Capsules and Separation

Jean-Baptiste Jeannin and Dexter Kozen
Computer Science Department
Cornell University
Ithaca, NY 14853-7501, USA

***Abstract*—We study a formulation of separation logic using capsules, a representation of the state of a computation in higher-order programming languages with mutable variables. We prove soundness of the frame rule in this context and investigate alternative formulations with weaker side conditions.**

***Index Terms*—capsules, separation logic**

I. Introduction

Capsules [8] give a simple and mathematically appealing approach to semantics of higher-order programs that is consistent with both the functional and imperative paradigms. They minimally extend the classical λ-calculus to allow mutable variables, thus enabling the construction of certain coterms (infinite terms) representing recursive functions without the need for fixpoint combinators. They have a well-defined statically-scoped evaluation semantics, are typable with simple types, and are Turing complete.

Perhaps the most important aspect of capsules is that their evaluation semantics captures static scoping without introducing cumbersome combinatorial machinery needed to implement closures. Closures have been well known since the earliest days of functional programming. They were introduced to rectify a bug[1] in the original evaluation semantics of LISP [12]. It is perhaps well understood that static scoping corresponds to β-reduction with safe substitution in the λ-calculus, and that closures correctly implement this, but there has apparently never been a formal proof until quite recently [8]. Capsules were the key ingredient that made this proof mathematically tractable.

Capsules replace heaps, stores, stacks, and pointers with the single mathematical concept of variable binding, yet are equally expressive and represent the same data dependencies and liveness structure. In a sense, *capsules are to closures what graphs are to their adjacency list representations*. One would not reason mathematically about graphs in terms of their adjacency list representations; neither should one reason about the evaluation semantics of higher order programs in terms of their implementations. With no side effects, λ-calculus with β-reduction is the model of choice, but it does not allow for mutable variables.

Separation logic is a logic for the study of locality and shared data. Introduced by Reynolds in a series of lectures in the late 1990s, based on an earlier idea of Burstall, separation logic has been widely studied in the last decade [1], [6], [14], [16], [20], [21]. The difficulties of reasoning in the presence of heaps, stores, stacks, and pointers are no more apparent than in this literature. Several papers [4], [5], [15], [24] cite notoriously thorny issues from dangling pointers to arcane side conditions needed for soundness. Reynolds himself acknowledged that there were some difficulties with his original formulation [21]. Chief among the problems is the issue of catastrophic failure due to the dereferencing of unbound variables or dangling pointers. There seems to be an unspoken belief that this is an unavoidable aspect that must be confronted in any realistic model of computation.

On the contrary, we believe that the essential structure of separation is independent of these encumbrances. It is our thesis that freedom from catastrophic failure is the responsibility of the programming language, not the logic. Capsule semantics provides this assurance, even in the presence of higher-order constructs and mutable variables.

In this paper, we propose a semantics for separation logic based on capsules. In §IV we give the semantics and prove the soundness of the frame rule in this context. In §IV-D, we study the motivation behind the nonstandard definition of partial correctness preferred in much of the literature of separation logic [5], [21] and investigate alternatives. It is here that the advantages of capsule semantics in the study of separation can best be seen.

A. Related Work

There are two main differences between our work and the previous work on separation logic. In previous work, the authors usually adopt either an imperative, C-style programming language with low-level heap operations, or a functional, ML-style programming language with immutable variables and explicit references. According to Mason and Talcott [11], in functional languages there are two approaches to introducing objects with memory: the LISP approach, where all variables are mutable, and the ML approach, where all variables are immutable and references are introduced. One of the reasons why the ML view is usually chosen for separation logic on functional programs is that having immutable variables is the only way to get a correct semantics based on closures [18]. By using capsules instead, we are able to use mutable variables in the style of LISP.

The second main difference in this work is that we insist that all capsule environments σ are closed, i.e., that every free variable appearing in a $\sigma(x)$ should be defined in σ. To us, this seems like a very natural thing. But as far as we know, none of the previous work requires anything like this. When using

[1] subsequently upgraded to a feature, now known as *dynamic scoping*

C-style languages with an environment and a heap, writing down a similar condition would require both the environment and the heap, whereas the separation logic definitions usually only use heaps. Even Neelakantan Krishnaswami et al. [10], though using an ML-style language, explicitly say that they permit dangling pointers as long as the pointers themselves are well typed. Note that, if trying to relate the semantics of capsules with, say, a more traditional semantics using closures and a heap, the capsule environment behaves like a heap rather than like an environment in the traditional sense [7].

The original work on separation logic, summarized by Reynolds [21], uses an imperative, C-style programming language with low-level commands and already gives a proof of a version of the frame rule.

