A Semantic Basis for Local Reasoning

Hongseok Yang[1] and Peter O'Hearn[2]

[1] ROPAS, KAIST
[2] Queen Mary, University of London

Abstract. We present a semantic analysis of a recently proposed formalism for local reasoning, where a specification (and hence proof) can concentrate on only those cells that a program accesses. Our main results are the soundness and, in a sense, completeness of a rule that allows frame axioms, which describe invariant properties of portions of heap memory, to be inferred automatically; thus, these axioms can be avoided when writing specifications.

1 Introduction

The need to say what memory cells or other resources are not changed, along with those that are, has always been a vexing problem in program specification.

Consider a specification of a program to copy a tree:

$$\{\texttt{tree}\,\tau\,p\}\ \texttt{CopyTree}(p;q)\ \{(\texttt{tree}\,\tau\,p) * (\texttt{tree}\,\tau\,q)\}.$$

Here, the parameter list indicates that p is a value and q a reference parameter. The predicate $\texttt{tree}\,\tau\,p$ says that p is, or points to, a data structure representing a binary tree τ, and (anticipating the work to come) $(\texttt{tree}\,\tau\,p)*(\texttt{tree}\,\tau\,q)$ says that p and q both represent this tree, but that their representations utilize disjoint storage.

This specification certainly captures part of what we intend to say about the procedure. But a Hoare triple typically only describes the effects an action has on the portion of program store it explicitly mentions; it does not say what cells among those not mentioned remain unchanged. As a result, the specification of $\texttt{CopyTree}(p;q)$ leaves open the possibility that the procedure alters a cell not in the data structure described in the precondition. To make a stronger specification we need to, in one way or another, take into account the notorious "frame axioms" [8], which describe cells that remain unchanged.

It might seem that this problem is just a nuisance, that we should be content for practical purposes to prove weak properties and not worry about frame axioms. This viewpoint is untenable, for the following reason. At a call site for $\texttt{CopyTree}(p;q)$ there will often be more cells active than those in p's data structure. In that case the specification is not strong enough to use, unless we somehow take the frame axioms into account. A particular example of such a call site is in the body of a recursive definition of $\texttt{CopyTree}(p;q)$, which uses recursive calls for each of two subtrees. As explained in [5], if we do not have some way of showing that each recursive call doesn't affect the other, then the specification will not be strong enough to use as an induction hypothesis when proving the program.

It might alternatively be thought that the problem can be easily solved, simply by listing the variables that a program might alter as part of the specification. This viewpoint is untenable when there are storage cells other than those directly named by variables, typically when there are pointers of one form or another. This solution is thus not applicable to realistic imperative languages.

Nonetheless, although much more than a nuisance, this frame problem is still irritating. It seems unfortunate to have to think up a general formula to function as a description of the cells left unchanged, covering all potential call sites, whenever we write a specification. And intuitively the specification of $\texttt{CopyTree}(p; q)$, for instance, seems to already carry enough information. The hitch is that for this intuition to be realized we need to somehow require that any state alteration not explicitly mandated by the specification is excluded.

Unfortunately, this last part, that "any state alteration not explicitly mandated by the specification is excluded" is difficult to make precise, and that difficulty has spawned hundreds of papers in AI and in program specification. Twice unfortunately, no completely convincing solution has emerged.

In this paper we study the semantics of an approach recently developed in an extension of Hoare's logic for reasoning about mutable data structures [5]. The scope of the approach is modest in intent, in that it is not put forward as a general solution to the frame problem. Rather, the suggestion is that, when certain assumptions are met, there is a natural and simple way to avoid frame axioms. Although, as we further demonstrate here, these assumptions are met in some natural models of imperative languages, there is no claim that they are universally applicable in reasoning about action.

The general idea is that by focusing on the idea of the memory footprint of a program we can get a concrete handle on the resources that a specification of a program needs to describe. More specifically, there are two components to the approach.

1. We interpret a specification $\{P\}\, C\, \{Q\}$ so that, when C is run in a state satisfying P, it must dereference only those cells guaranteed to exist by P or allocated during execution.
2. An inference rule, the Frame Rule, lets us obtain $\{P * R\}\, C\, \{Q * R\}$ from the initial specification $\{P\}\, C\, \{Q\}$ of a procedure or command, where $P * R$ is true just when P and R are true of separate areas of the current heap memory.

Point 1 is reminiscent of the old informal idea of a tight interpretation of specifications: we assume that a specification mentions all the resources relevant to understanding a program, that other resources are automatically unaffected. With it the specification of the `CopyTree` procedure above implies that any active cell not in p's tree before execution will remain unchanged; this allows us to avoid explicit frame axioms and instead work with the simple specification. Point

2 then gives us a proof method that enables us to infer the invariant properties supported by tightness.

We stress that the first of these points does not depend on the second. Given the tight interpretation, a host of invariant properties are simply true, and this is independent of the language used to describe pre and postconditions. The $*$ connective just gives us a direct way to exploit tightness, in a program logic.

The purpose of this paper is to provide a semantic analysis of these ideas. The basic conception of the interplay between points 1 and 2 above as a basis for local reasoning is due to the second author. Many of the semantic foundations, particularly those related to completeness described later, were first worked out thoroughly by the first author. We refer to the precursor papers [9, 2, 5] for examples of reasoning with the spatial formalism, for program logic axioms for specific heap-altering and accessing commands, and for references to the literature on program logic and on the frame problem (see also [8]).

