Program Logic and Equivalence in the Presence of Garbage Collection *

Cristiano Calcagno[1,2], Peter O'Hearn[1], and Richard Bornat[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. It is generally thought that reasoning about programs in memory safe, garbage collected languages is much easier than in languages where the programmer has more explicit control over memory. Paradoxically, 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 study of a small language whose operational semantics includes a rule for reclaiming garbage. Our main results include an analysis of propositions that are garbage insensitive, soundness of Hoare logic rules for the garbage collecting semantics, and full abstraction results connecting partial and total correctness to 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. It can also be used in a lower level language like C, coexisting with explicit deallocation primitives [10]. 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. Paradoxically, 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.

* A preliminary version of this paper appeared as "On Garbage and Program Logic" in the proceedings of FOSSACS'01, published as Springer LNCS 2030. This research was supported by the EPSRC under the "Verified Bytecode" and "Local Reasoning about State" projects.

1.1 A Logical Conundrum

The following problem was raised by Reynolds in [16].

Consider a statement form $x := \texttt{cons}(E, E')$ 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 by destroying the sole existing pointer to the cell, thus making it unreachable.

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

Now, ask the question: is there a pointer y to a cell in the heap that has 3 in its car and 4 in its cdr, 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.

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

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

The problem is that on termination there might not actually be a cell whose car is 3 and cdr is 4, if a garbage collector reclaims the detached cell. (Here, $y \mapsto 3, 4$ is a predicate that says that y points to a binary cons cell holding 3 and 4.)

It is worth looking at this example in more detail. After the statement $x := \texttt{cons}(3, 4)$ 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 $\{true\}\, x := \texttt{cons}(3, 4)\, \{\exists y.\, y \mapsto 3, 4\}$. But then either Hoare's or Floyd's assignment axiom gives us $\{\exists y.\, y \mapsto 3, 4\}\, x := z\, \{\exists y. y \mapsto 3, 4\}$ 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 is 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 $\exists y.\, y \mapsto 3, 4$ is just itself, so we only get $\{\exists y.\, y \mapsto 3, 4\}\, x := z\, \{\exists y.\, y \mapsto 3, 4\}$. Thus, if there are states where this precondition is false, then the logical context $\{true\} - \{\exists y.\, y \mapsto 3, 4\}$ 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 $x := \texttt{cons}(7, 8);\ x := z$ in place of the single statement $x := z$.)

The upshot is that previous program logics for pointer programs are unsound in the presence of garbage collection, including all of the pointer logic references in [5, 3, 9] (with the exception of [7]; see below). 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 [15], 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 $\{true\} - \{\exists y.\, y \mapsto 3, 4\}$ behaves exactly as described above. In a series of papers, the most recent of which is [6], 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 this section. Finally, Honsell, Smith, Mason and Talcott [8] 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 [2] very nearly qualifies, except for the use of a garbage sensitive atomic predicate hs for describing sharing constraints.

1.2 Reactions

There are several ways to react to the conundrum. One is to lay the blame on the Floyd-Hoare approach to assignment, and its emphasis on substitution. This reaction is difficult to uphold. For, although the soundness of the assignment axiom requires certain assumptions (such as no clashes between named variables), none of these are the essential problem here.

Another reaction is to regard the conundrum as inconsequential. For, the unsoundness phenomenon mentioned above is not disastrous. The various logics are consistent, as they are sound in non-garbage-collecting models, and the unsoundness only seems to show up in "useless" formulae such as $\exists y.\, y \mapsto 3, 4$. The logic will not lead us wrong on formulae that depend only on reachable elements, such as $x \mapsto 3, 4$.

This reaction has some merit, but is probably too glib. Some analysis should be provided of the propositions which do not lead you wrong.

A third reaction is to maintain the Floyd-Hoare approach, but to enrich the syntax of pre and postconditions with an extra parameter to keep track of the heap and its relation to program variables. This is the approach taken by Hoare and Jifeng [7], where there is an algebraic operation $\widehat{state}$ that removes garbage from the state. Then, the assignment $x := z$ can be modelled with an axiom that keeps track of alterations to heap cells, resulting from garbage collection, as well as stack variables.

$$\{P[\widehat{state'}/state]\}\, x := z\, \{P\} \qquad state' = (state \mid x \mapsto state(z))$$

By using $\widehat{(\cdot)}$, Hoare and Jifeng can keep the heap garbage collected at all times.

This third approach is certainly theoretically adequate. Its main regret is the need to carry around an explicit state parameter. However, starting from such an adequate position one might attempt to find situations when one can soundly hide the state parameter, to treat it implicitly in pre and postconditions as is usually done. We emphasize that this step from explicit to implicit state is hazardous, a potential source of unsoundness (as the conundrum shows). The problem is that in the process of hiding the state we might lose substitution-relevant syntactic information, which plays a crucial role in program logic. Indeed, the results of this paper might be viewed as an analysis of situations where implicit state is sound, a hidden-state cousin of the solution of Hoare and Jifeng.

1.3 Overview

The approach we take in this paper concentrates the impact of garbage collection in one place: the properties expressed by pre and postconditions. In the program logic, the impact is limited to the interpretation of implications used to strengthen preconditions or weaken postconditions. The idea is that if formulae are restricted, or their semantics altered, to ensure that propositions are insensitive to garbage, then there will be no need to doctor the other rules, for the various statement forms, to take account of the possibility of the collector taking control. Simplicity in the program logic can be obtained by limiting the expressivity of the formulae used in pre and postconditions.

We consider two forms of semantics, one based on total states (which have no dangling pointers) and the other based on partial states (which allow dangling). The partial semantics is the more theoretically cohesive, but is perhaps less approachable initially. After all, in a memory safe language like the one we will consider dangling pointers never arise during program execution. So we begin with the total semantics.

We use a possible-world semantics, where the current heap (a finite collection of cons 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. In this way the total states model adopts a view of the state as finite but arbitrarily extensible. The other connectives are interpreted pointwise, without changing worlds.

This semantics provides the basis for a resolution of the conundrum, but at the price of an unusual semantics of $\exists$. We also observe that the normal semantics of $\exists$ can be retained, as long as we rule out problematic formulae like $\exists y. y \mapsto 3, 4$. This is done using a form of guarded quantifier, expressed via a syntactic restriction on propositions.

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.

After showing these results we move on to consider partial states. We establish a connection between the total states model and a partial states model based on the celebrated "dense topology" semantics of classical logic [4, 12].

We also consider the relation to an intuitionistic semantics in the style of Kripke [11]. This is accompanied by a low-level interpretation of Hoare triples which is garbage insensitive but not fully abstract. We argue that this lower-level view has some advantages over the more abstract perspective of the total states model. In particular, it allows us to employ the method of local reasoning described in [14].

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.

There is a surprising amount of mathematical structure that follows from this simplistic model, but for the reader mainly interested in the implications for program proving much of the theoretical material we develop can be skipped. Such a reader might concentrate on: (i) the Guarding Lemma in section 4, which identifies a class of propositions that excludes the problematic example above; (ii) the Garbage Insensitivity Theorem in section 5; (iii) the spatial conjunction and program logic in sections 10 and 11; (iv) the lower level semantics and associated logic in sections 13 and 14.

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.

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

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 $dom(s)$ or $dom(h)$ for the domain of definition of a stack or heap.

The collection of `Pointers` can be taken either to be an infinite set, or a finite set if we are concerned with bounded memory. None of the results in this paper depend on the size of `Pointers`.

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 $x \in dom(s)$ 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 $p \in dom(h)$ 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.

- $prune(s, h) \; = (s, g)$, 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 7, 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:

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

We have not included expressions $E.1$ or $E.2$ for dereferencing a pointer. It is theoretically simpler to exclude such operations from expressions, and instead have dereferencing as part of the command forms. In the assertion language dereferencing will be accomplished using an atomic predicate, the points-to relation.