Our work is most closely related to work by Krishnaswami, Birkedal, Aldrich and Reynolds [9], [10], who give a separation logic for ML. However, our system allows mutable variables in the style of LISP, whereas theirs uses explicit references allocated in an explicit heap.

Birkedal, Torp-Smith and Yang [3] also study the frame rule in the context of a higher-order language, idealized Algol extended with heaps, but their stack variables are immutable as well.

There has been some work on so-called higher-order stores [2], [19], [22], where some code can be stored in a heap cell. Because any λ-abstractions can be stored in the environment, and executing some of them can have side-effects, our setup naturally supports higher-order stores.

II. Capsule Definitions

In this section we briefly review the definition of capsules and their semantics from [8].

A. Syntax

Expressions $\mathsf{Exp} = \{d, e, a, b, \ldots\}$ contain both functional and imperative features. There is an unlimited supply of *variables* $x, y, z, \ldots$ of all (simple) types, as well as *constants* $f, c, \ldots$ for primitive values. () is the only constant of type unit, and true and false are the only two constants of type bool. In the examples, $0, 1, 2, \ldots$ are predefined constants of type int. In addition, there are functional features

- λ-abstraction $\lambda x.e$
- application $(d\ e)$,

imperative features

- assignment $x := e$
- composition $d; e$
- conditional $\mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e$
- while loop $\mathsf{while}\ b\ \mathsf{do}\ e$,

and defined expressions

- $\mathsf{let}\ x = d\ \mathsf{in}\ e$ $(\lambda x.e)\ d$
- $\mathsf{let\,rec}\ x = d\ \mathsf{in}\ e$ $\mathsf{let}\ x = a\ \mathsf{in}\ x := d; e$

where a is any expression of the appropriate type. The technique for formation of recursive functions in the last definition is known as *Landin's knot*.

Let Var be the set of variables, Const the set of constants, and λ-Abs the set of λ-abstractions. Given an expression e, let $\mathsf{FV}(e)$ denote the set of free variables of e. Given a partial function $h : \mathsf{Var} \rightharpoonup \mathsf{Var}$ such that $\mathsf{FV}(e) \subseteq \mathsf{dom}\, h$, let $h(e)$ be the expression e where every instance of a free variable $x \in \mathsf{FV}(e)$ has been replaced by the variable $h(x)$. Thus $h : \mathsf{Exp} \rightharpoonup \mathsf{Exp}$ is the unique homomorphic extension of $h : \mathsf{Var} \rightharpoonup \mathsf{Var}$. Given two partial functions g and h, $g \circ h$ denotes their composition: $g \circ h(x) = g(h(x))$. Given a function h, we write $h[x/v]$ the function such that $h[x/v](y) = h(y)$ for $y \neq x$ and $h[x/v](x) = v$. Given an expression e, we write $e[x/y]$ for the expression e with y substituted for all free occurrences of x.

Types $\alpha, \beta, \ldots$ are ordinary simple types built inductively from an unspecified family of base types, including at least unit and bool, and the usual function type constructor $\to$. All constants c of the language have a type $\mathsf{type}(c)$; by convention, we use c for a constant of a base type and f for a constant of a functional type. Γ is a type environment, a partial function $\mathsf{Var} \rightharpoonup \mathsf{Type}$. As is standard, we write $\Gamma, x : \alpha$ for the typing environment Γ where x has been bound or rebound to α. The typing rules are standard:

$$\Gamma \vdash c : \alpha \text{ if } \mathsf{type}(c) = \alpha \qquad \Gamma, x : \alpha \vdash x : \alpha$$

$$\frac{\Gamma, x : \alpha \vdash e : \beta}{\Gamma \vdash \lambda x.e : \alpha \to \beta}$$

$$\frac{\Gamma \vdash d : \alpha \to \beta \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (d\ e) : \beta}$$

$$\frac{\Gamma \vdash x : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash x := e : \mathsf{unit}} \qquad \frac{\Gamma \vdash d : \mathsf{unit} \quad \Gamma \vdash e : \alpha}{\Gamma \vdash d; e : \alpha}$$

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash d : \alpha \quad \Gamma \vdash e : \alpha}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ d\ \mathsf{else}\ e : \alpha}$$

$$\frac{\Gamma \vdash b : \mathsf{bool} \quad \Gamma \vdash e : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\ b\ \mathsf{do}\ e : \mathsf{unit}}$$

Henceforth all the expressions we consider will be assumed to be well-typed with respect to these rules.

