of locations. The theorem is then established by proving three lemmas.

Lemma 5. $[\![C]\!]_{pc}$ *induces a partial function between quotients*

$$[\![C]\!]^{\sim}_{pc} : (\mathtt{TStates}/\sim) \rightharpoonup (\mathtt{TStates}_*/\sim_*)$$

where $-/-$ is the quotient operation and $\sim_*$ extends $\sim$ with $(wrong, wrong)$.

Lemma 6. $C \cong_{pc} C' \Leftrightarrow [\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc}$.

Lemma 7. $[\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc} \Leftrightarrow C$ *and* C' *satisfy the same partial correctness assertions.*

The proofs of lemmas 6 and 7 use as main ingredient the fact that for each equivalence class c, there exists a command that creates an instance of c and an expression/proposition that characterizes c.

It is also possible to prove a version of the theorem which characterizes an observational approximation relation, rather than equivalence.

8 Total Correctness

If P and Q are assertions and C is a command then we have the specification form $[P]\,C\,[Q]$.

TOTAL CORRECTNESS We say that $[P]\,C\,[Q]$ is true just if
- $\{P\}\,C\,\{Q\}$ is true, and
- for all s,h, if $s,h \models P$ then C,s,h terminates normally.

Using total correctness assertions we can no longer distinguish between divergence and going wrong in the way that we did with partial correctness. That is, neither divergence nor an always sticking command satisfies $[true]-[true]$, because neither terminates normally. This suggests to define a notion of observational equivalence that conflates divergence and going wrong. Since, in our language, there is no way to recover from, or handle, a runtime error, we can do this by observing termination only (ignoring wrongness).

We define $C \cong_{tc} C'$ to hold just if
for all closing contexts $G[\cdot]$,
– $G[C]()()$ terminates normally just if $G[C']()()$ does.

Theorem 3 (Full Abstraction, Total Correctness). *$C \cong_{tc} C'$ iff C and C' satisfy the same total correctness assertions.*

The proof idea is similar to the one for partial correctness, except that we use a function on quotients that conflates non-termination and getting stuck.

9 Partial States and the Dense Semantics

We now give an alternate presentation of the model using partial states, which leads to a connection with the dense semantics of classical logic [3, 9]. We begin by defining a notion of support, which works for partial states. Intuitively, $s, h \Vdash P$ holds if h contains enough information to conclude P. Technically, this means that, for any total state we wish to extend h to, P will hold. Suppose s, h is a state, perhaps partial. Then

$$s, h \Vdash P \quad \overset{\Delta}{\iff} \quad \forall g \sqsupseteq h.\, \text{if } s, g \text{ is total then } s, g \models P$$

Support has the following properties.

Totality Condition: $s, h \Vdash P \iff \forall g \sqsupseteq h.\, \text{if } s, g \text{ is total then } s, g \Vdash P$.

Shrinkage Condition: If $s, h \Vdash P$, $g \sqsubseteq h$, and the domain of g contains all those pointers reachable in s, h, then $s, g \Vdash P$.

Density Condition: $s, h \Vdash P \iff \forall h' \sqsupseteq h.\, \exists h'' \sqsupseteq h'.\, s, h'' \Vdash P$.

These conditions are not independent. In particular, Density follows from Monotonicity and Shrinkage. Totality and Shrinkage are of interest from the perspective of pointers, while Density is more of a logical property.

Theorem 4. *All propositions satisfy Totality, Shrinkage and Density. Furthermore, the standard clauses of dense semantics all hold.*

$$\begin{array}{lll} s, h \Vdash \texttt{false} & & \text{never} \\ s, h \Vdash P \to Q & \iff & \forall h' \sqsupseteq h\,.\, \text{if } s, h' \Vdash P \;\; \textit{then} \;\; s, h' \Vdash Q \\ s, h \Vdash \exists x.P & \iff & \forall h' \sqsupseteq h,\, \exists h'' \sqsupseteq h'.\, \exists v \in \texttt{Values}.\, (s \mid x \mapsto v), h'' \Vdash P \end{array}$$

As a result, the forcing relation $\Vdash$ satisfies all the laws of classical logic (with the appropriate side-condition on $\exists$-intro to account for definedness).

The conditions in this proposition can be taken as an alternate definition of $\Vdash$, one that does not appeal to $\models$. It is standard that the semantic clauses above validate all of classical logic, as long as all atomic propositions satisfy density [3, 9]. The reader may enjoy verifying $((P \to \texttt{false}) \to \texttt{false}) \to P$.