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

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

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.

$$
\begin{array}{rll}
[\![x]\!]s \triangleq & s(x) & \\
[\![E_1 + E_2]\!]s \triangleq & [\![E_1]\!]s + [\![E_2]\!]s & \text{if } [\![E_1]\!]s \in \texttt{Nat} \text{ and } [\![E_2]\!]s \in \texttt{Nat} \\
& \mathit{wrong} & \mathit{otherwise} \\
[\![E_1 == E_2]\!]s \triangleq & \mathit{wrong} & \text{if } [\![E_1]\!]s = \mathit{wrong} \text{ or } [\![E_2]\!]s = \mathit{wrong} \\
& \mathit{true} & \text{if } \mathit{wrong} \neq [\![E_1]\!]s = [\![E_2]\!]s \\
& \mathit{false} & \text{if } \mathit{wrong} \neq [\![E_1]\!]s \neq [\![E_2]\!]s \neq \mathit{wrong} \\
[\![\texttt{not}\, E]\!]s \triangleq & \mathit{false} & \text{if } [\![E]\!]s = \mathit{true} \\
& \mathit{true} & \text{if } [\![E]\!]s = \mathit{false} \\
& \mathit{wrong} & \text{otherwise}
\end{array}
$$

4 Propositions

The grammar for propositions P is as follows.

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

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

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 $\exists yz. x \mapsto y, z$ says that x is a pointer. In practice, one would also want atomic predicates for describing reachability properties (e.g. [2]), 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 $\mathrm{Free}(P) \subseteq \mathit{dom}(s)$, where $\mathrm{Free}(P)$ indicates the set of variables occurring free in P.

The quantifier-free fragment is straightforward.

$$
\begin{array}{lll}
s, h \models E = E' & \overset{\Delta}{\iff} & [\![E]\!]s = [\![E']\!]s = v \in \texttt{Values} \\
s, h \models E \mapsto E_1, E_2 & \overset{\Delta}{\iff} & h([\![E]\!]s) = \langle [\![E_1]\!]s, [\![E_2]\!]s \rangle \\
s, h \models \texttt{false} & & \text{never} \\
s, h \models P \to Q & \overset{\Delta}{\iff} & \text{if } s, h \models P \text{ then } s, h \models Q
\end{array}
$$

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

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

($(s \mid x \mapsto v)$ 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.

Another approach that is often suggested is to restrict quantifiers so that they range over reachable elements only.

EXISTENTIAL: SECOND TRY

$$\begin{aligned} s, h \models \exists x.P \stackrel{\Delta}{\iff} & \exists v \in \texttt{Values}.(s \mid x \mapsto v), h \models P \text{ and} \\ & v \in \texttt{Pointers} \text{ implies } v \text{ is reachable in } s, h \end{aligned}$$

The right-hand side is now well-defined, as the reachability requirement ensures that $(s \mid x \mapsto v)$ is total. Although tantalizing, this notion has several obstacles to overcome. It invalidates Hoare's assignment axiom, since $\{\exists y.\, y \mapsto 3, 4\}\, x := z\, \{\exists y.\, y \mapsto 3, 4\}$ fails for it. This happens when $x := z$ detaches a cell with car 3 and cdr 4. 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.

The solution we adopt in this paper 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

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

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

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

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; $\exists y.\, y \mapsto 3, 4$ 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 8 and

9, 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

$$\forall xy.\, \exists z.\, z \mapsto x, y$$

This says that for any pair of values we consider, there is some cell we can find which has the pair as its contents. This illustrates the extensible heap view taken by the semantics, where there is an inexhaustible 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, $\forall xy$ could saturate the set of available pointers leaving none for $\exists z.\, z \mapsto x, y$.)

Many specifications one uses when reasoning about pointer programs are garbage insensitive. For example: x points to a linked list; or x points to a linked list representing the sequence L. With the aid of recursive definitions, these properties can be expressed in our language. For example,

$$list(x) \;\Leftrightarrow\; x = \mathtt{nil} \;\vee\; (\exists z.\, x \mapsto -, z \;\wedge\; list(z))$$

can be regarded as a definition of "x points to a linked list".

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".

The semantics of $\exists$ might seem forbidding. When working with a memory safe language, and a total states model, it is hard to imagine an intuitive reason to quantify over non-reachable elements in program specifications that typically arise. Furthermore, the "problem" with the first and second attempts at a semantic definition arise from a seemingly silly example: why would one ever want to write an assertion of the form $\exists y.\, y \mapsto 3, 4$?

We can partially relieve this bother by observing that, in a variety of naturally occurring cases, the official definition dovetails with either of the first or the second. Put briefly, if we consider a guarded form of quantifier, where we know not just that the quantified variable is reachable but also how to reach it, then there is no need to select a larger heap.

We formulate a notion of guarding which involves looking only one-deep into the heap; deeper looking could also be considered.

Lemma 2 (Guarding Lemma). *Suppose y is a variable different from x and G is of the form $y \mapsto x, E$ or $y \mapsto E, x$. Then*

$$s, h \models \exists x. G \wedge P \iff \exists v \in \mathtt{Values}. (s \mid x \mapsto v), h \models G \wedge P$$

This lemma can be used to justify a simple semantics in many examples. But we will work with the more general semantics, the "official version", when proving theoretical properties. Also, the official semantics will prove useful later when we show a link to a semantics based on partial states, where quantifying over unreachable locations is more natural.

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* $\mathit{prune}(s, h) \models P$.

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. This fact is not represented explicitly in the semantic domains (it could be by incorporating naturality conditions), but holds for all definable propositions. If we were to allow unrestricted pointer arithmetic, then propositions (or programs) *would* be sensitive to the effects of a garbage collector; renaming is thus a property that is closely linked (though not in a logically necessary way) with memory safety.

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, 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 cases of the equality predicates and points-to relation are immediate from the definition. ■

An amusing point is that the proof of *Shrinkage*, the most unusual property of the semantics, is straightforward; the real work was in the *Growth* case of $\exists$. However, if we were to take $\forall$ rather than $\exists$ as primitive, then *Growth* would have been trivial, and the work would have had to be put in for *Shrinkage*.

6 Quantifier Laws

The semantics of section 4 clearly satisfies all of the laws of classical propositional logic. Close consideration of quantifier laws is essential, however, to justify the semantic clause for $\exists$.

We did not consider the quantifier laws before now because the validity of one of them, the elimination rule, is by no means immediate: it depends on Garbage Insensitivity. (To be precise, the proof of the elimination rule will refer only to Growth, but Growth itself depends on Shrinkage in the case of $\to$.)

To state the quantifier laws we set up a notion $P \models Q$ of logical consequence, where P and Q are propositions.

Semantic Consequence.
$P \models Q \overset{\Delta}{\iff}$ for all s, h if $s, h \models P$ then $s, h \models Q$.

(If we were being pedantic, we would parameterize the judgement form with a context X of variables containing those free in P and Q, and require that all stacks in the definition have domain X. To avoid notational clutter, we simply say that all semantic judgements are assumed to be well formed in our definitions. Similar remarks apply to the definitions of Hoare triples and observational equivalence given later.)

Theorem 2 (Quantifier Laws Theorem). *The semantics validates the usual rule of* $\exists$ *elimination*

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

and the following version of $\exists$ *introduction:*

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

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.

In the standard semantics of classical logic, the validity of $\exists$ elimination rests on a weakening lemma:

If $s \models P$ then $(s \mid x \mapsto v) \models P$

where v is a value and x is not free in P. Our semantics has a world component as well, so we might attempt:

If $s, h \models P$ then $(s \mid x \mapsto v), h \models P$.

But now we run into the problem of total states again, since if v is not defined in h, then $(s \mid x \mapsto v), h$ will not be total. The correct property is as follows.