An expression is *irreducible* if it is either a constant or a λ-abstraction. Note that variables are not irreducible. Let $\mathsf{Irred} = \mathsf{Const} + \lambda\text{-Abs}$ denote the set of irreducible terms. (These are often called *values* in the λ-calculus literature, but we avoid this terminology here because it is misleading, as they are not values in the intuitive sense.)

A *capsule environment* is a partial function $\sigma : \mathsf{Var} \rightharpoonup \mathsf{Irred}$ satisfying the following closure condition:

$$\forall x \in \mathsf{dom}\,\sigma \quad \mathsf{FV}(\sigma(x)) \subseteq \mathsf{dom}\,\sigma.$$

This says that all free variables appearing in expressions $\sigma(x)$ must also be bound to an expression. Thus free variables are not really free; every variable in σ either occurs in the scope of a λ or is bound by σ to an expression. There may be circularities; this enables a representation of recursive functions.

The *closure* of a set $A \subseteq \mathsf{dom}\,\sigma$ with respect to σ, denoted $\mathsf{cl}_\sigma(A)$, is the smallest set B containing A such that if $x \in B$ then $\mathsf{FV}(\sigma(x)) \subseteq B$. It is the domain of the least-defined

capsule environment whose domain contains A and that agrees with σ on its domain.

A *capsule* is a pair $\langle e, \sigma\rangle$, where e is an expression and σ is a capsule environment, such that $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma$. As above, every variable appearing in $\langle e, \sigma\rangle$ either occurs in the scope of a λ or is bound by σ to an expression. These conditions preclude catastrophic failure due to access of unbound variables.

The term *α-conversion* refers to the renaming of bound variables. With a capsule $\langle e, \sigma\rangle$, this can happen in two ways. The traditional form maps a subterm $\lambda x.d$ to $\lambda y.d[x/y]$, provided y would not be captured in d. We call this *α-conversion of the first kind*. One can also rename a variable $x \in \mathsf{dom}\,\sigma$ and all free occurrences of x in e and $\sigma(z)$ for $z \in \mathsf{dom}\,\sigma$ to y, provided $y \notin \mathsf{dom}\,\sigma$ already and y would not be captured. We call this *α-conversion of the second kind*.

B. Semantics

Capsule evaluation semantics looks very much like the original evaluation semantics of LISP, with the added twist that a fresh variable is substituted for the parameter in β-reductions. The relevant small-step rule is

$$\langle(\lambda x.e\ \ v),\, \sigma\rangle \to \langle e[x/y],\, \sigma[y/v]\rangle \quad (y \text{ fresh})$$

In the original evaluation semantics of LISP, the right-hand side is $\langle e, \sigma[x/v]\rangle$, which gives dynamic scoping. As proved in [8], this simple change faithfully models β-reduction with safe substitution in the λ-calculus, providing static scoping without closures.

Another evaluation rule of particular note is the assignment rule:

$$\langle x := v,\, \sigma\rangle \to \langle(),\, \sigma[x/v]\rangle$$

where v is irreducible. The closure conditions on capsules ensure that x must already be bound in σ. The variable x is rebound to the irreducible expression v.

Other small-step rules are

$$\langle x,\, \sigma\rangle \to \langle \sigma(x),\, \sigma\rangle$$
$$\langle f\ c,\, \sigma\rangle \to \langle f(c),\, \sigma\rangle$$
$$\langle(); e,\, \sigma\rangle \to \langle e,\, \sigma\rangle$$
$$\langle \mathsf{if\ true\ then}\ d\ \mathsf{else}\ e,\, \sigma\rangle \to \langle d,\, \sigma\rangle$$
$$\langle \mathsf{if\ false\ then}\ d\ \mathsf{else}\ e,\, \sigma\rangle \to \langle e,\, \sigma\rangle$$
$$\langle \mathsf{while}\ b\ \mathsf{do}\ e,\, \sigma\rangle \to \langle \mathsf{if}\ b\ \mathsf{then}\ (e; \mathsf{while}\ b\ \mathsf{do}\ e)\ \mathsf{else}\ (),\, \sigma\rangle$$

There are also context rules that define a standard shallow applicative-order (leftmost innermost, call-by-value) evaluation strategy. The reduction rules preserve types and cannot fail catastrophically. Thus every computation either continues forever or terminates with a well-typed final capsule $\langle v, \sigma\rangle$, where v is irreducible.

The relation $\overset{*}{\to}$ is the reflexive transitive closure of $\to$.

See [8] for several examples that illustrate how the system manages recursive functions, static scoping, and garbage collection.