The first problem we tackle is the soundness of the Frame Rule. This turns out to be surprisingly delicate, and it is not difficult to find situations where the rule doesn't work. So a careful treatment of soundness, appealing to the semantics of a specific language, is essential. We phrase our argument here in terms of a model devised by Reynolds [10, 5], for an extension of the language of `while` programs with operations for pointer manipulation, including address arithmetic.

In the course of proving soundness we will attempt to isolate the properties on which it relies. Once these properties are established, much of the work in this paper can be carried out at a more abstract level and we sketch the appropriate definitions. But a thorough abstract account will be left for other work; here our goal is to remain concrete and provide a detailed analysis of a single language.

After showing soundness we move on to prove a completeness result, which shows a sense in which no frame axioms are missing. Our approach to completeness follows a line of work which goes under the name of sharpness or adaptation completeness [4]. There one of the main issues is always to conclude certain invariant properties of variables not free in a command. Our completeness result can be seen as extending such results to handle the heap as well, where there are other store locations than those associated with program variables. The key idea for proving completeness is to view commands as predicate transformers satisfying a locality property.

2 A Programming Language

The programming language is an extension of the language of `while` programs, with operations for manipulating pointers in a heap. The syntax and domains for the language are in Table 1.

The model has two components, the store and the heap. The store is a mapping from variables to integers, and the heap is a finite mapping from natural numbers (addresses) to integers. The heap is accessed using indirect addressing $[E]$ where E is an arithmetic expression.

SYNTAX

$$
\begin{array}{lll}
C & ::= & x := E \mid x := [E] \mid [E] := F \mid x := \mathtt{cons}(E_1, ..., E_n) \mid \mathtt{dispose}(E) \\
 & \mid & C; C \mid \mathtt{while}\ B\ C \mid \mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C \\
E, F & ::= & x, y, ... \mid 0 \mid 1 \mid E + F \mid E \times F \mid E - F \\
B & ::= & \mathtt{false} \mid B \Rightarrow B \mid E = F \mid E < F
\end{array}
$$

DOMAINS

$$
\begin{array}{ll}
\mathtt{Nats} \triangleq \{0, 1, ..., 17, ...\} & \mathtt{Ints} \triangleq \{..., -17, ..., -1, 0, 1, ..., 17, ...\} \\
\mathtt{Variables} \triangleq \{x, y, ...\} & \mathtt{Stores} \triangleq \mathtt{Variables} \to \mathtt{Ints} \\
\mathtt{Heaps} \triangleq \mathtt{Nats} \rightharpoonup_{fin} \mathtt{Ints} & \mathtt{States} \triangleq \mathtt{Stores} \times \mathtt{Heaps}
\end{array}
$$

FUNCTIONALITY OF EXPRESSIONS

$$
[\![E]\!]s \in \mathtt{Ints} \qquad [\![B]\!]s \in \{true, false\} \qquad (\text{where } s \in \mathtt{Stores})
$$

Table 1. Syntax and Domains

We assume the standard denotational semantics of integer and boolean expressions. Note that expressions are heap-independent.

The crucial operation on heaps is disjoint combination. We write $h \# h'$ to indicate that the domains $dom(h)$ and $dom(h')$ are disjoint. When $h \# h'$ holds, $h * h'$ is the heap obtained by taking the union of disjoint partial functions. When $h \# h'$ does not hold, $h * h'$ is undefined. The empty heap $[]$ is the unit of $*$. The notation $[n \mapsto m]$ describes the singleton heap which maps n to m and which is undefined everywhere else.

The operational semantics of commands defines a relation $\rightsquigarrow$ on configurations. Configurations include terminal configurations s, h, triples C, s, h, and a special configuration *fault* indicating a memory fault. The command $x := [E]$ reads the value at address E in the heap and places it in x. $[E] := F$ updates address E so that its content is F. $x := \mathtt{cons}(E_1, ..., E_n)$ allocates a sequence of n contiguous heap cells, initializes them to $E_1, ..., E_n$, and places the address of the first cell in the segment in x. $\mathtt{dispose}(E)$ removes address E from the heap. The commands $x := [E]$, $[E] := F$ and $\mathtt{dispose}(E)$ generate a memory fault, a particular kind of error, if E is not an active address. Notice that the number m is chosen non-deterministically in the rule for cons. An example of a command that always faults is $\mathtt{dispose}(x)$; $[x] := 42$. In typical implementations an attempt to dereference a disposed address might not always lead immediately to a fault. Generating these faults early in the semantics is a device that allows us to

$$\frac{}{x := E, s, h \rightsquigarrow (s \mid x \mapsto [\![E]\!]s), h}$$

$$\frac{[\![E]\!]s = n}{\mathtt{dispose}(E), s, h * [n \mapsto m] \rightsquigarrow s, h} \qquad \frac{[\![E]\!]s = n \quad n \notin dom(h)}{\mathtt{dispose}(E), s, h \rightsquigarrow \mathit{fault}}$$

