Refinement in a Separation Context

Ivana Mijajlović
Queen Mary, University of London
ivanam@dcs.qmul.ac.uk

Noah Torp-Smith
The IT University of Copenhagen
noah@itu.dk

ABSTRACT

A separation context is a client program which does not dereference internals of the module with which it interacts. We use precise relations to unambiguously describe the storage of the module. We prove that separation contexts preserve such relations, as well as interesting properties of separation contextst in connection with refinement.

1. INTRODUCTION

Pointers wreak havoc with data abstractions [8, 7, 1, 12]. Hoare's treatment of refinement [6, 4] assumes a static-scope based separation between the abstract data type and variables of the client. Pointers break those assumptions. For example, a client, when interacting with a memory manager, will typically retain pointers to malloc blocks it has freed, which are then part of the supposedly hidden internal representation of the memory manager. However, the client must not use these pointers or otherwise its behaviour will depend on the internal representation of the malloc module.

Previous approaches to these issues [1, 12, 3, 2, 7] are based on linguistic mechanisms such as typing, to ensure that a client cannot hold a pointer into internals of a module. These solutions are limited and complex. In particular, the requirement that there are no pointers to a module internals is onerous.

Separation logic [13], on the other hand enables us to check code of a client for safety, even though the client may have pointers into the internals of a module [11].

This paper takes a first step towards bringing separation logic into refinement. We present a model, but not yet a logic, which ensures separation between a client and a module, throughout the process of refinement of the module. Even though at this stage we don't have a logic, our model is considerably simpler then ones given in [1, 12], and can easily handle examples with dangling pointers which are death to linguistic approaches. We illustrate this with the nastiest problem we know of - toy versions of `malloc` and `free`.

The paper is organized as follows. In Section 2, we give basic definitions regarding the programming language and relations on states. This enables us to define *unary separation contexts*, and to prove properties about them. A separation context is a client program that does not dereference pointers into module internals. The idea that a module owns a part of the heap is described by a *precise* relation, which is a special kind of relation that unambiguously identifies a specific portion of the heap. We show that separation contexts respect these unary relations, where arbitrary contexts do not. Finally, in Section 4 we prove a *simulation theorem* which asserts two things. The first part of the theorem states that any client that is a separation context with respect to some abstract module, is also a separation context with respect to any refinement of that module. The other part of the theorem claims that, having one module being a refinement of the other, the resulting programs obtained by plugging in the modules into a separation context will be a refinement of each other. This is a cousin of a classic logical relations or abstraction theorems. Again, it fails when a context is not a separation context. We give pointers to future work in Section 5 and conclude in Section 6.

2. PRELIMINARY DEFINITIONS

In this section, we first give a brief introduction to the novel Separation Logic, and then we give relevant definitions regarding relations in our storage model. The section ends by giving a programming language and its semantics, parametrized over the relations just defined.

2.1 An Introduction to Separation Logic

Separation logic uses the pointer model of [9] for **BI** [10] to give a program logic for programs involving pointer manipulations. It is an extension of Hoare logic, where *heaps* have been added to the storage model and the assertion language.

For the storage model, we assume a countably infinite set Var of variables given. We let S be the set of *stacks* (that is, finite, partial maps from variables to integers), and we let H be the set of *heaps* (finite, partial maps from *pointers* (an infinite subset of the integers) to integers). Then, the set of *states* is the set $(S \times H)$ of stack-heap pairs.

$$S = Var \rightharpoonup_{\text{fin}} Int \quad H = Ptr \rightharpoonup_{\text{fin}} Int$$

$$States = S \times H$$

To simplify matters, we assume that we are given two disjoint sets of variables, Var_{user} for the user language, and Var_{mod} for the implementations of the modules. The set Var is the disjoint union of these two sets. Therefore, a state $s, h \in S \times H$ can be written as $s_1 \uplus s_2, h$, where s_1 is a partial map from Var_{mod} to values and s_2 is a partial map from Var_{user} to values. We will use the notation $h\#h'$ later to denote that the two heaps h and h' have disjoint domains. If $h\#h'$, we can define the combined heap $h * h'$ as the map

$$n \mapsto \begin{cases} h(n) & \text{if } n \in dom(h) \\ h'(n) & \text{if } n \in dom(h') \end{cases}$$

The usual assertion language of Hoare logic is extended with assertions that express properties about heaps. The syntax of these assertions is

$$\mathsf{emp} \qquad e_1 \mapsto e_2 \qquad A * B \qquad \forall_* p \in m.\ A$$