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* $(s \mid x \mapsto v), g \models P$*.*

Proof: First, we we consider the following

Claim

$$s, h \models P \quad \Longleftrightarrow \quad \forall v \in \texttt{Values}.\ \text{if } v \in \mathit{dom}(h) \text{ or } v \notin \texttt{Pointers} \text{ then } (s \mid x \mapsto v), h \models P$$

The proof of the claim is by induction on P. The cases of equality and *false* are straightforward, and the two subcases of $\to$ each exercise both directions of the $\Longleftrightarrow$.

We consider $\exists$ in detail, where we α-rename the bound variable to $\exists y.P$, where y is different from x. For the $\Rightarrow$ direction, if $s, h \models \exists y.P$ then $(s \mid y \mapsto v_y), g \models P$ for some v_y and $g \sqsupseteq h$. Now if $v \in \mathit{dom}(h)$ or $v \notin \texttt{Pointers}$ then $((s \mid y \mapsto v_y) \mid x \mapsto v), g$ is total. Also, $((s \mid y \mapsto v_y) \mid x \mapsto v) = ((s \mid x \mapsto v) \mid$

$y \mapsto v_y)$. So, by the induction hypothesis, we get $((s \mid x \mapsto v) \mid y \mapsto v_y), g \models P$, and by the definition of $\exists$ this means $(s \mid x \mapsto v), h \models \exists y.P$

For the $\Leftarrow$ direction, we obtain $(s \mid x \mapsto \mathtt{nil}), h \models \exists y.P$ by choosing $v = \mathtt{nil}$. Then $((s \mid x \mapsto \mathtt{nil}) \mid y \mapsto v_y), g \models P$ for some v_y and $g \sqsupseteq h$. By induction we obtain $(s \mid y \mapsto v_y), g \models P$ and by the definition of $\exists$ this implies $s, h \models \exists y.P$. This completes the proof of the Claim.

To prove the lemma, suppose $s, h \models P$, $g \sqsupseteq h$ and $(s \mid x \mapsto v), g$ is total. Then s, g is also a total state, and by the *Growth* part of Garbage Insensitivity we obtain $s, g \models P$. Since $(s \mid x \mapsto v), g$ is total, we have that either $v \in dom(g)$ or $v \notin \mathtt{Pointers}$. By the Claim for s, g we obtain $(s \mid x \mapsto v), g \models P$. ∎

We can now prove the validity of $\exists$ elimination as follows. If $s, h \models Q$ then $(s \mid x \mapsto v), g \models P$ for some $g \sqsupseteq h$ by the top right premise and the definition of $\exists$. By the Weakening Lemma, $(s \mid x \mapsto v), g \models Q$. By the top left premise, $(s \mid x \mapsto v), g \models R$, and by Weakening and Garbage Insensitivity $s, h \models R$.

The validity of the $\exists$ introduction rule is comparatively straightforward. If $s, h \models Q$ then $[\![E]\!]s = v \in \mathtt{Values}$ is not wrong, by the assumption $E = E$, and $s, h \models P[E/x]$ by the top left premise, and $(s \mid x \mapsto v), h \models P$ by the Substitution Lemma to follow. (Note that no change of world is needed here.)

Lemma 5 (Substitution Lemma). *If* $[\![E]\!]s = v \in \mathtt{Values}$ *then the following hold*

1. $[\![E'[E/x]]\!]s = [\![E']\!](s \mid x \mapsto v)$
2. $s, h \models P[E/x]$ *iff* $(s \mid x \mapsto v), h \models P$.

The proof of the Substitution Lemma is not difficult, and does not much exercise the change of world used to interpret $\exists$ because the value v is reachable in s, h (since it is defined by a term, E).

7 A Programming Language and Garbage Collecting Semantics

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

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

The binding form $\mathtt{newvar}\ x.\ C\ \mathtt{end}$ 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 `newvar` we introduce a new command $\mathtt{dealloc}(x)$; 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.

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

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

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

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

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

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

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

$$\frac{g \sqsubseteq h,\ s, g\ \text{total}}{s, h \rightsquigarrow s, g} \qquad \frac{s, h =_\alpha s, g}{s, h \rightsquigarrow s, g}$$

The last two rules are for garbage collection and relocation. In the third-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.

Although we will not attempt a precise formulation, it should be clear that the language is memory safe: no execution sequence ever goes wrong from an attempt to dereference a pointer not in the domain of the current heap.

8 Partial Correctness

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

Partial Correctness
$\{P\}\, C\, \{Q\}$ is true just if for all total states s, h with $dom(s) \supseteq \mathrm{Free}(P, C, Q)$, 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 `newvar`.

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

The proof of the theorem occupies the remainder of this section. It goes by first relating $\cong_{pc}$ to a semantics of commands which ignores garbage and renaming of locations, and then relating this semantics to partial correctness.

Consider a semantics $[\![-]\!]_{pc}$ of commands

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

where `TStates` is the set of total states, and $\mathtt{TStates}_* \stackrel{\Delta}{=} \mathtt{TStates} \cup \{wrong\}$. The definition of $[\![C]\!]_{pc}$ is as follows

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

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

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

where $=_\alpha$ is equality modulo renaming of locations.

Lemma 6. $[\![C]\!]_{pc}$ *induces a partial function between quotients*

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

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

Lemma 7. $C \cong_{pc} C' \Leftrightarrow [\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc}$.

Proof: In the $\Leftarrow$ direction, assume $[\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc}$. Then (lemma 6) $[\![G[C]]\!]^{\sim}_{pc} = [\![G[C']]\!]^{\sim}_{pc}$ for all closing contexts $G[-]$. The conclusion $C \cong_{pc} C'$ follows by observing that the properties "goes wrong" and "terminates normally" in the definition of $\cong_{pc}$ are preserved by $[\![-]\!]^{\sim}_{pc}$.

For the $\Rightarrow$ direction, assume $[\![C]\!]^{\sim}_{pc} \neq [\![C']\!]^{\sim}_{pc}$. Then there exist $c \in \mathtt{TStates}/\!\sim$ and $d \in \mathtt{TStates}_*/\!\sim_*$ such that

$$[\![C]\!]^{\sim}_{pc}(c) = d \qquad [\![C']\!]^{\sim}_{pc}(c) \neq d$$

In general there exists a command $\mathit{Create}(c)$ that creates a store in class c from the empty store. There are two cases:

1. Case $d = \mathit{wrong}$. Consider the following context

   $$G[-] \triangleq \mathit{Create}(c); -$$

   Then $G[C]$ "goes wrong" and $G[C']$ does not, thus $C \not\cong_{pc} C'$.
2. Case $d \neq \mathit{wrong}$. In general there exists a command ending in a boolean expression $\mathit{Check}(d)$ that evaluates to true if the current state is in class d, and to wrong otherwise (to define an analogous returning false instead of wrong one would need language constructs to test if $x.i$ is defined before de-referencing).
   To define $\mathit{Check}(d)$, consider a garbage-free store (s, h) in class d. For each $p \in \mathit{dom}(h)$, there exists a path in the heap reaching p from a stack variable x, let this path be represented by the sequence $i_1 \cdot \ldots \cdot i_n$ where each i_j represents the component followed at step j. Define the command

   $$C_p \triangleq x_p := x; x_p := x_p.i_1; \ldots; x_p := x_p.i_n; x_p^1 := x_p.1; x_p^2 := x_p.2$$

   with free variables x_p, x_p^1, x_p^2.
   Define the expression $E_p \triangleq x_p$, extended to values with $E_v \triangleq v$ if v is a number, or a boolean, or nil. We can now define $\mathit{Check}(d)$:

   $$CH_1 \triangleq \bigwedge_{p \neq q \in \mathit{dom}(h)} E_p \neq E_q$$
   $$CH_2 \triangleq \bigwedge_{(x,v) \in s} x == E_v$$
   $$CH_3 \triangleq \bigwedge_{(p,\langle v_1,v_2\rangle) \in h} (x_p^1 == E_{v_1}) \wedge (x_p^2 == E_{v_2})$$
   $$\mathit{Check}(d) \triangleq \mathtt{newvar}\, \boldsymbol{x_p}, \boldsymbol{x_p^1}, \boldsymbol{x_p^2}.\ \left(\mathit{Seq}_{p \in \mathit{dom}(h)} C_p\right);$$
   $$\mathtt{if}\ \mathtt{not}(CH_1 \wedge CH_2 \wedge CH_3)\ \mathtt{then}\ (\mathit{nil}.1 == \mathit{nil}.1)\ \mathtt{else}\ \mathit{true}$$

   where $\boldsymbol{x_p}, \boldsymbol{x_p^1}, \boldsymbol{x_p^2}$ are sequences of variables for $p \in \mathit{dom}(h)$ and Seq is sequencing of commands.