$$\frac{[\![E]\!]s = n \in dom(h) \quad h(n) = m}{x := [E], s, h \rightsquigarrow (s \mid x \mapsto m), h} \qquad \frac{[\![E]\!]s \notin dom(h)}{x := [E], s, h \rightsquigarrow \mathit{fault}}$$

$$\frac{[\![E]\!]s = n \in dom(h)}{[E] := F, s, h \rightsquigarrow s, (h \mid n \mapsto [\![F]\!]s)} \qquad \frac{[\![E]\!]s \notin dom(h)}{[E] := F, s, h \rightsquigarrow \mathit{fault}}$$

$$\frac{m, ..., n+m-1 \notin dom(h) \quad v_1 = [\![E_1]\!]s, ..., v_n = [\![E_n]\!]s}{x := \mathtt{cons}(E_1, ..., E_n), s, h \rightsquigarrow (s \mid x \mapsto m), (h * [m \mapsto v_1, ..., n+m-1 \mapsto v_n])}$$

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

$$\frac{[\![B]\!]s = true}{\mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C', s, h \rightsquigarrow C, s, h} \qquad \frac{[\![B]\!]s = false}{\mathtt{if}\ B\ \mathtt{then}\ C\ \mathtt{else}\ C', s, h \rightsquigarrow C', s, h}$$

$$\frac{[\![B]\!]s = false}{\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od}, s, h \rightsquigarrow s, h} \qquad \frac{[\![B]\!]s = true}{\mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od}, s, h \rightsquigarrow (C; \mathtt{while}\ B\ \mathtt{do}\ C\ \mathtt{od}), s, h}$$

Table 2. The Programming Language: Syntax and Semantics

arrange the formalism in a conservative manner, where well-specified programs will never try to dereference a disposed address.

In the semantics we use $(f \mid i \mapsto j)$ for the (perhaps partial) function like f except that i goes to j. This notation is used both when i is and is not in the domain of f.

3 Specifications

We treat predicates semantically in this paper, so a predicate is just a subset of the set of states.

$$\mathtt{Pred} \stackrel{\Delta}{=} \mathcal{P}(\mathtt{States})$$

To evoke the semantics of the assertion languages from [9, 2, 5], we sometimes use the satisfaction notation

$$s, h \models p$$

as an alternative to $(s, h) \in p$.

To define the semantics of Hoare triples first recall point 1 from the Introduction, where we guarantee that if $\{p\}C\{q\}$ holds then C must not access any cells not guaranteed to exist by p. We can formalize this by observing that if C did guarantee to access such a cell, then it could be made to fault by running it in a state in which that cell is not active. This is the role of the following notion of safety.

– "C, s, h is safe" when $C, s, h \not\rightsquigarrow^* \mathit{fault}$.

For partial correctness the interpretation of triples is the standard one, with an additional safety requirement.

Partial Correctness. $\{p\}C\{q\}$ is true just when, for all s, h,
if $s, h \models p$ then
– C, s, h is safe, and
– if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models q$.

This fault-avoiding interpretation of triples is not new, but the connection to the intuitive notion of tight specification does not seem to have been observed before (excepting [2, 5]). To describe this, suppose $E \hookrightarrow F$ is a predicate saying that F is the contents of the active address E in the current heap. Suppose, further, that the triple

$$\{x \hookrightarrow 5\}C\{x \hookrightarrow 6\}$$

holds and that C does not alter any variables (though it might alter heap cells). Then we claim that, just from this information, we can infer that C does not modify any heap cell existing in the starting state, other than the one denoted by x. To see the reason, suppose that C does modify such a cell and call it y. Then the specification says that the program will not fault if it is run in the singleton state where x is the only active address (with contents 5). But we just said that C alters the address y, and so this attempt to dereference y must generate a fault starting from the singleton state.

We will also consider a total correctness form of specification. For total correctness, we do not need an explicit safety assumption, because total correctness is about "must termination" which itself includes a safety requirement.

– "C, s, h must terminate normally" when C, s, h is safe and there is no infinite $\rightsquigarrow$-sequence starting from C, s, h.

Total Correctness. $\{p\}C\{q\}$ is true just when, for all s, h
if $s, h \models p$ then
– C, s, h must terminate normally, and
– if $C, s, h \rightsquigarrow^* s', h'$ then $s', h' \models q$.

To formulate the Frame Rule we will need the $*$ connective; if p and q are predicates then

$$p * q \stackrel{\Delta}{=} \{(s, h * h') \mid s, h \models p \wedge s, h' \models q \wedge h \# h'\}$$

The statement of the Frame Rule from [2, 5] is as follows.

Frame Rule, Syntactic Version

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

where Modifies(C) denotes the set of variables updated in the command C, i.e., those appearing as the destination of an assignment statement in C. To be precise, the Modifies set of each of $x := \cdots$ is $\{x\}$, while for $[E] := F$ and $\texttt{dispose}(E)$ it is empty; these latter two statements affect the heap but not the values of variables in the store. Since we are working semantically with predicates in this paper, we need to reformulate the rule slightly and replace the reference to the free variables of R in the side condition. We will precede this with a short discussion.