The first of these asserts that the heap is empty, the second says that the current heap has exactly one pointer in its domain, and the third is the *separating conjunction* and means that the current heap can be split into two disjoint parts for which A and B hold, respectively. Finally, the last of the assertion forms is an iterated separating conjunction over a finite set. More formally, the semantics of assertions is given by a judgment

$$s, h \Vdash A,$$

which asserts that the assertion A holds in the state (s, h). We require that the free variables of A are included in the domain of s. We give the semantics of the three new assertion forms here

$$\begin{array}{rll} s, h \Vdash \mathsf{emp} & \text{iff} & dom(h) = \varnothing \\ s, h \Vdash e_1 \mapsto e_2 & \text{iff} & dom(h) = \{[\![e_1]\!]s\} \text{ and} \\ & & h([\![e_1]\!]s) = [\![e_2]\!]s \\ s, h \Vdash A * B & \text{iff} & \text{there exist } h_1, h_2 \text{ with} \\ & & h_1\#h_2, h = h_1 * h_2, \text{and} \\ & & s, h_1 \Vdash A \text{ and } s, h_2 \Vdash B \end{array}$$

$$s, h \Vdash \forall_* p \in m.\ A \text{ iff} \\ \begin{cases} s, h \Vdash \mathsf{emp} & \text{if } \mathcal{M} = \varnothing \\ s, h \Vdash A[p_1/p] * \cdots * A[p_k/p] & \text{if } \mathcal{M} = \{p_1, \ldots, p_k\} \end{cases}, \\ \text{where } \mathcal{M} = [\![m]\!]s$$

The programming language is also extended with constructs to manipulate the heap. See Sec. 2.3.

2.2 Relations

Definition 1. A relation $M \subseteq S \times H$ is *precise* if for any state s, h there is at most one subheap $h_0 \sqsubseteq h$, such that $(s, h_0) \in M$.

We illustrate precise unary relations with an example. Let α be a sequence of pairs of integers, let ls and le be pointers to the beginning and the end of the list respectively, and let start and end be the delimiters of the list. The predicate dlist is taken from [13] and is defined by induction on α as follows.

$$\begin{array}{l} \mathsf{dlist}\ \varepsilon\ (ls, \mathsf{start}, \mathsf{end}, le) \equiv \\ \quad \mathsf{emp} \wedge ls = \mathsf{end} \wedge \mathsf{start} = le \\ \mathsf{dlist}\ (a, b) \cdot \alpha\ (ls, \mathsf{start}, \mathsf{end}, le) \equiv \\ \quad \exists j.\ ls \mapsto a, b, j, \mathsf{start} * \mathsf{dlist}\ \alpha\ (j, ls, \mathsf{end}, le). \end{array}$$

where ε represents the emtpy sequence and $\cdot$ conses an element (a, b) onto the front of a sequence α. This is illustrated in Figure 1.

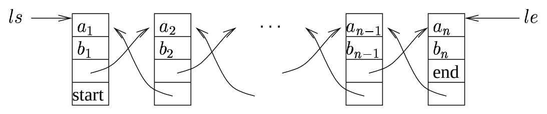

Figure 1: Illustration of the predicate dlist

Now,

$$M = \{(s, h) \mid s, h \Vdash \mathsf{dlist}\ \alpha\ (ls, nil, nil, le)\}$$

is a precise unary relation.

Definition 2. Given a precise relation M and a state (s, h), we write $dom(s, h, M)$ for the domain $dom(h_0)$ of the unique subheap $h_0 \sqsubseteq h$ such that $(s, h_0) \in M$, if it exists. Otherwise, $dom(s, h, M) = \varnothing$. Note that this definition makes sense only if M is precise.

Let $M, M' \subseteq S \times H$ be two unary relations. Then we define the *separating conjuction of unary relations*

$$M * M' = \{(s, h) \mid \exists h_0, h_1. \\ h_0\#h_1 \ \wedge\ h = h_0 * h_1 \ \wedge\ (s, h_0) \in M \ \wedge\ (s, h_1) \in M'\}.$$

Thus, we can split states into disjoint substates, such that M holds in one substate and M' holds in the other. As a special case, let $M \subseteq S \times H$ be a precise relation, and let $\mathsf{T} \subseteq S \times H$ be the relation that contains all $(s, h) \in S \times H$. Then,

$$M * \mathsf{T} = \{(s, h) \mid \exists h_0, h_1.\ h_0\#h_1 \wedge h = h_0 * h_1 \wedge (s, h_0) \in M\},$$

i.e., $M * \mathsf{T}$ consists of states (s, h) where M holds in some subheap of h

The denotation of programs will be given by binary relations $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$. The relation $M \subseteq S \times H$ is said to be *preserved* by such a relation t if for all $(s, h), (s', h')$, $(s, h) \in M$ and $(s, h)[t](s', h')$, imply $(s', h') \in M$.

Local Relations. We will consider relations on states that obey a certain "Frame discipline" only. More formally, we say that a relation $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ is *local* [11] if it satisfies the following two properties

- **Safety Monotonicity**: For all states (s, h) and heaps h_1 such that $h\#h_1$, if $\neg(s, h)[t]wrong$, then $\neg(s, h * h_1)[t]wrong$
- **Frame Property**: For all states (s, h) and heaps h_1 with $h\#h_1$, if $\neg(s, h)[t]wrong$ and $(s, h * h_1)[t](s', h')$ then there is a subheap $h'_0 \sqsubseteq h'$ such that $h'_0\#h_1$, $h'_0 * h_1 = h'$ and $(s, h)[t](s', h'_0)$.

The properties are those needed to prove the important Frame Rule from [9]. We will only consider local relations in this paper.