C. Assertions

Assertions $P, Q, \ldots$ are statements in some logical system, possibly with free variables in Var. We write $\mathsf{FV}(P)$ for the set of free variables of P. These variables are subject to interpretation provided by a capsule environment σ.

The exact nature of the underlying logic is unimportant—it could be propositional, first order, second order or higher order—but we do require a few basic properties common to standard logical systems. There must be a well-defined satisfaction relation $\models$ such that $\sigma \models P$ has a truth value when the free variables of P are interpreted by the capsule environment σ. Use of the metaexpression $\sigma \models P$ carries the tacit assumption that $\mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma$. We will augment the logic with the separation logic operators $*$ and $-\!\!*$ (defined later in §IV). Finally, we require the following (natural) property to hold: if $\sigma \models P$, and $z \in \mathsf{dom}\,\sigma - \mathsf{FV}(P)$, then z can be renamed via α-conversion of the second kind without affecting the truth of P. In examples, we will use first order logic with $*$ and $-\!\!*$, and equality on base types.

III. Partial Correctness

The traditional definition of partial correctness and the definition used in the literature on separation logic diverge in a subtle and interesting way. The difference hinges on whether the precondition is required to assert the absence of catastrophic failure due to dangling pointers or lookup of unbound variables; this is the **abort** condition of Reynolds [21] or the **fault** condition of Calcagno, O'Hearn, and Yang [5]. Our view, however, is that avoidance of catastrophic failure is the responsibility of the programming language semantics, not the program logic, and capsules do just that. Can this condition then be eliminated? In this section we shed some light on this question.

Let P, Q be assertions and e a program. At issue is the meaning of the partial correctness assertion $\{P\}\, e\, \{Q\}$. Consider the following three metastatements, each parameterized by a closed environment σ:

(A_σ) $\sigma \models P$

(B_σ) $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma$

(C_σ) if $\langle e, \sigma\rangle \overset{*}{\to} \langle v, \tau\rangle$ and v is irreducible, then $\tau \models Q$.

Statement (A_σ) entails $\mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma$, because the definition of $\models$ does not make sense without it. More strongly, $\mathsf{cl}_\sigma(\mathsf{FV}(P)) \subseteq \mathsf{dom}\,\sigma$, since σ is closed. Statement (B_σ) is equivalent to the assertion that $\langle e, \sigma\rangle$ is a valid capsule. Reynolds's definition [21] uses a slightly different formulation

(B'_σ) $\neg(\langle e, \sigma\rangle \overset{*}{\to} \textbf{abort})$

in place of (B_σ). Here $\langle e, \sigma\rangle$ need not be a valid capsule. The semantics of capsule evaluation already precludes **abort**, thus (B'_σ) is always true if $\langle e, \sigma\rangle$ is a capsule; that is, (B_σ) implies (B'_σ).

Now consider the following potential interpretations of $\{P\}\, e\, \{Q\}$.

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \wedge (\mathrm{B}_\sigma) \Rightarrow (\mathrm{C}_\sigma) \qquad (1)$$

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{B}_\sigma) \wedge (\mathrm{C}_\sigma) \qquad (2)$$

Definition (1) says that if the precondition P holds of the input state σ and the evaluation of $\langle e, \sigma\rangle$ terminates normally, then the output state τ satisfies the postcondition Q. This is the naive interpretation used in traditional forms of Hoare logic. Alternatively, the version preferred in the literature on separation logic would be (2), the difference being that the precondition P must ensure that the evaluation of $\langle e, \sigma\rangle$ cannot terminate abnormally.

Reynolds's version [21] is actually slightly weaker, using (B'_σ) instead of (B_σ):

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; \forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{B}'_\sigma) \wedge (\mathrm{C}_\sigma) \tag{3}$$

However, the difference is inconsequential: if $\{P\}e\{Q\}$ holds in the sense of (3) but not (2), then there exists a variable $x \in \mathsf{FV}(e) - \mathsf{dom}\,\sigma$ for some σ satisfying P, and consequently $x \in \mathsf{FV}(e) - \mathsf{cl}_\sigma(\mathsf{FV}(P))$; but by (B'_σ), x can never be referenced or assigned in the evaluation of $\langle e, \sigma\rangle$. Thus the presence or absence of x in the domain of σ affects neither the truth of P nor the evaluation of $\langle e, \sigma\rangle$.

But there is a much more important benefit to (2) over (3). Consider the metastatement

(B) $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$.