Finally, consider the context

$$G[-] \triangleq \mathit{Create}(c); -; \texttt{if } \mathit{Check}(d) \texttt{ then skip}$$

Then $G[C]$ terminates normally, and $G[C']$ does not, thus $C \not\cong_{pc} C'$.

■

Lemma 8. $[\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc} \Leftrightarrow C$ *and* C' *satisfy the same partial correctness assertions.*

Proof: In the $\Rightarrow$ direction, assume $[\![C]\!]^{\sim}_{pc} = [\![C']\!]^{\sim}_{pc}$. Since the semantics of propositions respects $\sim$ (immediate consequence of garbage insensitivity and invariance under renaming), and since the definition of partial correctness involves the properties "terminates normally" and "goes wrong", we can conclude that C and C' satisfy the same partial correctness assertions.

For the $\Leftarrow$ direction, suppose that $[\![C]\!]^{\sim}_{pc} \neq [\![C']\!]^{\sim}_{pc}$. For each $c \in (\texttt{TStates}/\sim)$ there exists a proposition P_c such that P_c is true if and only if the current state is in class c.

$$P_c \triangleq \exists \boldsymbol{x_p}.\, CH'_1 \wedge CH'_2 \wedge CH'_3$$

where the propositions CH'_i are analogous to the expressions CH_i in the definition of $\mathit{Check}(c)$ in the proof of lemma 7:

$$CH'_1 \triangleq \bigwedge_{p \neq q \in dom(h)} E_p \neq E_q$$

$$CH'_2 \triangleq \bigwedge_{(x,v) \in s} x = E_v$$

$$CH'_3 \triangleq \bigwedge_{(p,\langle v_1,v_2\rangle) \in h} E_p \mapsto E_{v_1}, E_{v_2}$$

Note that the existential does not exercise the change of world since the value of all the quantified variables is determined by assertions $- \mapsto -, -$. There are three cases:

1. Case $[\![C]\!]^{\sim}_{pc}(c) = wrong$ and $[\![C']\!]^{\sim}_{pc}(c) \neq wrong$. Then $\{P_c\}\, C\, \{true\}$ is false and $\{P_c\}\, C'\, \{true\}$ is true.
2. Case $[\![C]\!]^{\sim}_{pc}(c) = d \neq wrong$ and $[\![C']\!]^{\sim}_{pc}(c)$ is undefined. Then $\{P_c\}\, C\, \{false\}$ is false and $\{P_c\}\, C'\, \{false\}$ is true.
3. Case $[\![C]\!]^{\sim}_{pc}(c) = d$ and $[\![C']\!]^{\sim}_{pc}(c) = d'$ with $d, d' \neq wrong$ and $d \neq d'$. Then $\{P_c\}\, C\, \{P_d\}$ is true and $\{P_c\}\, C'\, \{P_d\}$ is false.

In all the cases C and C' do not satisfy the same partial correctness assertions.
■

So far we have considered a notion of observational equivalence, but a natural question is if the results can be generalized to a notion of observational preorder.

For the logical view, a natural generalization is asking whether one command satisfies *at least* the partial correctness assertions that the other does. As for equivalence, we proceed through the semantics $[\![-]\!]_{pc}$, so we define a preorder on the set of functions $(\texttt{TStates}/\sim) \rightharpoonup (\texttt{TStates}_*/\sim_*)$:

$$f \leq f' \stackrel{\Delta}{\iff} \text{for each } c \in dom(f),\ f'(c) \text{ is defined and } f(c) \leq f'(c)$$

where $d \leq d' \stackrel{\Delta}{\iff} d = d' \in (\texttt{TStates}/\sim)$ or $d' = wrong$.

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

The following theorem relates the above orders.

Theorem 4 (Ordered Full Abstraction Theorem, Partial Correctness). *The following are equivalent:*

1. *For each P, Q, if $\{P\}\, C\, \{Q\}$ is true then $\{P\}\, C'\, \{Q\}$ is true.*
2. *$[\![C']\!]_{pc} \leq [\![C]\!]_{pc}$.*
3. *$C' \sqsubseteq_{pc} C$.*

Note that the orders in point 2 and 3 in the above theorem are inverted, since a less defined command satisfies more partial correctness assertions. In particular, an always-diverging command satisfies all the assertions.

9 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 5 (Full Abstraction Theorem, Total Correctness). *$C \cong_{tc} C'$ iff C and C' satisfy the same total correctness assertions.*

Similarly to what we did with partial correctness, we define a total semantics on quotients

$$[\![C]\!]^{\sim}_{tc} : (\mathtt{TStates}/\sim) \to (\mathtt{TStates}_{\perp}/\sim_{\perp})$$

where $\mathtt{TStates}_{\perp} \stackrel{\Delta}{=} \mathtt{TStates} \cup \{\perp\}$, $\sim_{\perp}$ extends $\sim$ with $(\perp, \perp)$, and $\perp$ conflates non-termination and getting stuck. The formal definition of $[\![C]\!]_{tc}$ is as follows

$$\begin{array}{l} [\![C]\!]^{\sim}_{tc}(c) = d \stackrel{\Delta}{\Longleftrightarrow} [\![C]\!]^{\sim}_{pc}(c) = d \quad \text{when } d \in \mathtt{TStates}/\sim \\ [\![C]\!]^{\sim}_{tc}(c) = \perp \stackrel{\Delta}{\Longleftrightarrow} [\![C]\!]^{\sim}_{pc}(c) = \{wrong\} \text{ or } [\![C]\!]^{\sim}_{pc}(c) \text{ undefined} \end{array}$$

To justify the definition of $[\![C]\!]^{\sim}_{tc}$ we observe that the identification of non-termination and stuckness is compatible with the (operational) semantics of commands, hence it is compositional.

Lemma 9. $C \cong_{tc} C' \Leftrightarrow [\![C]\!]^{\sim}_{tc} = [\![C']\!]^{\sim}_{tc}$.

Proof: In the $\Leftarrow$ direction, assume $[\![C]\!]^{\sim}_{tc} = [\![C']\!]^{\sim}_{tc}$. Then $[\![G[C]]\!]^{\sim}_{tc} = [\![G[C']]\!]^{\sim}_{tc}$ for all closing contexts $G[-]$. The conclusion $C \cong_{tc} C'$ follows by observing that the property "terminates normally" in the definition of $\cong_{tc}$ is preserved by $[\![-]\!]^{\sim}_{tc}$ (the same would not hold of property "goes wrong").