2.3 The Language

The programming language is an extension of the simple while-language [5] with statements for manipulating the heap. The *user language* has the following syntax:

$$\begin{array}{rl} c_{user} ::= & \mathbf{skip} \\ & \mid x := e \\ & \mid x := [e] \\ & \mid [e] := e \\ & \mid \mathsf{oper}_i,\ i \in I \\ & \mid c_1; c_2 \\ & \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\ & \mid \mathbf{while}\ e\ \mathbf{do}\ c, \end{array}$$

where

$$\begin{array}{rl} e ::= & int \mid var \mid e + e \mid \\ & e \times e \mid e - e \mid e/e, \\ & int \in Int,\ var \in Var \end{array}$$

$$\begin{array}{rcl} Int & = & \{\ldots -1, 0, 1, \ldots\} \\ Var & = & \{x, y, \ldots\} \\ I & - & \text{finite indexing set} \end{array}$$

Here, the x in the forms $x := e$ and $x := [e]$ may be taken from the set Var_{user} only. Also, the expressions used in the language may only use these variables. This means that for any program c without occurrences of oper_i executed in the state $s, h = s_1 \uplus s_2, h$ if $c, s, h \rightsquigarrow s', h'$, then s' can be written as $s_1 \uplus s'_2$ for some map $s_2 : Var_{user} \rightharpoonup Val$. This is easily verified by induction on the program c.

If $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ is a relation on states, we say that t is *constant on user variables*, if for any two states, $(s, h) = (s_1 \uplus s_2, h) \in S \times H, s_1 : Var_{mod} \rightharpoonup Int, s_2 : Var_{user} \rightharpoonup Int$ and (s', h') such that $(s, h)\ [t]\ (s', h')$, (s', h') can be written as $s'_1 \uplus s_2, h'$ for some map s'_1 from Var_{mod} to values.

The operational semantics of the language is parameterized by a precise relation M and a collection $(oper_i)_{i \in I}$ of binary relations that preserve M. It defines a big-step transition relation $\rightsquigarrow \subseteq (Comms \times (S \times H)) \times ((S \times H) \uplus \{wrong\} \uplus \{av\})$ on configurations. Informally, *wrong* is a state in which a user program dereferences a pointer which is not in the domain of the current heap. Access violation is denoted by *av* and is a state in which a user program dereferences a pointer which belongs to the domain of the current heap, but does not belong to the user's part of the heap, i.e. a pointer owned by the module. We use precise relations to keep track of this. The operational semantics of the language is given by the following inference rules:

$$\frac{}{\mathbf{skip}, s, h \rightsquigarrow s, h}$$

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

$$\frac{[\![e_1]\!]s = p\ \wedge p \in dom(h)\ \wedge\ p \notin dom(s, h, M) \quad [\![e_2]\!]s = n}{[e_1] := e_2, s, h \rightsquigarrow s, h(p \mapsto n)}$$

$$\frac{[\![e_1]\!]s = p\ \wedge\ p \notin dom(h)}{[e_1] := e_2, s, h \rightsquigarrow wrong} \quad \frac{[\![e_1]\!]s = p\ \wedge\ p \in dom(s, h, M)}{[e_1] := e_2, s, h \rightsquigarrow av}$$

$$\frac{[\![e]\!]s = p\ \wedge p \in dom(h)\ \wedge\ p \notin dom(s, h, M) \quad h(p) = n}{x := [e], s, h \rightsquigarrow s(x \mapsto n), h}$$

$$\frac{[\![e]\!]s = p\ \wedge\ p \notin dom(h)}{x := [e], s, h \rightsquigarrow wrong} \quad \frac{[\![e]\!]s = p\ \wedge\ p \in dom(s, h, M)}{x := [e], s, h \rightsquigarrow av}$$

$$\frac{(s, h)[oper_i](s', h')}{\mathsf{oper}_i, s, h \rightsquigarrow s', h'} \quad \frac{(s, h)[oper_i]wrong}{\mathsf{oper}_i, s, h \rightsquigarrow wrong}$$

$$\frac{c_1, s, h \rightsquigarrow s', h' \quad c_2, s', h' \rightsquigarrow K}{c_1; c_2, s, h \rightsquigarrow K}$$

$$\frac{c_1, s, h \rightsquigarrow wrong}{c_1; c_2, s, h \rightsquigarrow wrong} \quad \frac{c_1, s, h \rightsquigarrow av}{c_1; c_2, s, h \rightsquigarrow av}$$

$$\frac{[\![e]\!]s = 0 \quad c_2, s, h \rightsquigarrow K}{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, s, h \rightsquigarrow K} \quad \frac{[\![e]\!]s \neq 0 \quad c_1, s, h \rightsquigarrow K}{\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, s, h \rightsquigarrow K}$$

$$\frac{[\![e]\!]s = 0}{\mathbf{while}\ e\ \mathbf{do}\ c, s, h \rightsquigarrow s, h}$$

$$\frac{[\![e]\!]s = 1 \quad c; \mathbf{while}\ e\ \mathbf{do}\ c, s, h \rightsquigarrow K}{\mathbf{while}\ e\ \mathbf{do}\ c, s, h \rightsquigarrow K}$$

where $K \in (S \times H) \uplus \{wrong\} \uplus \{av\}$.