The condition for variables in the rule is straightforward; it simply checks whether any variables in R are modified by a command C. However, the condition for heap cells is more elaborate. With the spatial conjunction, it says that for every state satisfying $P * R$, the current heap can be split into two subheaps so that P holds for the one and R for the other; then, the tight interpretation of the Hoare triple $\{P\}\,C\,\{Q\}$ says that the command can only access the first part, i.e. the part for P, consequently making R an invariant during execution.

As an example, starting from the specification of CopyTree in the Introduction we can infer that copying p's tree does not affect a cell not in it's data structure.

$$\frac{\big\{\texttt{tree}\,\tau\,p\big\}\ \texttt{CopyTree}(p;q)\ \big\{(\texttt{tree}\,\tau\,p) * (\texttt{tree}\,\tau\,q)\big\}}{\big\{(\texttt{tree}\,\tau\,p) * (x \hookrightarrow y)\big\}\ \texttt{CopyTree}(p;q)\ \big\{(\texttt{tree}\,\tau\,p) * (\texttt{tree}\,\tau\,q) * (x \hookrightarrow y)\big\}}$$

Here, the Modifies set of $\texttt{CopyTree}(p;q)$ is assumed to be $\{q\}$.

To describe a version of the rule which refers to semantic rather than syntactic predicates we utilize a notion $X\#p$ of independence of a predicate from a set of variables. This can be formulated simply in terms of quantification. If X is a set of variables then

$$\forall X.p \stackrel{\Delta}{=} \{(s,h) \mid \forall s_X \in [X \to \texttt{Ints}].\,(s[s_X],h) \in p\}$$

where $s[s']$ denotes the update of s by s' defined by:

$$s[s'](y) \;\stackrel{\Delta}{=} \begin{cases} s'(y) & \text{if } y \in \mathit{dom}(s') \\ s(y) & \text{otherwise} \end{cases}$$

Then

- $X\#p$ holds just if $p = \forall X.p$.

Frame Rule, Semantic Version

$$\frac{\{p\}\,C\,\{q\}}{\{p * r\}\,C\,\{q * r\}}\ \text{Modifies}(C)\#r$$

4 Soundness of the Frame Rule

The Frame Rule codifies a notion of local behaviour, and in this section we undertake to describe that notion in terms of the operational semantics.

It will be helpful to first consider a plausible property that does not hold.

If $C, s, h_0 \rightsquigarrow^* s', h'_0$ and $h_0 \# h_1$, then $C, s, h_0 * h_1 \rightsquigarrow^* s', h'_0 * h_1$.

The intuition behind this property is just that we can add on extra state, and any execution that works for a smaller state can still go ahead. This property fails because of the behaviour of `cons`. An address that is allocated during an execution from a small state cannot be allocated starting in a bigger state where it is already active. For example,

$$x := \mathtt{cons}(2,3), [x \mapsto m], [\,] \rightsquigarrow [x \mapsto 0], [0 \mapsto 2, 1 \mapsto 3]$$

but if we run $x := \mathtt{cons}(2,3)$ in a heap where 0 is already active, then a different address than 0 must be chosen by $\rightsquigarrow$.

This is an example of where an action on a little state can be disabled when moving to bigger states. Such behaviour might make us doubt that the Frame Rule could be sound at all. However, it only indicates that the language does not behave locally with respect to "may" properties. With Hoare triples we are interested in a form of "must" property. If $\{p\}C\{q\}$ holds and, starting from a state satisfying p, the command terminates, then the final state must satisfy q. Put another way, it is not individual computations that we will judge local, but properties of classes of computations.

The correct property says that if a command is safe in a given state, then the result of executing it in a larger state can be tracked to *some* execution computation on the little state.

Lemma 1 (Safety and Termination Monotonicity).

1. *If C, s, h is safe and $h \# h'$, then $C, s, h * h'$ is safe.*
2. *If C, s, h must terminate normally and $h \# h'$, then $C, s, h * h'$ must terminate normally.*

Lemma 2 (Frame Property). *Suppose*

$$C, s, h_0 \text{ is safe, and } C, s, h_0 * h_1 \rightsquigarrow^* s', h'.$$

Then there is h'_0 where

$$C, s, h_0 \rightsquigarrow^* s', h'_0, \text{ and } h' = h'_0 * h_1.$$

Proof. For $x := \mathtt{cons}(E_1, ..., E_n)$, consider $m, ..., n+m-1 \notin dom(h_0 * h_1)$, The operational rule gives us $s' = (s \mid x \mapsto m)$ and $h' = h_0 * h_1 * [m \mapsto v_1, ..., m+n-1 \mapsto v_n]$. Then since $m, ..., m+n-1 \notin dom(h_0)$, this segment may be selected by the operational rule for `cons` applied in the smaller heap h_0. So $h'_0 = h_0 * [m \mapsto v_1, ..., m+n-1 \mapsto v_n]$ gives us the desired result.

For $[E] := F$, since $[E] := F, s, h_0$ is safe by assumption we know $[\![E]\!]s \in dom(h_0)$ and therefore $[\![E]\!]s \notin dom(h_1)$. Thus, the assignment leaves h_1 unchanged, and taking $h'_0 = (h_0 \mid [\![E]\!]s \mapsto [\![F]\!]s)$ gives the result.