For the $\Rightarrow$ direction, assume $[\![C]\!]^{\sim}_{tc} \neq [\![C']\!]^{\sim}_{tc}$. Then there exist c, d, d' such that $[\![C]\!]^{\sim}_{tc}(c) = d$ and $[\![C']\!]^{\sim}_{tc}(c) = d'$ and $d \neq d'$, hence one of d, d' must be different from $\perp$, say d. Consider the context

$$G[-] \stackrel{\Delta}{=} \mathit{Make}(c); -; \mathit{Check}(d)$$

Then $G[C]$ terminates normally while $G[C']$ does not, hence $C \not\cong_{tc} C'$. ■

Lemma 10. *$[\![C]\!]^{\sim}_{tc} = [\![C']\!]^{\sim}_{tc} \Leftrightarrow C$ and C' satisfy the same total correctness assertions.*

Proof: In the $\Rightarrow$ direction, assume $[\![C]\!]^{\sim}_{tc} = [\![C']\!]^{\sim}_{tc}$. Since the semantics of propositions respects $\sim$, and since the definition of total correctness involves the property "terminates normally" (and not "goes wrong"), we can conclude that C and C' satisfy the same total correctness assertions.

For the $\Leftarrow$ direction, assume $[\![C]\!]^{\sim}_{tc} \neq [\![C']\!]^{\sim}_{tc}$. Then there exist c, d, d' such that $[\![C]\!]^{\sim}_{tc}(c) = d$ and $[\![C']\!]^{\sim}_{tc}(c) = d'$ and $d \neq d'$, hence one of d, d' must be different from $\perp$, say d. Now, $[P_c]\, C\, [P_d]$ is true and $[P_c]\, C'\, [P_d]$ is false. This concludes the proof that C and C' do not satisfy the same total correctness assertions. ■

As for partial correctness, the above results generalize to preorders. The partial order on functions $(\mathtt{TStates}/\sim) \to (\mathtt{TStates}_{\perp}/\sim_{\perp})$ is defined as

$$f \leq f' \stackrel{\Delta}{\Longleftrightarrow} f(c) \leq f'(c) \text{ for each } c \in (\mathtt{TStates}/\sim)$$

where $d \leq d' \stackrel{\Delta}{\Longleftrightarrow} d = d' \in (\mathtt{TStates}/\sim)$ or $d = \perp$.

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

The following theorem relates the above orders.

Theorem 6 (Ordered Full Abstraction Theorem, Total Correctness). *The following are equivalent:*

1. *For each P, Q, if $[P]\, C\, [Q]$ is true then $[P]\, C'\, [Q]$ is true.*
2. $[\![C]\!]_{tc} \leq [\![C']\!]_{tc}$.
3. $C \sqsubseteq_{tc} C'$.

A difference emerges with respect to partial correctness: while less defined commands prove more partial correctness assertions, they prove fewer total correctness assertions. Note however that in partial correctness *wrong* was the "most defined" element, and in total correctness it is conflated with undefined.

10 Partial States and the Spatial Conjunction

Thus far we have we focussed on total states. Our purpose in this and subsequent sections is to argue that partial states, which allow dangling pointers, are conceptually and technically useful, even in a memory safe language where dangling pointers do not arise during execution. The conceptual point is that, often, a program uses only portion of a state, a portion that does not include all pointers reachable from a given point; a partial state provides a convenient description of the information that is needed. The most important tool for realizing this intuition, the spatial conjunction $P_1 * P_2$ [16], is described in this section. The technical point, pursued in later sections, is that partial states make the connection to other forms of interpretation clear; this in turn clears the way for certain improvements.

We begin by defining a notion of support, which works for partial states. The idea is that a state s, h supports P if it contains enough information to conclude P. Technically, this means that, for any total state we wish to extend the partial state to, P will hold.

Suppose s, h is a state, perhaps partial. Then

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

We use support to define the spatial conjunction. Intuitively, $P * Q$ is true when the current heap can be split into parts, one of which gives enough information to conclude P and the other of which supports Q. This heap splitting generally results in partial subheaps.

Given heaps h and g, the notation $h\#g$ means that h and g have disjoint domains, and in that case $h * g$ is defined by $(h * g)(p) = v$ iff $h(p) = v$ or $g(p) = v$.

Spatial Conjunction

$s,h \models P * Q \stackrel{\Delta}{\iff} \exists h_1, h_2.\, h_1 \# h_2,\ h_1 * h_2 = h,\ s, h_1 \Vdash P$ and $s, h_2 \Vdash Q$.

Notice that partial states play a subsidiary role in the definition of $\models$, which is defined for total states only. That is, the total states judgement $s, h \models P * Q$ is defined by reference to the partial states judgement form $\Vdash$.

The heap splitting in the definition ensures that x and y must be distinct cells in a formula of the form $x \mapsto a, b * y \mapsto c, d$. However, the area of heap used to satisfy $x \mapsto a, b$ when such a splitting is chosen does not have to include a and b: the disjointness in heap splitting is only with respect to the domain, not the range, of the heap. As a result, there can be overlap between pointers to the left of $\mapsto$ in one conjunct and to the right of $\mapsto$ in another. This difference in the treatment of the domain and range is the essential device that allows linking to go together with disjointness. For example, $x \mapsto 3, y * y \mapsto 4, x$ describes a two-element, circular linked list.

The locality provided by the spatial conjunction comes out most immediately in propositions of the form $x \mapsto a, b * Q$. When such a proposition holds, we know that Q must be supported by a partial heap that does not include pointer x in its domain. As a result, if we update x's car or cdr then Q will still be supported. This is the basis for a program logic devised by Reynolds in [16] and further developed in [9, 14]. We give a garbage insensitive variant of it in the next section.

Finally, a useful fact is that the spatial conjunction can be used in place of $\wedge$ when guarding $\exists$.

Lemma 11 (Guarding Lemma, II). *Suppose y is a variable different from x and G is of the form $y \mapsto x, E$ or $y \mapsto E, x$. Then*

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

11 Program Logic, I

Here are the Hoare Logic axioms for the four basic statements.

Heap update:

$\{Q * x \mapsto a, b\}\, x.1 := c\, \{Q * x \mapsto c, b\}$

$\{Q * x \mapsto a, b\}\, x.2 := c\, \{Q * x \mapsto a, c\}$

Allocate:

$\{P\}\, x := \texttt{cons}(y, z)\, \{P * x \mapsto y, z\}$

where $x \notin FV(P)$.

Heap lookup:

$\{z \mapsto a, b \wedge Q[a/x]\}\, x := z.1\, \{z \mapsto a, b \wedge Q\}$

$\{z \mapsto a, b \wedge Q[b/x]\}\, x := z.2\, \{z \mapsto a, b \wedge Q\}$

Hoare Assignment:

$$\{P[E/x] \wedge E = E\}\, x := E\, \{P\}$$

It is assumed here that the variables a, b, c, x, y and z are distinct. We have given restricted versions of the allocation and lookup axioms. When x is free in P in the former, or x and z are the same variable in the latter, then existential quantification in the postcondition can be used to talk about the previous value of x [16].

The final axiom has $E = E$ in the precondition to ensure that E does not produce a wrong answer. In many cases this can be avoided. For example, if E is a variable we know that it won't go wrong, so that $E = E$ will automatically hold. We need similar definedness checks in the rules for `while` and `if`.

$$\frac{\{P\}\, C\, \{Q\}}{\{P\}\, \texttt{newvar}\, x.\, C\, \{Q\}}\; x \notin \mathrm{Free}(P, Q) \qquad \frac{\{P\}\, C_1\, \{Q\} \quad \{Q\}\, C_2\, \{R\}}{\{P\}\, C_1; C_2\, \{R\}}$$

$$\frac{\{P \wedge (E = \texttt{true})\}\, C\, \{Q\} \quad \{P \wedge E = \texttt{false}\}\, C'\, \{Q\}}{\{P\}\, \texttt{if}\, E\, \texttt{then}\, C\, \texttt{else}\, C' \{Q\}}$$

$$\frac{\{P \wedge E = \texttt{true}\}\, C\, \{P\}}{\{P\}\, \texttt{while}\, E\, \texttt{do}\, C\, \{P \wedge B = \texttt{false}\}}$$

In addition, there are three rules for manipulating triples.