We employ precise unary relations in order to uniquely identify the storage of the module. This allows us to explicitely ask if a particular cell belongs to the part of the heap described by the precise relation, i.e. to the module.

Remark 1. We are interested only in starting states which are related by $M * \top$, but the operational semantics is also valid for commands starting in states not related by $M * \top$.

3. UNARY SEPARATION CONTEXTS

Definition 3. Let $M \subseteq S \times H$ be a precise unary relation, and for $i \in I$ let $oper_i \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ preserve relation $M * \top$. A program c is a *unary separation context* for M and $(oper_i)_{i \in I}$ if for all executions and all $(s, h) \in M * \top$ $c, s, h \not\rightsquigarrow av$ and $c, s, h \not\rightsquigarrow wrong$.

The intuitive meaning of this definition is that M describes the set of pointers owned by the module, and even though a user program may have the pointers to heap cells owned by the module, if it is a separation context, it will never dereference those pointers.

Remark 2. It is not the case that all subcommands of commands that are separation contexts are separation contexts themselves. This can be seen from the counterexample **if** 0 **then** c **else** $x := 4$, where c is a command which results in an access violation.

If a user program uses more than one module, then it is a separation context if it is a separation context with respect to all the modules. This approach is non-modular, since, for all we know, two different modules might share resources. This means that dereferencing internals of one module might imply dereferencing internals of the other. In future research, we will investigate the concept of "independent modules".

Unary separation contexts preserve precise unary relations. We say that a precise unary relation is *independent of user variables* if $(s, h) = (s_m \uplus s_u, h) \in M$ then for all $s'_u : Var_{user} \rightarrow Int$, $(s_m \uplus s'_u, h) \in M$.

THEOREM 1. *Let $M \subseteq S \times H$ be a precise relation independent on user variables, and for $(i \in I)$ let $oper_i \subseteq S \times H \times (S \times H) \uplus \{wrong\}$ preserve $M * \top$, and let c be a separation context for M and $(oper_i)_{i \in I}$. If $(s, h) \in M * \top$, and $c, s, h \rightsquigarrow s', h'$, then $(s', h') \in M * \top$.*

Proof: The proof is by induction on the command c. The simplest cases are **skip**, $x := e$, and $x := [e]$; in these cases we use the fact that M is independent of user variables.

If $c \equiv [e_1] := e_2$, we consider the inference rule

$$\frac{[\![e_1]\!]s = p \ \wedge p \in dom(h) \ \wedge \ p \notin dom(s, h, M) \quad [\![e_2]\!]_s = n}{[e_1] := e_2, s, h \rightsquigarrow s, h(p \mapsto n)},$$

supposing $(s, h) \in M * \top$. M is precise, which means that there is at most one $h_0 \sqsubseteq h$ such that $(s, h_0) \in M$. Then $h = h_0 * h_1$ and $(s, h_0) \in M$. By the assumption $p \notin dom(s, h, M) = (s, h_0)$. Now,

$$s, h(p \mapsto n) \ = \ s, h_0 {*} h_1(p \mapsto n) \ = \ s, h_0 {*} h_1(p \mapsto n) \ \in \ M {*} \top,$$

and $(s, h_0) \in M$ and $(s, h_1(p \mapsto n)) \in \top$.

In the case of oper_i, we use the rule for oper_i in the operational semantics, and the assumption that $oper_i$ preserves $M * \top$.

For the induction step, we first see that the cases of composition and **if**-branches are straightforward, noting that we only need to consider one of the rules in the operational semantics in each case. For the **while** loop, we do an inner induction on the smallest derivation of **while** b **do** $c, s, h \rightsquigarrow s', h'$. $\square$

3.1 Examples

We illustrate unary separation contexts with an example and a non-example.

We define $\mathsf{new}()$ and $\mathsf{dispose}$ as the operations of the memory management module, in the following way.

$$\begin{array}{rcl}
\mathsf{new}(x) & = & \{((s,h),(s',h')) \mid \\
& & (s(f) = Z \uplus \{n\} \wedge s' = s[x \mapsto n, f \mapsto Z] \wedge \\
& & \quad h' = h[n \mapsto nil, nil]) \ \vee \\
& & (s(f) = \varnothing \wedge n \notin dom(h) \wedge \\
& & \quad h' = h * n \mapsto nil, nil \wedge s' = s[x \mapsto n])\} \\
\mathsf{dispose}(x) & = & \{((s,h),(s',h')) \mid s(f) = Z \subseteq dom(h) \wedge \\
& & \quad s(x) = n \wedge n \in dom(h) \wedge n \notin s(f) \\
& & \quad s' = s[f \mapsto Z \uplus \{n\}]\} \ \cup \\
& & \quad \{(s,h), wrong) \mid s(x) = n \ \wedge \\
& & \quad n \notin dom(h) \ \vee \ n \in s(f)\}
\end{array}$$