For $\mathtt{dispose}(E)$, if $[\![E]\!]s = n$ then $n \in dom(h_0)$ by safety, and h_0 decomposes as $h'_0 * [n \mapsto m]$ for some m. This h'_0 satisfies the requirement of the theorem.

For $x := [E]$, safety ensures that $[\![E]\!]s \in dom(h_0)$, and taking $h'_0 = h_0$ gives the result.

To cover $C; C'$, `while` and `if`, we need to prove a slightly stronger result. We state the property, which is needed to get the right induction hypothesis, but omit the detailed proof. Consider a variant of the theorem which considers non-terminal configurations; specifically, where the terminal configurations s', h' and s', h'_0 are replaced by C', s', h' and C', s', h'_0. The terminal and non-terminal variants are proven simultaneously, by induction on the derivation of $C, s, h_0 * h_1 \rightsquigarrow^* s', h'$ or $C, s, h_0 * h_1 \rightsquigarrow^* C', s', h'$. The rules for atomic commands above were already considered above, and each of the rules for `while` and $C; C'$ has an immediate proof. □

Theorem 1 (Soundness). *The* Frame Rule *is sound for both partial and total correctness.*

Proof. The proof uses the Frame Property, and the following locality property for variables.

If $C, s, h \rightsquigarrow^* s', h'$ and a variable x is not assigned in C, then $s(x) = s'(x)$.

For partial correctness, suppose the premise of the Frame Rule holds, and that $s, h_0 \models p$ and $s, h_1 \models r$. The premise gives us that C, s, h_0 is safe, and safety of $C, s, h_0 * h_1$ follows from Safety Monotonicity. If $C, s, h_0 * h_1 \rightsquigarrow^* s', h$, the Frame Property yields h'_0 where $h = h'_0 * h_1$ and $C, s, h_0 \rightsquigarrow^* s', h'_0$. The premise then ensures $s', h'_0 \models q$. The variable locality property implies that s' agrees with s on all variables not in Modifies(C), so $s', h_1 \models r$ follows since we know $s, h_1 \models r$ and Modifies$(C)\#r$. The semantics of $*$ then yields $s', h \models q * r$.

The argument for total correctness appeals additionally to Termination Monotonicity. □

4.1 The Scope and Delicacy of the Frame Rule

Some remarks are in order on the delicacy of the soundness result.

First, the non-deterministic nature of `cons` was relied on in an essential way. If we had interpreted `cons` so that, say, the smallest possible free address was always chosen for allocation, then adding memory would change what this new address was, and the difference could be detected with address arithmetic; this would invalidate the Frame Rule. Non-deterministic allocation is used to force a program proof not to depend on details of how the allocator might work. (In a language without address arithmetic, we could use invariance under location renaming rather than non-determinism in allocation to ensure this sort of independence.)

Second, suppose we were to add an operation for trapping a memory fault to our language. If we did this, without changing $*$, then the Frame Rule would be invalid; the reason is that we could branch on whether or not a cell is active, and this would contradict Safety Monotonicity. This does not necessarily mean that fault trapping is incompatible with the Frame Rule. We could perhaps change the interpretation of $*$ so that the undefinedness it introduces is regarded as introducing the possibility of a further kind of error, different from memory fault. (The way Calcagno puts it, we need a notion that is detectable in the program logic, but not in the programming language.)

Although delicate, the scope of the Frame Rule is wider than the specific programming language considered here. We briefly sketch an abstract setting. Suppose we have an arbitrary partial commutative monoid (pcm), in place of $(\mathtt{Heaps}, *, [])$, and an arbitrary set V of values, which we use to define $\mathtt{Stores} = \mathtt{Variables} \to V$. Then a "local action" is a binary relation between $\mathtt{States}$ and $\mathtt{States} \cup \{\bot, \mathit{fault}\}$ satisfying Safety and Termination Monotonicity, and the Frame Property. With this definition, the Frame Rule is sound for every local action.

An example somewhat removed from heap storage that fits this definition is given by Petri nets. A net without capacity $\mathcal{N} = (P, T, \mathit{pre}, \mathit{post})$ consists of sets P and T of places and transitions, and two functions $\mathit{pre}, \mathit{post}\colon T \to \mathcal{M}$ from transitions to markings, where a marking is a finite multiset of places and $\mathcal{M}$ denotes the set of all markings. $\mathcal{M}$ forms a pcm whose commutative monoid structure comes from the multiset union and the empty set. This is a total commutative monoid. If we regard a transition that is not enabled as equivalent to faulting, then each transition $t \in T$ determines a local action[1] by the firing rule: $M\,[t]\,N$ iff $\exists t. \exists M'.\, M = \mathit{pre}(t) * M'$ and $N = \mathit{post}(t) * M'$.

Nets with capacity 1 provide a counterexample. Suppose that places in a net can hold at most one token; consequently, markings are simply subsets of places, and $\mathit{pre}, \mathit{post}$ map transitions to finite subsets of places. Transitions are fired in this case only when it is guaranteed that all the tokens to be produced do not violate the capacity requirement. We can define a pcm on the set of places, where $*$ is union of disjoint sets (which is undefined on sets that overlap). A transition t that violates Safety Monotonicity has $\mathit{pre}(t) = \{a\}$, $\mathit{post}(t) = \{b\}$, where $a \neq b$. t is enabled in $\{a\}$ but not in $\{a, b\}$ because the post-place of t is already filled in $\{a, b\}$. (It is possible to define a different pcm, for which transitions do correspond to local actions, by recording when a place is known to be unmarked; we can do this using $P \rightharpoonup \{\mathit{full}, \mathit{empty}\}$, with $*$ as union of partial functions with disjoint domains.)