Auxiliary Variable Elimination

$$\frac{\{P\}\, C\, \{Q\}}{\{\exists x. P\}\, C\, \{\exists x. Q\}}\; x \notin \mathrm{Free}(C)$$

Variable Substitution

$$\frac{\{P\}\, C\, \{Q\}}{(\{P\}\, C\, \{Q\})[E_1/x_1, ..., E_k/x_k]}\; \begin{array}{l} \{x_1, ..., x_k\} \supseteq \mathrm{Free}(P, C, Q), \text{ and} \\ x_i \in \mathrm{Modifies}(C) \text{ implies} \\ E_i \text{ is a variable not free in any other } E_j \end{array}$$

Rule of Consequence

$$\frac{P' \rightarrow P \qquad \{P\}\, C\, \{Q\} \qquad Q \rightarrow Q'}{\{P'\}\, C\, \{Q'\}}$$

Modifies(C) is the set of variables occurring alone on the left-and side in statements $x := -$ in C. (We stress that *Modifies*$(x.i := E)$ is empty.)

Theorem 7 (Soundness Theorem). *The rules and axioms of this section are sound according to the partial correctness interpretation of triples.*

Proof: The first main difficulty lies in the axioms for assignment statements. For $x := E$, if $s, h \models P[E/x] \wedge E = E$ then the $E = E$ part implies that that $x := E, s, h$ does not go wrong. If $x := E, s, h \rightsquigarrow^* s', h'$ then the derivation must consist of an application of the operational rule for $x := E$ followed by several applications of the relocation and collection rules. The Substitution Lemma ensures that P satisfies the state $(s \mid x \mapsto [\![E]\!]s), h$ resulting from the $x := E$ rule. Any applications of the relocation rule preserve P by the Renaming Lemma, and any applications of collection preserve P by the Garbage Insensitivity Theorem. This shows the soundness of the axiom for $x := E$.

In a similar way, the soundness of the other axioms follows, first, from an argument about the operational rule for a given assignment statement, followed by preservation of truth by relocation and collection.

The other nontrivial case is the rule for Auxiliary Variable Elimination. Suppose that $s, h \models \exists x.P$. Then there is some $g \sqsupseteq h$ and v such that $(s \mid x \mapsto v), g \models P$. We can factor g as $h * h'$ for the restriction h' of g to those pointers in the domain of g but not h. Suppose $C, s, h \rightsquigarrow^* s'', h''$, then by renaming (lemma 3) we can assume $h' \# h''$. Since $x \notin \mathrm{Free}(C)$, $C, (s \mid x \mapsto v), h * h' \rightsquigarrow^* (s'' \mid x \mapsto v), h'' * h'$. Now, the premise $\{P\}\, C\, \{Q\}$ implies $(s'' \mid x \mapsto v), h'' * h' \models Q$, hence $s'', h'' \models \exists x.Q$.

The soundness of all of the other rules is straightforward. ■

We illustrate the use of the rules by presenting a proof outline, where intermediate assertions are used to show how to construct a proof. We will not explicitly consider definedness issues in the application of the rules for `while` or Hoare assignment.

We consider the program for reversing a linked list by reversing its links. For simplicity we will not prove the complete correctness of the program: we will just show that if you start with a list you end with a list. (This implies that the program does not go wrong, by the interpretation of triples, and is the kind of data-independent property one might consider in automatic program proving.) For this example we will assume a predicate satisfying

$$\mathit{list}(x) \;\Leftrightarrow\; x = \texttt{nil} \;\vee\; (\exists z.\, x \mapsto -, z \;*\; \mathit{list}(z))$$

Notice how the use of $\exists$ here is in line with the second Guarding Lemma.

Here is the outline.

$\{\mathit{list}\, c\}$
$\{\mathit{list}\, c \;*\; \mathit{list}\, \texttt{nil}\}$
 $p := \texttt{nil}$
$\{\mathit{list}\, c \;*\; \mathit{list}\, p\}$
 $\texttt{while}\ c \neq \texttt{nil}\ \texttt{do}$
  $\{(\mathit{list}\, c \;*\; \mathit{list}\, p) \wedge c \neq \texttt{nil}\}$
  $\{\exists t'.\, c \mapsto -, t' \;*\; \mathit{list}\, t' \;*\; \mathit{list}\, p\}$
   $t := c.2$
  $\{c \mapsto -, t \;*\; \mathit{list}\, t \;*\; \mathit{list}\, p\}$
   $c.2 := p$
  $\{c \mapsto -, p \;*\; \mathit{list}\, t \;*\; \mathit{list}\, p\}$
  $\{\mathit{list}\, t \;*\; \mathit{list}\, c\}$
   $p := c$
  $\{\mathit{list}\, t \;*\; \mathit{list}\, p\}$

$$
\begin{array}{l}
\qquad c := t \\
\qquad \{list\, c * list\, p\} \\
\{(list\, c * list\, p) \wedge c = \mathtt{nil}\} \\
\{list\, p\}
\end{array}
$$

In this proof outline two propositions that appear in succession indicate an application of the Rule of Consequence. The triple for $t := c.2$ requires several steps, involving Consequence and Auxiliary Variable Elimination. The loop invariant is $list\, c * list\, p$: we show that it is preserved in the loop, and thus get to conclude the invariant with $\neg(c \neq \mathtt{nil})$ on completion of the loop, using the usual `while` rule for partial correctness given in section 11.

In section 14 we discuss some limitations of the axiom system.

12 Partial States and Density

Focussing on partial states opens up possibilities for connecting the semantics to other, more general, forms of interpretation. In this section we establish a connection to the dense semantics of classical logic [4, 12]. This connection provides welcome theoretical support for the clauses in the total states model, since utmost care is needed whenever the semantics of quantifiers is altered.

Although $\Vdash$ was defined in terms of $\models$, it can be regarded as a semantics in its own right, in that it can be defined by using semantic clauses, without reference to $\models$. However, a pointwise characterization of the connectives does not work. To see why, consider that $s, [] \Vdash (x \mapsto 3, 4) \to false$ does not hold, while the property "$s, [] \Vdash x \mapsto 3, 4$ implies $s, [] \Vdash false$" does. Thus, for the partial state semantics we need to consider changes of world to interpret implication.

A first point to note is that all assertions satisfy the monotonicity condition from Kripke's interpretation of intuitionistic logic.

Monotonicity Condition:
$s, h \Vdash P \Longrightarrow \forall g \sqsupseteq h.\, s, g \Vdash P.$

This follows from the Growth property in the Garbage Insensitivity Theorem and the definition of $\Vdash$ in terms of $\models$.

The Monotonicity Condition is not the whole story, in that it does not account for excluded middle, garbage, or the central role of total states. These are summed up in further properties satisfied by $\Vdash$.

Totality Condition:
$s, h \Vdash P \Longleftrightarrow \forall g \sqsupseteq h.$ if s, g is total then $s, g \Vdash P$.

Partial 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:
$s, h \Vdash P \Longleftrightarrow \forall h' \sqsupseteq h.\, \exists h'' \sqsupseteq h'.\, s, h'' \Vdash P.$

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

Theorem 8 (Dense Semantics Theorem). *All propositions satisfy Totality, Partial Shrinkage and Density. Furthermore, the standard clauses of dense semantics all hold.*

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

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).*

Proof: The $\Rightarrow$ direction of Density follows at once from the definition of $\Vdash$. For the $\Leftarrow$ direction, suppose that the right-hand side holds but $s, h \not\Vdash P$. This means that there is a total state s, g extending s, h where $s, g \not\models P$. By the Growth part of the Garbage Insensitivity Theorem, it follows that $s, g' \not\models P$ for all $g' \sqsupseteq g$ where s, g' is total. This contradicts the assumed right-hand side of Density: g is a heap that has no superheap supporting P. This shows Density. Totality and Partial Shrinkage are straightforward.