Now an example of a separation context interacting with the memory management module is

```
$Program\ User$ :
begin
  $x := \mathsf{new}()$;
  $x := 47$;
  $\mathsf{dispose}(x)$;
end
```

Here are two examples of programs which are not separation contexts:

```
$Program\ User_1$ :
begin
  $x := \mathsf{new}()$;
  $\mathsf{dispose}(x)$;
  $[x] := 47$;
end
```

```
$ProgramUser_2$ :
begin
  $x := \mathsf{new}()$;
  $\mathsf{dispose}(x)$;
  $\mathsf{dispose}(x)$;
end
```

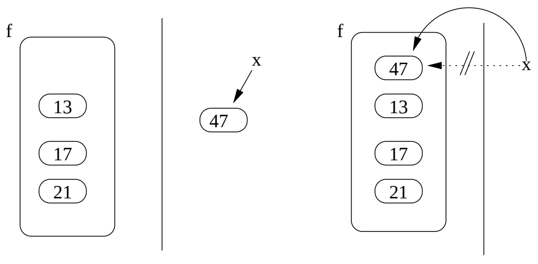

Figure 2: Heap of module and $User_1$ after allocating and disposing pointer in variable x

In both examples, user program dereferences a pointer which it does not own. Program $User_1$ after allocating a new cell, calls procedure dispose, and thereby gives up the ownership of the pointer denoted by x. Afterwards it dereferences variable x, which no longer belongs to the user program which causes an access violation. In the second example, $User_2$ after allocating and disposing a pointer held in variable x, attempts to dispose the same pointer again, which makes the program go *wrong*.

4. REFINEMENT AND SEPARATION

4.1 Binary Relations

Let $R \subseteq (S \times H) \times (S \times H)$ be a binary relation on states. We say that R is *precise*, if each of its two projections on the set of states is precise; that is, R is precise if

- for any state (s, h) there is at most one $h_1 \sqsubseteq h$ such that there exists a state (s', h') such that $(s, h_1)[R](s', h')$, and
- for any (s', h'), there is at most one $h'_1 \sqsubseteq h'$ such that there is a state (s, h) with $(s, h)[R](s', h'_1)$.

To illustrate precise binary relations with an example, suppose we have two different implementations of a storage manager module. In the first implementation we assume that f is a set variable, which keeps track of all owned locations. In the second implementation, we let this information be kept in a list. We use a list predicate $\mathsf{list}(\alpha, \mathsf{ls}, \mathsf{end})$, which is defined inductively on the sequence α of integers, by

$$\mathsf{list}(\varepsilon, i, j) \stackrel{\text{def}}{=} \mathsf{emp} \wedge i = j$$
$$\mathsf{list}(p \cdot \alpha, i, j) \stackrel{\text{def}}{=} i = p \wedge \exists k.\ i \mapsto -, k * \mathsf{list}(\alpha, k, j)$$

Note that if $s, h \ \Vdash \ \mathsf{list}(\alpha, i, j)$, then there are no elements occurring twice in α.

Now, a precise binary relation R relating these two implementations is given by

$$R = \{((s,h),(s',h')) \mid (s, h \ \Vdash \ \forall_* p \in f.\ p \mapsto -, -) \ \wedge \\ (s', h' \ \Vdash \ \mathsf{list}(\alpha, \mathsf{ls}, nil)) \ \wedge \ set(\alpha) = s(f)\},$$

where $set(\alpha)$ is defined as the set of pointers in the sequence α.

Relation R relates pairs of states, such that one state can be described as a set of different pointers, while the other one is determined by the list of exactly the pointers that appear in mentioned set.

For two binary relations $R, R' \subseteq (S \times H) \times (S \times H)$ on states, we define their separating conjunction [12] as

$$\begin{aligned} R * R' \;=\; & \{((s,h),(s',h')) \mid \exists s_1, h_1, s_2, h_2, s_1', h_1', s_2', h_2'. \\ & h = h_1 * h_2 \;\wedge\; s = s_1 \uplus s_2 \;\wedge \\ & h' = h_1' * h_2' \;\wedge\; s' = s_1' \uplus s_2' \;\wedge \\ & (s_1,h_1)\ [R]\ (s_1',h_1') \;\wedge\; (s_2,h_2)\ [R']\ (s_2',h_2')\} \end{aligned}$$

Note that this set might be empty. The idea is that R and R' relate disjoint part of the states.

As a special case, let $R \subseteq (S \times H) \times (S \times H)$ be a precise binary relation, and let $\mathbf{Id} \subseteq (S \times H) \times (S \times H)$ be the identity relation. Then,

$$\begin{aligned} R * \mathbf{Id} \;=\; & \{((s,h),(s',h')) \mid \exists s_0, s_1, h_0, h_1, s_0', h_0'. \\ & h = h_0 * h_1 \;\wedge\; s = s_0 \uplus s_1 \;\wedge \\ & h' = h_0' * h_1 \;\wedge\; s' = s_0' \uplus s_1 \wedge \\ & (s_0,h_0)\ [R]\ (s_0',h_0')\}. \end{aligned}$$

4.2 Refinement and Separation Contexts

In this section, we will formally express what it means for one module to be an implementation (or *refinement*) of another. For simplicity, we will assume that there is only one operation of the module, i.e., that the index set I from the syntax of the user language is singleton. In [4], our definition of refinement is called upward simulation.