These examples and counterexamples indicate that the Frame Rule is not something we expect to be automatically valid, in the way that we expect, say, the rule of Consequence always to be. Close attention must be paid to the interplay between the definition of $*$ and the kinds of operation present in a language.

5 Completeness of the Frame Rule

Suppose we are given a Hoare triple specification $\{p\}C\{q\}$ but we are not told exactly what C is. The question we are concerned with in this section is whether we can derive all other specifications that follow from it, without making use of knowledge of C.

To formulate this we consider Hoare triples $\{p\} - \{q\}$ with an unspecified command. Following [3, 12, 1], we call such a Hoare triple with a hole a *specifi-*

[1] We take $\mathtt{Stores}$ to be the singleton set $[\mathtt{Variables} \to \{1\}]$.

CONSEQUENCE $$\frac{p' \subseteq p \quad \{p\} - \{q\} \quad q \subseteq q'}{\{p'\} - \{q'\}}$$	FRAME RULE $$\frac{\{p\} - \{q\}}{\{p * r\} - \{q * r\}} X\#r$$

Table 3. Proof system for specification statements with Modifies set X

cation statement. Throughout this section we assume a given set X of variables, regarded as the Modifies set. To derive one specification statement from the other we use the usual rule of Consequence from Hoare logic, and the Frame Rule; see Table 3.

The completeness question, which is variously called sharpness or adaptation completeness, is whether this system lets us derive one specification statement from another just when this inference holds semantically. Since the rule of Consequence is itself treated semantically, in that it uses inclusion between predicates-as-sets rather than a provable implication, all of the stress in this question is placed on the Frame Rule: it is essentially asking whether we obtain enough frame axioms.

To show completeness we need to define a notion of semantic consequence between specification statements. There is a niggling problem here: Not all pre/post pairs determine a relation in a direct way. The traditional way around this problem in work on specification statements is to use predicate transformers, which correspond more directly to pre/post pairs. One then separately singles out special kinds of transformers and pre/post pairs that have a good correspondence with relations. In the remainder of the paper we just work with transformers. The tie-up with relations is possible, but omitted for lack of space; we refer the reader to Yang's thesis for further information [11].

5.1 Local Predicate Transformers

In a predicate transformer interpretation, a command C is interpreted as a mapping from a postcondition to a precondition. For instance, the predicate transformer induced by $x := 2$ maps $y = x$ to $y = 2$.

Mathematically, predicate transformers are monotone maps from predicates to predicates.

$$\mathtt{PT} \stackrel{\Delta}{=} \mathtt{Pred} \rightarrow_{monotone} \mathtt{Pred}$$

Given a predicate transformer t the Hoare triple "$\{p\}\, t\, \{q\}$" corresponds to the property $p \subseteq t(q)$. Then, the monotonicity requirement is equivalent to saying that the rule of Consequence is valid.

The domain PT contains predicate transformers which, when viewed operationally, don't exhibit the local behavior of commands as described in Section 4. For instance, the predicate transformer $\lambda q. \{(s, h) \in q \mid |dom(h)| \leq 3\}$ corresponds to a command which generates a memory fault when there are more than 3 active heap cells and skips otherwise;[2] this command doesn't satisfy Safety Monotonicity. So, it is not surprising that the Frame Rule is not valid for PT since the rule is closely related to the locality properties of Section 4.

We refine PT by requiring predicate transformers to satisfy a locality condition. Recall that in this section we are assuming a given set X of modifiable variables.

$$\text{Locality for } X\text{: } \forall r, q \in \texttt{Pred}.\ X\#r \Longrightarrow t(q) * r \subseteq t(q * r)$$

The condition is equivalent to saying that t satisfies the Frame Rule, which in predicate transformer terms is

$$\forall p, r, q \in \texttt{Pred}.\ X\#r \wedge p \subseteq t(q) \Longrightarrow p * r \subseteq t(q * r)$$

The domain $\texttt{LPT}(X)$ of *local predicate transformers with Modifies set* X is defined as:

$$\texttt{LPT}(X) \stackrel{\Delta}{=} \{t \in \texttt{PT} \mid t \text{ satisfies locality for } X\}.$$

$\texttt{LPT}(X)$ inherits the ordering from PT, which is just a pointwise ordering induced by the subset ordering of Pred. With such an ordering, $\texttt{LPT}(X)$ forms a complete lattice, in fact, a complete sublattice of PT.