A consequence of (2) is that (A_σ) implies (B_σ) for all σ. If P is satisfiable at all, say by some σ, then (B) must hold, since variables in $\mathsf{dom}\,\sigma$ not occurring free in P can be renamed (by an α-conversion of the second kind—see §II-A) without affecting the truth of P. Thus (2) holds with (B) in place of (B_σ). Moreover, since (B) is independent of σ, assuming P is satisfiable at all, (2) is equivalent to the definition

$$\{P\}\, e\, \{Q\} \;\Leftrightarrow\; (\mathrm{B}) \wedge (\forall\sigma\ (\mathrm{A}_\sigma) \Rightarrow (\mathrm{C}_\sigma)) \tag{4}$$

Note that, unlike (B_σ) and (B'_σ), the condition (B) is syntactically checkable, thus suitable as a side condition in a rule of inference. If we like, we may remove the condition (B) in the definition of $\{P\}e\{Q\}$ and instead introduce it as a side condition in the frame rule. However, can it be eliminated entirely? That is, is the formulation (1) sound? We show in §IV-D that it is not. In fact, even only slightly weaker forms of the side condition (B) do not suffice for soundness.

IV. Capsules and Separation Logic

A. Definitions

Here is our semantics for separation logic in terms of capsules. Call closed environments σ and τ *independent* and write $\sigma \perp \tau$ if their domains are disjoint. Define $\sigma + \tau$ to be the join of σ and τ, provided they are independent. That is,

$$(\sigma+\tau)(x) = \begin{cases} \sigma(x), & \text{if } x \in \mathsf{dom}\,\sigma, \\ \tau(x), & \text{if } x \in \mathsf{dom}\,\tau, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Define *separating conjunction* by

$$\sigma \models P * Q$$

if there exist σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$, $\sigma_1 \models P$, and $\sigma_2 \models Q$. Define *separating implication* by

$$\sigma \models P \mathrel{-\!\!*} Q$$

if $\sigma + \tau \models Q$ whenever $\tau \models P$ and $\sigma + \tau$ exists. It is easily seen that capsule environments form a *separation algebra* in the sense of [5] under these definitions. That is, the structure

$$(\{\text{capsule environments}\},\, +,\, \varnothing)$$

is a cancellative partial commutative monoid. This means that $+$ is a commutative and associative partial binary operation with identity $\varnothing$ satisfying the *cancellative property*: the partial function $+$ is injective in each variable. The relation $\sigma \perp \tau$ holds if and only if $\sigma + \tau$ is defined.

It follows from results of [5] that separating conjunction $*$ and separating implication $\mathrel{-\!\!*}$ satisfy the usual intuitionistic relationship: For all closed σ such that $\mathsf{FV}(P) \cup \mathsf{FV}(Q) \cup \mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma$,

$$\sigma \models (P * Q) \mathrel{-\!\!*} R \;\Leftrightarrow\; \sigma \models P \mathrel{-\!\!*} (Q \mathrel{-\!\!*} R).$$

Other axioms of separation logic mentioned in [21] are also easily checked:

$$\begin{aligned} &(P \vee Q) * R \Leftrightarrow (P * R) \vee (Q * R) \\ &(P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R) \\ &(\exists x\ P) * Q \Leftrightarrow \exists x\ (P * Q) \quad (x \notin \mathsf{FV}(Q)) \\ &(\forall x\ P) * Q \Rightarrow \forall x\ (P * Q) \quad (x \notin \mathsf{FV}(Q)). \end{aligned}$$

B. The Frame Rule

The soundness of the frame rule was first proved in [24] for the heap model of computation. Our proof is essentially the same as the one given in [21], but somewhat shorter due to the simplifications afforded by capsule semantics.

Lemma 4.1: If

$$\langle e, \sigma_1 + \sigma_2\rangle \xrightarrow{*} \langle e, \tau\rangle$$

and $\mathsf{FV}(e) \subseteq \mathsf{dom}\,\sigma_1$ (that is, $\langle e, \sigma_1\rangle$ is a capsule), then for some τ_1, $\langle e, \sigma_1\rangle \xrightarrow{*} \langle e, \tau_1\rangle$ and $\tau = \tau_1 + \sigma_2$.

Proof: By induction on the derivation. None of the small-step evaluation rules listed in §II-B access any variable outside the domain of σ_1 except for fresh variables introduced in the application rule. In particular, the environment σ_2 is not touched during the evaluation. ∎

Theorem 4.2: Under capsule semantics, the frame rule

$$\frac{\{P\}\, e\, \{Q\}}{\{P * R\}\, e\, \{Q * R\}}$$

is sound with respect to definition (2) or (4) of partial correctness assertions. Equivalently, the frame rule is sound with respect to definition (1) of partial correctness assertions in the presence of the side condition $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$.