Definition 4. We define $oper_2 \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ to be a *refinement* of $oper_1 \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ with respect to $R \subseteq (S \times H) \times (S \times H)$, if

- for all states $(s_1,h_1), (s_2,h_2), (s_2',h_2')$ such that (s_1,h_1) $[R * \mathbf{Id}]$ (s_2,h_2) and (s_2,h_2) $[oper_2]$ (s_2',h_2') there exists a state (s_1',h_1') such that $(s_1,h_1)[oper_1](s_1',h_1')$ and $(s_1',h_1')[R * \mathbf{Id}](s_2',h_2')$, and
- for all states $(s_1,h_1), (s_2,h_2)$, if $(s_1,h_1)[R*\mathbf{Id}](s_2,h_2)$, then $(s_2,h_2)[oper_2]wrong$ implies $(s_1,h_1)[oper_1]wrong$.

This can be illustrated by the following diagrams.

$$\begin{tikzcd} (s_1,h_1) \arrow[r, leftrightarrow, "oper_1"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & (s_1',h_1') \arrow[d, leftrightarrow, "R*\mathbf{Id}"] \\ (s_2,h_2) \arrow[r, leftrightarrow, "oper_2"] & (s_2',h_2') \end{tikzcd} \qquad \begin{tikzcd} (s_1,h_1) \arrow[r, leftrightarrow, "oper_1"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & wrong \\ (s_2,h_2) \arrow[r, leftrightarrow, "oper_2"] & wrong \end{tikzcd}$$

For the following theorem, we let $R \subseteq (S \times H) \times (S \times H)$ be a precise binary relation independent of user variables, and let $oper_2 \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ be a refinement of $oper_1 \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ with respect to R. Let c be a program, and let $c_i \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ be a relation denoted by c in the operational semantics defined by R_i and $oper_i$, $i = 1, 2$, where R_i is the ith projection of R onto $(S \times H)$.

THEOREM 2 (SIMULATION THEOREM). *Let R, $oper_i$, c, c_i be as above. Then, if c_1 is a separation context for R_1 and $oper_1$, then c_2 is a separation context for R_2 and $oper_2$ and for all $(s_1,h_1), (s_2,h_2), (s_2',h_2')$ if (s_1,h_1) $[R * \mathbf{Id}]$ (s_2,h_2) and $(s_2,h_2)[c_2](s_2',h_2')$ then there exists a state (s_1',h_1') such that $(s_1,h_1)[c_1](s_1',h_1')$ and $(s_1',h_1')[R * \mathbf{Id}](s_2',h_2')$.*

To prove Theorem 2, we prove a lemma similar to it. For this, we need the following definition.

Definition 5. Let $M \subseteq (S \times H) \times (S \times H)$ be a precise unary relation on states, let $oper \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ be an operation, and let (s,h) be a state. We say that the command c is *safe at (s,h) with respect to $(M, oper)$*, if $c, s, h \not\rightsquigarrow wrong$ and $c, s, h \not\rightsquigarrow av$, where $\rightsquigarrow$ is the operational semantics defined by $oper$ and M.

LEMMA 1. *Let R, $oper_i$, c, c_i be as above, and suppose $(s_1,h_1)[R * \mathbf{Id}](s_2,h_2)$. Then, if c_1 is safe at (s_1,h_1) with respect to $(R_1, oper_1)$, c_2 is safe at (s_2,h_2) with respect to $(R_2, oper_2)$, and if $(s_2,h_2)[c_2](s_2',h_2')$ then there exists a state (s_1',h_1') such that $(s_1,h_1)[c_1](s_1',h_1')$ and $(s_1',h_1')[R*\mathbf{Id}](s_2',h_2')$.*

Proof: The proof is by induction on the program c. We will make use of diagrams in the argumentation, since these convey good understanding of refinement conditions. The cases for **skip** and $x := e$ are easy and the case for oper just uses the assumption that the semantic *oper*s are refinements of each other.

Next, we consider the case where c is $x := [e]$, and suppose $(s_1,h_1)[R*\mathbf{Id}](s_2,h_2)$. By assumptions about user variables, $s_1(x) = s_2(x)$, and $[\![e]\!]s_1 = [\![e]\!]s_2 = n$. Since c_1 is safe at (s_1,h_1) with respect to $(R_1, oper_1)$, $n \in dom(h_{12})$ (see the diagram below), and we let $n' = h_{12}(n)$. Since heap lookup does not alter the heap, c_2 is safe at (s_2,h_2) with respect to $(R_2, oper_2)$, and we have the following diagram.

$$\begin{tikzcd} (s_{11},h_{11}) * (s_{12},h_{12}) \arrow[r, equal] & (s_1,h_1) \arrow[r, leftrightarrow, "x:={[e]}"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & (s_1[x \mapsto n'],h_2) \arrow[d, leftrightarrow, "R*\mathbf{Id}"] \\ (s_{21},h_{21}) * (s_{12},h_{12}) \arrow[r, equal] & (s_2,h_2) \arrow[r, leftrightarrow, "x:={[e]}"'] & (s_2[x \mapsto n'],h_2) \end{tikzcd}$$

We have that the rightmost states in this diagram are related by $R * \mathbf{Id}$ because the heaps are unchanged and the stacks are only changed on a user variable. The case where c is $[e_1] := e_2$ is similar to this case.