Thus, because $\Vdash$ and $\models$ agree on total states, the total state semantics can be viewed as a specialization of a well known semantics. Beyond the justification it provides for the total semantics, the partial state perspective reveals further territory worth exploring. For example, there is a wider collection of properties that are garbage insensitive but neither necessarily dense nor total (such properties can be expressed using an $\exists$-free fragment of intuitionistic logic). We started with the total states model in this paper because the conceptual justification for the use of partial states in a garbage collected language is less than immediate. We just stress that totality is not theoretically necessary for garbage insensitivity (and we defer further discussion of this point to a future occasion).

10 Conclusion

As far as we are aware, no previous semantics of pointer assertions is garbage insensitive, including all of the pointer logic references in [4, 2, 7]. We take a few representative examples. In an early work Oppen and Cook described a complete proof system for deriving true Hoare triples about pointer programs [10], but their interpretation of quantifiers is the usual first-order interpretation; as a result their propositions are not garbage insensitive, and the Hoare triple context $\{true\}-\{\exists y.\, y.1 = 3\}$ behaves exactly as described in Section 1.1. In a series of papers, the most recent of which is [5], de Boer has advocated an approach where the quantifiers are restricted to range over currently active cells only. However, the currently active cells can contain garbage, and because of this de Boer's approach falls foul of the conundrum, and fails to characterize observational equivalence, using essentially the same examples we used in the Section 1.1. Finally, Honsell, Smith, Mason and Talcott [6] have given a characterization of equivalence (the "ciu theorem") in an expressive language with higher-order store. However, after giving this characterization a notion of "contextual assertion" is introduced, and an example is given showing a logical context that breaks observational equivalence. The sticking point in all of these approaches is the interpretation of quantifiers. If one considers a quantifier free language then garbage insensitivity is easy to achieve, at the cost of some expressivity. The decidable logic of [1] very nearly qualifies, except for the use of a garbage sensitive atomic predicate hs for describing sharing constraints.

We have given one resolution of the conundrum involving program logic and garbage collection, but our answer is not the only possible reaction. We conclude by discussing several of the others.

One reaction is to lay the blame not on the interpretation of propositions, but on the Floyd-Hoare approach to assignment. This reaction is difficult to uphold. For, although the soundness of the assignment axioms requires certain assumptions (such as no clashes between named variables), none of these are the essential problem here.

Another approach that is often suggested is to restrict quantifiers so that they range over reachable elements only. Although tantalizing, this notion has several obstacles to overcome. It invalidates Hoare's assignment axiom, as we would expect $\{\exists y.\, y.1 = 3\}\, x := z\, \{\exists y.\, y.1 = 3\}$ to fail if $\exists$ ranges over reachable elements only, since $x := z$ can detach a cell whose first component is 3. Just as significantly, it invalidates the rule of Weakening of contexts in first-order logic. This rule says that if P holds in a context with a collection X of variables, then we know that P holds for all bigger collections. These points do not erect a comprehensive roadblock to the "reachable elements" approach, but they do indicate that such a suggestion requires careful analysis and development.

There is a final reaction to the logical conundrum, and that is not to worry about a higher level of abstraction, on the basis we naturally do consider lower level aspects of program execution, even in garbage collected languages. Although this stance has some intuitive merit, it still has to deal with the logical problem. That is, if we admit that a garbage collector *might* intervene, and we adopt a concrete semantics of $\exists$, then we will have to deny

$$\{true\}\, x := \mathtt{cons}(3,4);\ x := z\, \{\exists y.\, y.1 = 3\}.$$

The question then would be what alterations to program logic axioms (for assignment and memory allocation) could give a useful logic, while maintaining this denial.

References

[1] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *ESOP '99: European Symposium on Programming*, pages 2–19. Lecture Notes in Computer Science, Vol. 1576, S.D. Swierstra (ed.), Springer-Verlag, New York, NY, 1999.

[2] C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in Hoare logic. In *ACM-SIGPLAN 2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*. ACM Press, September 2000.

[3] P. Cohen. *Set Theory and the Continuum Hypothesis*. Benjamin, San Fancisco, 1966.

[4] P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 843–993. Elsevier, Amsterdam, and The MIT Press, Cambridge, Mass., 1990.

[5] F. de Boer. A WP calculus for OO. In *Proceedings of FOSSACS'99*, 1999.

[6] F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, may 1995.

[7] S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. To appear in POPL'01, 2001.

[8] R. Jones and R. Lins. *Garbage Collection*. Wiley, 1996.

[9] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.

[10] D. C. Oppen and S. A. Cook. Proving assertions about programs that manipulate data structures. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation*, pages 107–116, Albuquerque, New Mexico, 5–7 May 1975.

[11] J.C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, Palgrave, 2000.