Proof: As argued in §III, in all cases we can assume $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$. Suppose $\{P\}\, e\, \{Q\}$. Let $\sigma \models P * R$. Then $\sigma = \sigma_1 + \sigma_2$ with $\sigma_1 \models P$ and $\sigma_2 \models R$. Then $\mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma_2$ and $\mathsf{FV}(e) \subseteq \mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma_1$, therefore $\langle e, \sigma_1\rangle$ is

a valid capsule. Since $\langle e, \sigma\rangle \xrightarrow{*} \langle v, \tau\rangle$, by Lemma 4.1 there exists τ_1 such that $\langle e, \sigma_1\rangle \xrightarrow{*} \langle v, \tau_1\rangle$ and $\tau = \tau_1 + \sigma_2$, and $\tau_1 \models Q$ by the premise of the rule. Thus $\tau \models Q * R$. ■

C. Discussion

Calcagno, O'Hearn, and Yang [5] argue that the soundness of the frame rule for a given evaluation semantics is equivalent to the following two properties.

Safety Monotonicity: If $\langle e, \sigma_0\rangle$ is safe and $\sigma = \sigma_0 + \sigma_1$, then $\langle e, \sigma\rangle$ is safe.

Frame Property: If $\langle e, \sigma_0\rangle$ is safe, $\sigma = \sigma_0 + \sigma_1$, and $\langle e, \sigma\rangle \xrightarrow{*} \langle e, \sigma'\rangle$, then there exists σ_0' such that $\sigma' = \sigma_0' + \sigma_1$ and $\langle e, \sigma_0\rangle \xrightarrow{*} \langle e, \sigma_0'\rangle$.

(Here we are allowing $\langle e, \sigma\rangle$ to violate the closure conditions in the definition of capsules, and *safe* means that (B'_σ) holds.) In their words, "The first condition says that if a state has enough resources for safe execution of a command, then so do superstates. The second condition says that if a state has enough resources for the command to execute safely, then execution on any bigger state can be tracked back to the small state."

With capsules, the safety monotonicity property is vacuously true, and the frame property reduces to Lemma 4.1.

D. Alternative Conditions

Recall from §III the side condition

(B) $\mathsf{FV}(e) \subseteq \mathsf{FV}(P)$,

for which the frame rule with semantics (1) for partial correctness assertions is sound. One might ask whether there is a weaker side condition that suffices for soundness. In this section we show that there is not much hope.

The frame rule as stated by Reynolds has a side condition, which says that "no variable occurring free in R is modified by e" [21]. A literal formulation of the side condition in terms of capsules is

(C) $\mathsf{AV}(e) \cap \mathsf{FV}(R) = \varnothing$,

where $\mathsf{AV}(e)$, the *assigned variables* of e, is the set of $x \in \mathsf{FV}(e)$ having a free occurrence on the left-hand side of an assignment $:=$. This is a bit confusing, because (C) seems to serve no purpose:

Theorem 4.3:

(a) Under semantics (2) of partial correctness assertions, the side condition (C) in the frame rule is redundant.
(b) Under semantics (1) of partial correctness assertions, the frame rule with side condition (C) is not sound.

Proof: First (a). As argued in §III, semantics (2) is equivalent to semantics (1) with side condition (B), provided P is satisfiable at all. We show that in all such nontrivial instances, (C) is subsumed by (B).

Suppose $\sigma \models P * R$. Then

$$\sigma = \sigma_1 + \sigma_2 \qquad \sigma_1 \models P \qquad \sigma_2 \models R.$$

By (B), we have

$$\mathsf{AV}(e) \subseteq \mathsf{FV}(e) \subseteq \mathsf{FV}(P) \subseteq \mathsf{dom}\,\sigma_1$$

and also

$$\mathsf{FV}(R) \subseteq \mathsf{dom}\,\sigma_2 \qquad \mathsf{dom}\,\sigma_1 \cap \mathsf{dom}\,\sigma_2 = \varnothing,$$

therefore (C) holds.

For (b), we give a counterexample to soundness. Let σ consist of the bindings

$$\sigma(f) = \lambda().x \qquad \sigma(x) = 2.$$

Let $R = R(f)$ be the assertion $f() = 2$. Let e be the program $x := 3$. Let $P = Q = \mathsf{true}$. The corresponding instance of the frame rule is

$$\frac{\{\mathsf{true}\}\ x := 3\ \{\mathsf{true}\}}{\{\mathsf{true} * f() = 2\}\ x := 3\ \{\mathsf{true} * f() = 2\}}$$

The premise $\{\mathsf{true}\}\ x := 3\ \{\mathsf{true}\}$ holds, but the conclusion does not. We have

$$\sigma \models \mathsf{true} * f() = 2,$$

where $\sigma = \varnothing + \sigma$, $\varnothing \models \mathsf{true}$, and $\sigma \models f() = 2$ and $\varnothing$ is the empty environment. The program e does not assign to f, the only variable free in R, yet it indirectly alters the value of f by assigning a new value to x, making R false. ■

We remark that Theorem 4.3(b) holds not just for capsules, but for any programming language with records, arrays, objects, pointers, or any form of aliasing whatsoever.