For the induction step, the case for **if**-branches is easy when we use the fact that R is independent of user variables again. For the case of composition, assume that c is $c'; c''$. Since c_1 is safe at (s_1,h_1) with respect to $(R_1, oper_1)$, c_1' is too. By the induction hypothesis, this implies that c_2' is safe at (s_2,h_2) with respect to $(R_2, oper_2)$, suppose $c_2', s_2, h_2 \rightsquigarrow s_2'', h_2''$. Then we have that there is a state (s_1'',h_1'') such that $c_1', s_1, h_1 \rightsquigarrow s_1'', h_1''$ and $(s_1'',h_1'')[R * \mathbf{Id}](s_2'',h_2'')$, by the induction hypothesis. c_1'' is safe at (s_1'',h_1'') with respect to $(R_1, oper_1)$, since c_1 was safe at (s_1,h_1) with respect to $R_1, oper_1$, so we can use the induction hypothesis and repeat the argument to get the following diagram.

$$\begin{tikzcd} (s_1,h_1) \arrow[r, leftrightarrow, "c_1'"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & (s_1'',h_1'') \arrow[r, leftrightarrow, "c_1''"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & (s_1',h_1') \arrow[d, leftrightarrow, "R*\mathbf{Id}"] \\ (s_2,h_2) \arrow[r, leftrightarrow, "c_2'"'] & (s_2'',h_2'') \arrow[r, leftrightarrow, "c_2''"'] & (s_2',h_2') \end{tikzcd}$$

When c is **while** b **do** c', and c_1 is safe at (s_1, h_1) with respect to $(R_1, oper_1)$, we prove by induction that for all $(s_1, h_1)[R * \mathbf{Id}](s_2, h_2)$ and derivations $c_1, s_1, h_1 \rightsquigarrow s_1', h_1'$ of depth n, there is a state (s_2', h_2') such that $c_2, s_2, h_2 \rightsquigarrow s_2', h_2'$ and $(s_1', h_1')[R * \mathbf{Id}](s_2', h_2')$. When $n = 0$, we use the rule

$$\frac{[\![b]\!]s = 0}{c_1, s_1, h_1 \rightsquigarrow s_1, h_1,}$$

and it is easy to see that nothing happens for the program c_2 either, so the conditions for the theorem are clearly fulfilled. Now, suppose that the last step in the derivation is an application of the rule

$$\frac{[\![b]\!]s_1 = 1 \quad c_1', s_1, h_1 \rightsquigarrow s_1'', h_1'' \quad c_1, s_1'', h_1'' \rightsquigarrow s_1', h_1'}{c_1, s_1, h_1 \rightsquigarrow s_1', h_1'}$$

By the outer induction hypothesis, c_2' is safe at (s_2, h_2) with respect to $(R_2, oper_2)$, and there is a state (s_2'', h_2'') such that the left part of the diagram below "commutes".

$$\begin{tikzcd}
(s_1, h_1) \arrow[r, leftrightarrow, "c_1'"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"'] & (s_1'', h_1'') \arrow[r, leftrightarrow, "c_1"] \arrow[d, leftrightarrow, "R*\mathbf{Id}"] & (s_1', h_1') \arrow[d, leftrightarrow, "R*\mathbf{Id}"] \\
(s_2, h_2) \arrow[r, leftrightarrow, "c_2'"'] & (s_2'', h_2'') \arrow[r, leftrightarrow, "c_2"'] & (s_2', h_2')
\end{tikzcd}$$

The derivation in the upper right corner of this diagram is shorter, so the inner induction hypothesis gives us that there is a state (s_2', h_2') with $c_2, s_2'', h_2'' \rightsquigarrow s_2', h_2'$ and $(s_1', h_1')[R * \mathbf{Id}](s_2', h_2')$. This means that c_2 is safe at (s_2, h_2) with respect to $(R_2, oper_2)$. We also have that when $c_2, s_2'', h_2'' \rightsquigarrow (s_2', h_2')$, the diagram gives us a (s_1', h_1') with the desired properties. This completes the proof of the lemma. $\square$

The proof of Theorem 2 is easy using Lemma 1. The meaning of Theorem 2 is the following: if a user program c is a separation context for an abstract module, then it is a separation context for any refinement of that module. For any separation context c, the concrete denotation c_2 of c refines the abstract denotation c_1 of c.