To check the semantic clauses we consider implication only. The $\Rightarrow$ direction is immediate. For $\Leftarrow$, suppose the right-hand side holds and $s, h \not\Vdash P \to Q$. Then there is a larger total state s, g where $s, g \models P$ and $s, g \not\models Q$. But this contradicts the rhs, because is we pick $g \sqsupseteq h$ then we must have that $s, g \Vdash P$ implies $s, g \Vdash Q$ (note that $\Vdash$ and $\models$ agree on total states). This shows $\to$. ■

The conditions in this theorem can be taken as the basis of 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 [4, 12]. The reader may enjoy verifying $((P \to \mathit{false}) \to \mathit{false}) \to P$ assuming P is dense. Thus, because $\Vdash$ and $\models$ agree on total states, the total state semantics can be viewed as a specialization of a well known semantics.

13 The Monotone Semantics

We have seen that $\Vdash$ satisfies the monotonicity property from intuitionistic logic. It is natural to ask what happens if we take an intuitionistic semantics as primitive. In this section we map out the theoretical territory, while in the next we discuss the advantages of the monotone semantics.

The forcing relation $s, h \Vdash_i P$ for a (possibly partial) state s, h is given by the following clauses (i is for intuitionistic).

$$
\begin{array}{lll}
s, h \Vdash_i \texttt{false} & & \text{never} \\[6pt]
s, h \Vdash_i P \to Q & \overset{\Delta}{\iff} & \forall h' \sqsupseteq h \,.\, \text{if } s, h' \Vdash_i P \ \text{ then } \ s, h' \Vdash_i Q \\[6pt]
s, h \Vdash_i P \vee Q & \overset{\Delta}{\iff} & s, h \Vdash_i P \text{ or } s, h \Vdash_i Q \\[6pt]
s, h \Vdash_i P \wedge Q & \overset{\Delta}{\iff} & s, h \Vdash_i P \text{ and } s, h \Vdash_i Q \\[6pt]
s, h \Vdash_i \forall x.P & \overset{\Delta}{\iff} & \forall v \in \texttt{Values}.\,(s \mid x \mapsto v), h \Vdash_i P \\[6pt]
s, h \Vdash_i \exists x.P & \overset{\Delta}{\iff} & \exists v \in \texttt{Values}.\,(s \mid x \mapsto v), h \Vdash_i P \\[6pt]
s, h \Vdash_i P * Q & \overset{\Delta}{\iff} & \exists h_1, h_2 .\, h_1 \# h_2,\ h_1 * h_2 = h,\ s, h_1 \Vdash_i P \text{ and } s, h_2 \Vdash_i Q \\[6pt]
s, h \Vdash_i P \mathrel{-\!\!*} Q & \overset{\Delta}{\iff} & \forall h' .\, h \# h' \text{ and } s, h' \Vdash_i P \text{ implies } s, h * h' \Vdash_i Q
\end{array}
$$

In the intuitionistic semantics we are able to define the implication adjoint $\mathrel{-\!\!*}$ of $*$. The reason for not including $\mathrel{-\!\!*}$ in the dense semantics is that $P \mathrel{-\!\!*} Q$ is not necessarily dense when P and Q are. An example is $(x \mapsto 3, 4) \mathrel{-\!\!*} \texttt{false}$: it asserts that it is not possible to extend the current heap with a disjoint heap where x points to $3, 4$; in other words, x already points to some cell in the current heap. If we take the state (s_0, h_0) where s_0 maps x to a location and h_0 is the empty heap, we have that $\forall h' \sqsupseteq h_0 .\, \exists h'' \sqsupseteq h' .\, s_0, h'' \Vdash P$ holds since *any* heap h' can be extended to one which supports $(x \mapsto 3, 4) \mathrel{-\!\!*} \texttt{false}$, but $s_0, h_0 \not\Vdash P$. This contradicts the $\Leftarrow$ direction of Density.

The dense semantics can be regarded as a special case of the intuitionistic semantics, obtained by a $\neg\neg$ transform. The transform maps propositions from the language of section 4 into the $\to$, $\texttt{false}$, $\forall$ fragment of the intuitionistic language. [1]

> The transform $\overline{P}$ defines $\overline{\exists x.P}$ to be $\neg\forall x.\neg\overline{P}$, and maps and everything else (inductively) to itself.

It is standard that this transform preserves semantics.

Lemma 12 (Translation Correctness Lemma). $s, h \Vdash P \Leftrightarrow s, h \Vdash_i \overline{P}$.

So, the dense propositions correspond to a special case of the intuitionistic ones. The class of intuitionistic propositions that are garbage insensitive is, however, wider than the range of this transform. Put another way, the range of the transform has special properties – Density, Shrinkage, Totality – that do not hold for the whole intuitionistic language.

[1] note how some connectives of section 4 are defined indirectly, for example $P \vee Q$ is short for $(\neg P) \to Q$. Had $\vee$ been a primitive, we could have defined $\overline{P \vee Q}$ as $(\neg\overline{P}) \to \overline{Q}$ or $\neg\neg(\overline{P} \vee \overline{Q})$.

The most important observation is that the intuitionistic $\exists$ violates Shrinkage. Consider $\exists x.\, x \mapsto 5, 42$. This is true of the state $[], [p \mapsto \langle 5, 42\rangle]$ but false of $[], []$, where the latter is the prune. But it is true of both states in the dense semantics model; the reason is that the dense $\exists$ is essentially $\neg\forall\neg$, and so quantifies over larger heaps.

However, if we consider a guarded version of $\exists$, then Shrinkage is preserved. Specifically, if G is as in the Guarding Lemma then $\exists x.\, G \wedge P$ and $\exists x.\, G * P$ satisfy Shrinkage. The monotone semantics gives us a sharper view of the problem with quantification: in the intuitionistic setup (where $\exists$ and $\forall$ are not interdefinable) a guard is required for $\exists$, but not for $\forall$.

Intuitionistic disjunction does preserve Shrinkage, but it violates Totality. To see why, $y \mapsto 3, 4 \vee \neg(y \mapsto 3, 4)$ can be false at a state where y denotes a location not in the domain of the heap, but it is true at all total states. Thus, there can be no extension of the total state semantics $\models$ with a connective corresponding to the intuitionistic disjunction of partial state semantics.

We sum up this discussion with the following result.

Theorem 9 (Intuitionistic Garbage Insensitivity Theorem). *All propositions satisfy Monotonicity (Growth). Furthermore, suppose P is a proposition in which every existential quantifier is of the form $\exists x.\, G \wedge Q$ or $\exists x.\, G * R$ for G of the form $y \mapsto x, E$ or $y \mapsto E, x$. Then P satisfies Partial Shrinkage.*

Proof: Both Monotonicity and Partial Shrinkage are by induction on the structure of the proposition. The only interesting cases are Partial Shrinkage for $\forall$ and $-\!\!*$.

For $\forall$, suppose $s, h \Vdash_i \forall x.\ P$ and $h = h' * h''$ and that the pointers in $dom(h'')$ are unreachable in s, h'. We have to show $s, h' \Vdash_i \forall x.\ P$. Take any value v; by hypothesis we have $(s \mid x \mapsto v), h' * h'' \Vdash_i P$. Modulo renaming (lemma 3) we can assume w.l.o.g. that the pointers in $dom(h'')$ are unreachable in $(s \mid x \mapsto v), h'$. Then $(s \mid x \mapsto v), h' \Vdash_i P$ by Partial Shrinkage of P. This proves Partial Shrinkage for $\forall$.