Proposition 1. *$\texttt{LPT}(X)$ is a complete sublattice of* PT. *Therefore, given a set $\{t_i\}_{i \in I}$ of elements in $\texttt{LPT}(X)$, its least upper bound is given by $\lambda q.\ \bigcup_{i \in I} t_i(q)$, and its greatest lower bound is given by $\lambda q.\ \bigcap_{i \in I} t_i(q)$.*

Proof. Let r be a predicate such that $X\#r$. It suffices to show that $\bigcup_{i \in I} t_i(q * r)$ and $\bigcap_{i \in I} t_i(q * r)$ include $(\bigcup_{i \in I} t_i(q)) * r$ and $(\bigcap_{i \in I} t_i(q)) * r$, respectively. The first inclusion follows because $\bigcup_{i \in I}(t_i(q) * r) = (\bigcup_{i \in I} t_i(q_i)) * r$, and the second inclusion holds since $*$ is monotone with respect to the subset ordering. □

5.2 Operational Sensibility

The locality condition in $\texttt{LPT}(X)$ has an operational explanation via mappings, *wp* and *wlp*, from commands to predicate transformers, which correspond to total correctness and partial correctness. Let C be a command and q a predicate. Then we define the weakest, and weakest liberal, precondition predicate

$$\begin{aligned} wp(C)(q) \stackrel{\Delta}{=} \{(s,h) \mid\ & C, s, h \text{ must terminate normally} \\ & \text{and if } C, s, h \rightsquigarrow^* s', h', \text{then } s', h' \models q\} \\ wlp(C)(q) \stackrel{\Delta}{=} \{(s,h) \mid\ & C, s, h \text{ is safe} \\ & \text{and if } C, s, h \rightsquigarrow^* s', h', \text{then } s', h' \models q\} \end{aligned}$$

We can now establish the operational sensibility of local predicate transformers, which says that if $X \supseteq \text{Modifies}(C)$, both $wp(C)$ and $wlp(C)$ are local predicate transformers in $\texttt{LPT}(X)$.

[2] This command, call it C, makes $\{\{(s,h) \in q \mid |dom(h)| \leq 3\}\}\, C\, \{q\}$ true for all q in both total and partial correctness, and C is the smallest such in the sense that all other such commands satisfy more Hoare triples than C.

Proposition 2. *For a command C, both $wp(C)$ and $wlp(C)$ satisfy locality for X iff C satisfies Safety and Termination Monotonicity, Frame Property and the following locality property for variables:*

if C, s, h is safe and $C, s, h \leadsto^* s', h'$, then $s(y) = s'(y)$ for all $y \in \texttt{Variables} - X$.

Notice that this proposition establishes as close a correspondence with the locality properties of Section 4 as we would expect; since both *wp* and *wlp* completely ignore all unsafe configurations we can not obtain any properties of unsafe configurations with predicate transformers.

5.3 Proof of Completeness

We have already given a proof system for specification statements in Table 3. This determines a notion of consequence between specification statements.

$$\{p\} - \{q\} \vdash_X \{p'\} - \{q'\} \text{ iff } \{p'\} - \{q'\} \text{ can be derived from } \{p\} - \{q\} \text{ using the rules in Table 3.}$$

The semantic interpretation of each specification statement is given by a satisfaction relation between local predicate transformers in $\texttt{LPT}(X)$ and specification statements. For t in $\texttt{LPT}(X)$

$$t \models_X \{p\} - \{q\} \text{ iff } p \subseteq t(q).$$

We can now define the semantic consequence relation $\models$ by requiring that any transformer satisfying the antecedent also satisfies the consequent.

$$\{p\} - \{q\} \models_X \{p'\} - \{q'\} \text{ iff for all } t \in \texttt{LPT}(X),\ t \models_X \{p\} - \{q\} \text{ implies } t \models_X \{p'\} - \{q'\}.$$

The proof system is sound with respect to this interpretation since the monotonicity condition in $\texttt{PT}$ ensures that the rule of Consequence is sound and the locality condition for X guarantees that the Frame Rule is sound.

Proposition 3 (Soundness, II). *If $\{p\} - \{q\} \vdash_X \{p'\} - \{q'\}$, then $\{p\} - \{q\} \models_X \{p'\} - \{q'\}$.*

In the remainder of this section we concentrate on the proof of the converse.

Theorem 2 (Completeness). *If $\{p\} - \{q\} \models_X \{p'\} - \{q'\}$, then $\{p\} - \{q\} \vdash_X \{p'\} - \{q'\}$.*

The proof of completeness proceeds by finding the smallest local predicate transformer t in $\texttt{LPT}(X)$, amongst those that satisfy a given specification. To describe this transformer it will be convenient to use a notation for the spatial implication $-\!\!*$ of predicates in pointer logic [2, 5], which is itself taken from the logic of Bunched Implications [6, 7].

$$p -\!\!* \, q \stackrel{\Delta}{=} \{(s, h) \mid \forall h'.\, h' \# h \wedge s, h' \models p \Longrightarrow (s, h * h') \models q\}$$

In words, $p -\!\!* \, q$ holds of a given heap if, whenever we are given new or fresh heap satisfying p, the combined new and current heap satisfies q.