The problem at first seems to be that it is not enough to say that no variable in $\mathsf{FV}(R)$ may be modified by e; we must ensure that no variable in the closure of $\mathsf{FV}(R)$ may be modified by e, so that e cannot even indirectly alter R. This is the condition

(B_1) $\forall\sigma \quad \sigma \models P * R \Rightarrow \mathsf{AV}(e) \cap \mathsf{cl}_\sigma(\mathsf{FV}(R)) = \varnothing$

which is not expressible by any syntactic property of e, P, Q, and R.

But even this is not enough for soundness. Condition (B_1) is implied by the very strong syntactic property

(B_2) $\mathsf{AV}(e) \subseteq \mathsf{FV}(P)$

which is only slightly weaker than (B). It asserts that all free variables assigned by e are mentioned by P. Nevertheless, even (B_2) is not enough for soundness. At first this may seem quite counterintuitive, because (B_2) implies that starting in any state satisfying $P * R$, e cannot change any variable in the closure of $\mathsf{FV}(R)$, therefore cannot affect the truth of R. We state it as a theorem.

Theorem 4.4: The frame rule under semantics (1) for partial correctness assertions with side condition (B_2) is not sound.

Proof: Let σ consist of the bindings

$$\sigma(g) = \lambda x.2 \qquad \sigma(f) = \lambda().3.$$

Let $R = R(f)$ be the assertion $f() = 3$. Let e be the program

$$g := \lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ f()\ \mathsf{else}\ 2.$$

Let P and Q both be the assertion $g(0) = 2$. The corresponding instance of the frame rule is

$$\frac{\{g(0) = 2\}\ e\ \{g(0) = 2\}}{\{g(0) = 2 * f() = 3\}\ e\ \{g(0) = 2 * f() = 3\}}$$

The premise $\{g(0) = 2\}\, e\, \{g(0) = 2\}$ holds, as does the side condition (B_2), since

$$\mathsf{AV}(e) = \{g\} = \mathsf{FV}(P).$$

However, the conclusion does not. We have

$$\sigma \models g(0) = 2 * f() = 3,$$

where $\sigma = \sigma_1 + \sigma_2$, $\mathsf{dom}\,\sigma_1 = \{g\}$, $\mathsf{dom}\,\sigma_2 = \{f\}$, $\sigma_1 \models g(0) = 2$, and $\sigma_2 \models f() = 3$. However, after execution of the program e, the resulting environment binds g to a term containing a free occurrence of f, so g and f cannot be separated. ∎

V. Conclusion and Future Work

We were motivated to undertake this study in response to an anonymous review of [8] claiming that capsules "contradict the insights of separation logic, which has been extensively researched for the last decade." We hope that we have convinced the reader that there is no contradiction whatsoever—in fact quite the opposite! Capsules provide a novel perspective on separation logic, because they capture the same locality and persistence structure as traditional heap models, but in a simpler, more mathematically tractable framework. We feel that this has great potential for enhancing the understanding of separation by focusing on the essentials.

In the future, we would like to investigate other structures that have arisen in the study of separation logic in this framework. In particular, higher-order separation logic [1] proposes to use the much more powerful higher-order logic in predicates. Nested Hoare triples [22] are a neat idea to specify code stored in the heap. The anti-frame rule [17], [23] presents a way of modeling hidden state. Finally, we would like to study the concurrency rule [13] in the context of capsules.

Acknowledgments

We would like to thank Neelakantan Krishnaswami for suggesting that we look at the relation between capsules and separation logic after attending a presentation of [7]. We would also like to thank Mark Bickford, Bob Constable, and François Pottier for many useful discussions.

References

[1] B. Biering, L. Birkedal, and N. Torp-Smith, "BI-hyperdoctrines, higher-order separation logic, and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 29, August 2007. [Online]. Available: http://doi.acm.org/10.1145/1275497.1275499

[2] L. Birkedal, B. Reus, J. Schwinghammer, and H. Yang, "A simple model of separation logic for higher-order store," in *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ser. ICALP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 348–360. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70583-3_29

[3] L. Birkedal, N. Torp-Smith, and H. Yang, "Semantics of separation-logic typing and higher-order frame rules for algol-like languages," *CoRR*, vol. abs/cs/0610081, 2006.