For $-\!\!*$, suppose $s, h \Vdash_i P -\!\!* Q$ and $h = h' * h''$ and that the pointers in $dom(h'')$ are unreachable in s, h'. We have to show $s, h' \Vdash_i P -\!\!* Q$. Take any g' such that $g' \# h'$ and $s, g' \Vdash_i P$. Modulo renaming (lemma 3) we can assume w.l.o.g. that $g' \# h$ and that the pointers in $dom(h'')$ are unreachable in $s, h' * g'$. Then $s, h * g' \Vdash_i Q$ holds by the hypothesis $s, h \Vdash_i P -\!\!* Q$, hence $s, h' * g' \Vdash_i Q$ by Partial Shrinkage of Q. This proves Partial Shrinkage for $-\!\!*$. ■

14 Program Logic, II

We now describe two ways in which the intuitionistic semantics can be used to obtain an improved program logic. The treatment here is just a sketch; we refer to [9, 14, 17] for more information on the low-level viewpoint adopted in this section.

The first improvement is to use $-\!\!*$ to express weakest preconditions for allocation and heap update [9].

$$\{\exists xy.\, (E \mapsto x, y) * ((E \mapsto E', y) \mathrel{-\!\!*} P)\}$$
$$E.1 := E'$$
$$\{P\}$$

$$\{\forall x'.\, (x' \mapsto E_1, E_2) \mathrel{-\!\!*} P[x'/x]\}$$
$$x := \mathtt{cons}(E_1, E_2)$$
$$\{P\}$$

In these axioms we require that x and y are not free in E, E', or P and that x' is not free in E_1, E_2 or P. Notice how it is natural to quantify over unreachable pointers in the precondition for $x := \mathtt{cons}(E_1, E_2)$.

A more important limitation of the total semantics is the unsoundness of the Frame Rule ([9, 14]):

Frame Rule:

$$\frac{\{P\}\, C\, \{Q\}}{\{P * R\}\, C\, \{Q * R\}}\ \mathit{Modifies}(C) \cap \mathrm{Free}(R) = \{\}$$

This rule lets one reason locally about the portion of storage accessed by programs, automatically inferring that the other areas of storage are left unchanged.

For a counter example to the soundness of the Frame Rule, consider the following true triple:

$$\{\mathit{true}\}\ \mathtt{if}\ \ x == \mathit{nil}\ \ \mathtt{then\ skip\ else}\ \ x.1 := 5\, \{\mathit{true}\}$$

Applying the Frame Rule with invariant $x \mapsto 3, 4$ we get the false triple

$$\{x \mapsto 3, 4\}\ \mathtt{if}\ \ x == \mathit{nil}\ \ \mathtt{then\ skip\ else}\ \ x.1 := 5\, \{x \mapsto 3, 4\}$$

The Frame Rule is unsound because of total states: checking that x is not nil in a total state is enough to guarantee safety of access to its components. In other words, given a true triple, it is possible that the command accesses heap cells not mentioned in the precondition.

The Frame Rule is not so important in *intraprocedural* program proving, when there are no procedure calls; then the program logic from section 11 can be used, perhaps with the addition of $\mathrel{-\!\!*}$. But when it comes to *interprocedural* proving, framing is essential to keep procedure specifications manageable; see [14].

The Frame Rule can be saved if we adopt a lower-level interpretation of triples. Thus far we have used the operational semantics of commands on total states only, but the definition applies equally to partial states; the difference is that commands go wrong more often.

The only alteration to the previous interpretation of triples that we need is to allow partial states.

PARTIAL CORRECTNESS
$\{P\}\, C\, \{Q\}$ is true just if for all (possibly partial) states s, h with $\mathit{dom}(s) \supseteq \mathrm{Free}(P, C, Q)$, if $s, h \Vdash_i P$ then
- C, s, h doesn't go wrong, and

– if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \Vdash_i Q$.

According to this interpretation,

$$\{true\}\ \texttt{if}\ x == nil\ \texttt{then skip else}\ x.1 := 5\ \{true\}$$

is no longer a true triple, because the command goes wrong when applied to the state $[x \mapsto p], []$ for some pointer p. We refer to [14, 17] for more information on the low-level interpretation of triples and the Frame Rule.

15 Final Remarks

In this paper we have presented a number of semantic models, and two interpretations of Hoare triples. The reason for this variety is mainly presentational. The most satisfactory program logic, based on the intuitionistic interpretation of assertions and low-level interpretation of triples in section 13, is not the first that comes to mind. And the argument for it is not immediate. So, we felt it worthwhile to first set out the total states model, which is partially satisfactory, and point out its limitations. Curiously, we found that a lower level of abstraction, which is incompatible with full abstraction, led to a smoother program logic. The best logic is based on the intuitionistic semantics of [16, 9], with a constraint placed on the use of the existential quantifier.

We have investigated the idea of garbage insensitivity in this paper using several interpretations of the logical connectives, but it is the property of garage insensitivity itself, rather than these non-standard interpretations, that is of most interest. It is worth noting that the intuitionistic semantics can be embedded in a still lower level, pointwise model [9], which has a standard semantics of quantifiers and which validates the law of the excluded middle. The pointwise semantics $\Vdash_p$ replaces the intuitionistic clause for implication with

$$s, h \Vdash_p P \to Q \overset{\Delta}{\iff} \text{if } s, h \Vdash_p P \text{ then } s, h \Vdash_p Q.$$

The intuitionistic language can be mapped into the pointwise language using a modal transform. If we observe that

$$s, h \Vdash_p \texttt{true} \mathrel{-\!\!*} P \ \text{ iff } \forall h' \sqsupseteq h.\, s, h' \Vdash_p P$$

holds, then we can regard $\Box P \overset{\Delta}{=} \texttt{true} \mathrel{-\!\!*} P$ as the S4 necessity modality for heap extension. The modal translation $(\cdot)^\circ$ sends $P \to Q$ to $\Box(P^\circ \to Q^\circ)$ and everything else (inductively) to itself. In the pointwise semantics one can describe exact properties of the heap, which violate monotonicity, such as "there are exactly three active cells".

Our operational semantics treated the garbage collector as a black box. But if one was to verify a runtime system, which includes both a compiled user program and a collector, then it would be desirable to be able to state both garbage sensitive and insensitive properties. For example, in applications of proof-carrying code [13, 1] a compiler automatically produces a proof of safety of a compiled program and, from an abstract perspective, one might expect such properties to be garbage insensitive (and perhaps even dense) if we start with a memory safe source language. On the other hand, to verify the collector itself, we would typically want to consider lower level properties. So, if we were to try to account for the interaction of a user program with a collector, it would probably be better to work with the pointwise semantics, and simply keep an eye out for propositions that are garbage insensitive. From this perspective the results here can be viewed as showing some of the structure that these propositions possess.

References

[1] A. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 247–256. IEEE Computer Society Press, 2001.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] C.A.R. Hoare and J. He. A trace model for pointers and objects. In Rachid Guerraoui, editor, *ECCOP'99 - Object-Oriented Programming, 13th European Conference*, pages 1–17, 1999. LNCS. 1628, Springer.

[8] 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.

[9] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programmoing Languages (POPL)*, pages 14–26. ACM Press, jan 2001.

[10] R. Jones. *Garbage Collection: Algorighme for Automatic Dynamic Memory Management.* John Wiley and Sons, 1997.

[11] S. A. Kripke. Semantical analysis of intuitionistic logic I. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland, Amsterdam, 1965.

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

[13] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, 1998. SIGPLAN Notices 33(5).

[14] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume ??? of

Lecture Notes in Computer Science, pages ??–??, Paris, France, September 2001. Springer-Verlag, Berlin.

[15] 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.

[16] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

[17] H. Yang. *Local Reasoning for Stateful Programs.* Ph.D. thesis, Univ of Illinois at Urbana-Champaign, 2001.