Lemma 3. *Given predicates* $p, q \in \texttt{Pred}$*, the predicate transformer* $t = \lambda r.\, p * \forall X.\,(q \mathrel{-\!\!*} r)$ *is in* $\texttt{LPT}(X)$ *and satisfies* $p \subseteq t(q)$*. Moreover, it is the smallest such in* $\texttt{LPT}(X)$ *with respect to the pointwise ordering.*

Proof. It is straightforward to see that t is monotone. To see that t satisfies the locality condition for X, pick predicate r, r' with $X\#r'$. Then, we have:

$$\begin{array}{rclcl} t(r * r') & = & p * (\forall X.\,(q \mathrel{-\!\!*} r * r')) & \supseteq & p * (\forall X.\,(q \mathrel{-\!\!*} r) * r') \\ & \supseteq & p * (\forall X.\,(q \mathrel{-\!\!*} r)) * (\forall X.\, r') & = & p * (\forall X.\,(q \mathrel{-\!\!*} r)) * r' \;=\; t(r) * r' \end{array}$$

The condition $p \subseteq t(q)$ is also easily verified as follows:

$$\begin{array}{rclcl} t(q) & = & p * (\forall X.\,(q \mathrel{-\!\!*} q)) & \supseteq & p * (\forall X.\,\mathsf{emp}) \\ & \supseteq & p * \mathsf{emp} & \supseteq & p \end{array}$$

where $\mathsf{emp} = \{(s, []) \mid s \in \texttt{Stores}\}$ is the unit of $*$. Finally, t is the smallest satisfying $p \subseteq t(q)$ because for any other $t' \in \texttt{LPT}(X)$ with $p \subseteq t'(q)$,

$$t'(r) \;\supseteq\; t'(q * \forall X.\,(q \mathrel{-\!\!*} r)) \;\supseteq\; t'(q) * \forall X.\,(q \mathrel{-\!\!*} r) \;\supseteq\; p * \forall X.\,(q \mathrel{-\!\!*} r).$$

□

We denote the smallest local predicate transformer in Lemma 3 by $\mathit{smallest}(p, q, X)$.

To prove completeness, first note that for predicates p, q, r, the precondition $\mathit{smallest}(p, q, X)(r)$ can be calculated with the Frame Rule and the rule of Consequence:

$$\dfrac{\dfrac{\{p\} - \{q\}}{\{p * \forall X.(q \mathrel{-\!\!*} r)\} \;-\; \{q * \forall X.(q \mathrel{-\!\!*} r)\}} \qquad q * \forall X.(q \mathrel{-\!\!*} r) \subseteq r}{\{p * \forall X.(q \mathrel{-\!\!*} r)\} \;-\; \{r\}}$$

The right-hand inclusion is just an application of the usual rule of $\forall$-elimination, together with a monotonicity rule for $*$ and the version of modus ponens that connects $*$ and $\mathrel{-\!\!*}$.

Now, let $\{p'\}-\{q'\}$ be a specification statement with $\{p\}-\{q\} \models_X \{p'\}-\{q'\}$. Then, we have $p' \subseteq \mathit{smallest}(p, q, X)(q')$ by Lemma 3, and $\{p'\} - \{q'\}$ can be derived from $\{p\} - \{q\}$ as follows:

$$\dfrac{p' \subseteq p * \forall X.(q \mathrel{-\!\!*} q') \qquad \dfrac{\{p\} - \{q\}}{\{p * \forall X.(q \mathrel{-\!\!*} q')\} \;-\; \{q'\}}}{\{p'\} \;-\; \{q'\}}$$

This finishes the proof of Theorem 2.

To sum up, in this paper we have shown the soundness of a rule for automatically inferring frame axioms, and we have also shown a sense in which the rule gets all the frame axioms we need. We did this by appealing to the operational semantics of a specific programming language: Perhaps the main unresolved question is whether the approach here and in [9, 2, 5] can be adapted to problems further afield. We outlined a more abstract setting for the work, and a range of other models do fit the abstract definitions, but pointer models are the only ones whose program logic has been examined in detail so far. Particularly worthwhile would be to attempt to apply the notion of local action, mentioned in Section 4.1, in an AI setting, where the frame problem originally arose and where it continues to be intensely studied [8].

Acknowledgments

Thanks to David Naumann and Uday Reddy for advice on predicate transformers, to John Reynolds for discussions on the significance of fault avoidance in specifications, and to the anonymous referees for suggesting improvements to the presentation. Yang was supported by the US NSF under grant INT-9813854 and by Creative Research Initiatives of the Korean Ministry of Science and Technology. O'Hearn was supported by the EPSRC under the Local Reasoning about State project.

References

1. R.-J. R. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978. Report A-1978-4.
2. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, pages 14–26, January 2001.
3. C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), Jul 1988.
4. D. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 2000. To appear.
5. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic: CSL 2001*, pages 1–19. Springer-Verlag. LNCS 2142.
6. P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
7. D. J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications.* Kluwer Academic Publishers, Boston/Dordrecht/London, 2002. To appear.
8. R. Reiter. *Knowledge in Action.* MIT Press, 2001.
9. 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.
10. J. C. Reynolds. Lectures on reasoning about shared mutable data structure. *IFIP Working Group 2.3 School/Seminar on State-of-the-Art Program Design Using Logic.* Tandil, Argentina, September 2000.
11. H. Yang. *Local Reasoning for Stateful Programs.* Ph.D. thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 2001.
12. H. Yang and U. S. Reddy. On the semantics of refinement calculi. In *Foundations of Software Science and Computation Structures*, pages 359–374. Springer-Verlag, 2000.