[4] R. Bornat, C. Calcagno, and H. Yang, "Variables as resource in separation logic," in *Proc. 21st Conf. Math. Found. Programming Semantics*, 2005, pp. 247–276.

[5] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local action and abstract separation logic," in *Proc. 22nd Annual IEEE Symp. Logic in Computer Science (LICS07)*. IEEE, 2007, pp. 366–378.

[6] S. S. Ishtiaq and P. W. O'Hearn, "Bi as an assertion language for mutable data structures," in *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '01. New York, NY, USA: ACM, 2001, pp. 14–26. [Online]. Available: http://doi.acm.org/10.1145/360204.375719

[7] J.-B. Jeannin, "Capsules and closures," in *Proc. 27th Conf. Math. Found. Programming Semantics (MFPS XXVII)*, M. Mislove and J. Ouaknine, Eds. Pittsburgh, PA: Elsevier Electronic Notes in Theoretical Computer Science, May 2011.

[8] J.-B. Jeannin and D. Kozen, "Computing with capsules," Computing and Information Science, Cornell University, Tech. Rep. http://hdl.handle.net/1813/22082, January 2011.

[9] N. R. Krishnaswami, "Verifying higher-order imperative programs with higher-order separation logic," Ph.D. dissertation, Carnegie Mellon University, 2010.

[10] N. R. Krishnaswami, L. Birkedal, J. Aldrich, and J. C. Reynolds, "Idealized ML and its separation logic," http://www.cs.cmu.edu/~neelk/, 2007.

[11] I. Mason and C. Talcott, "Axiomatizing operational equivalence in the presence of side effects," in *Fourth Annual Symposium on Logic in Computer Science. IEEE*. IEEE Computer Society Press, 1989, pp. 284–293.

[12] J. McCarthy, "History of LISP," in *History of programming languages I*, R. L. Wexelblat, Ed. ACM, 1981, pp. 173–185.

[13] P. W. O'Hearn, "Resources, concurrency and local reasoning," *Theoretical Computer Science*, vol. 375, no. 1-3, pp. 271–307, May 2007.

[14] P. W. O'Hearn, J. C. Reynolds, and H. Yang, "Local reasoning about programs that alter data structures," in *Proceedings of the 15th International Workshop on Computer Science Logic*, ser. CSL '01. London, UK: Springer-Verlag, 2001, pp. 1–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=647851.737404

[15] M. Parkinson, R. Bornat, and C. Calcagno, "Variables as resource in hoare logics," in *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 137–146. [Online]. Available: http://dl.acm.org/citation.cfm?id=1157735.1158051

[16] M. J. Parkinson and G. M. Bierman, "Separation logic, abstraction and inheritance," in *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '08. New York, NY, USA: ACM, 2008, pp. 75–86. [Online]. Available: http://doi.acm.org/10.1145/1328438.1328451

[17] F. Pottier, "Hiding local state in direct style: a higher-order anti-frame rule," in *Twenty-Third Annual IEEE Symposium on Logic In Computer Science (LICS'08)*, Pittsburgh, Pennsylvania, Jun. 2008, pp. 331–340.

[18] ——, 2012, private communication.

[19] B. Reus and J. Schwinghammer, "Separation logic for higher-order store," in *In Proc. CSL*. Springer, 2006, pp. 575–590.

[20] J. C. Reynolds, "Intuitionistic reasoning about shared mutable data structures," in *Millennial Perspectives in Computer Science*, J. Davies, B. Roscoe, and J. Woodcock, Eds. Palgrave, 2000, pp. 303–321.

[21] ——, "Separation logic: A logic for shared mutable data structures," in *Proc. 17th IEEE Symp. Logic in Computer Science (LICS'02)*. IEEE, 2002, pp. 55–74.

[22] J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang, "Nested Hoare triples and frame rules for higher-order store," in *In Proceedings of the 18th EACSL Annual Conference on Computer Science Logic*, 2009.

[23] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, and B. Reus, "A semantic foundation for hidden state," in *FOSSACS*, 2010, pp. 2–17.

[24] H. Yang and P. W. O'Hearn, "A semantic basis for local reasoning," in *Proc. 5th Foundations of Software Science and Computation Structures (FOSSACS02)*, ser. Lecture Notes in Computer Science, M. Nielsen and U. Engberg, Eds., vol. 2303. Springer-Verlag, 2002, pp. 402–416.