4.3 Example

We have three implementations of `malloc`, one which is completely abstract, one which is more concrete, but still has a certain degree of abstractness to it, and finally, a "concrete" one that uses a free-list. We will argue that the "intermediate" implementation is a refinement of the abstract one and that the concrete implementation is a refinement of the intermediate one. For each of the implementations, we exhibit relations which implement the operations new and dispose. In the following, we assume that heaps map pointers to *pairs* of integers. We use $n \mapsto -, -$ to denote a singleton heap.

For the abstract implementation, we have the following relations.

$$\begin{array}{rcl}
\mathsf{new}_1(x) & = & \{((s,h),(s',h')) \mid n \notin \mathit{dom}(h) \wedge \\
 & & \quad h' = h * n \mapsto \mathit{nil}, \mathit{nil} \wedge s' = s[x \mapsto n]\} \\
\mathsf{dispose}_1(x) & = & \{((s,h),(s',h')) \mid n \in \mathit{dom}(h) \wedge \\
 & & \quad s(x) = n \wedge h' = h \setminus \{(n, h(n))\}\} \cup \\
 & & \{((s,h), \mathit{wrong}) \mid s(x) \notin \mathit{dom}(h)\}
\end{array}$$

The intention is that the abstract module does not own any of the locations, but when a user program asks for a new location, the module also asks for a location from the system. When the user program gives up on a location, it is immediately returned to the system. Therefore, the resource invariant of the module is emp.

For the intermediate implementation, we have the following relations.

$$\begin{array}{rcl}
\mathsf{new}_2(x) & = & \{((s,h),(s',h')) \mid \\
 & & (s(f) = Z \uplus \{n\} \wedge s' = s[x \mapsto n, f \mapsto Z] \wedge \\
 & & \quad h' = h[n \mapsto \mathit{nil}, \mathit{nil}]) \vee \\
 & & (s(f) = \varnothing \wedge n \notin \mathit{dom}(h) \wedge \\
 & & \quad h' = h * n \mapsto \mathit{nil}, \mathit{nil} \wedge s' = s[x \mapsto n])\} \\
\mathsf{dispose}_2(x) & = & \{((s,h),(s',h')) \mid s(f) = Z \subseteq \mathit{dom}(h) \wedge \\
 & & \quad s(x) = n \wedge n \in \mathit{dom}(h) \wedge n \notin s(f) \\
 & & \quad s' = s[f \mapsto Z \uplus \{n\}]\} \cup \\
 & & \quad \{(s,h), \mathit{wrong}) \mid s(x) = n \wedge \\
 & & \quad n \notin \mathit{dom}(h) \ \vee \ n \in s(f)\}
\end{array}$$

The intention is that we keep a set of owned locations in the module. If this set becomes empty, we call a "system routine" (like `sbrk`) to get a new location.

In this case, the resource invariant is determined by the variable f:

$$\forall_* p \in f.\ p \mapsto -, -$$

The concrete implementation is the more realistic one. We keep a *free-list* of the cells that have been disposed by the user program. When the user calls new, we return the first element in the list, if it is not empty. If it is empty, we call a system routine like in the intermediate implementation. The list is singly linked, and the delimiters are ls and *nil*. Here are the relations for this implementation.

$$\begin{array}{l}
\mathsf{new}_3(x) = \\
\{((s,h),(s',h')) \mid \\
\quad (s' = s[x \mapsto s(\mathsf{ls}), \mathsf{ls} \mapsto h(s(\mathsf{ls})).2, \alpha \mapsto \mathsf{tl}(s(\alpha))] \wedge \\
\quad h' = h[s(\mathsf{ls}) \mapsto \mathit{nil}, \mathit{nil}] \wedge \ s(\alpha) \neq \varepsilon) \vee \\
\quad (s(\mathsf{ls}) = s'(\mathsf{ls}) = \mathit{nil} \ \wedge \ n \notin \mathit{dom}(h) \ \wedge \ s(\alpha) = \varepsilon \wedge \\
\quad h' = h * n \mapsto \mathit{nil}, \mathit{nil} \ \wedge \ s' = s[x \mapsto n])\} \\
\mathsf{dispose}_3(x) = \\
\{((s,h),(s',h')) \mid s(x) = n \wedge n \in \mathit{dom}(h) \wedge n \notin s(\alpha) \wedge \\
\quad s' = s[\alpha \mapsto s(x) \cdot s(\alpha), \mathsf{ls} \mapsto s(x)] \ \wedge \\
\quad h' = h[s(x).2 \mapsto s(\mathsf{ls})]\} \cup \{((s,h), \mathit{wrong}) \mid s(x) = n \\
\quad \wedge\ n \notin \mathit{dom}(h) \ \vee \ n \in \mathit{set}(\alpha)\}
\end{array}$$

where $\mathit{set}(\alpha)$ is the set of pointers in the sequence α.The resource invariant for this implementation is the variant of the well-known list predicate defined in Section 4.1.

To show that the intermediate implementation is a refinement of the abstract one, we exhibit a binary relation on states and argue that the intermediate is a refinement of the abstract one with respect to this relation.

The binary relation $R_1 \subseteq (S \times H) \times (S \times H)$ is:

$$\begin{array}{l}
R_1 = \{((s,h),(s',h')) \mid \\
\qquad s, h \Vdash \mathsf{emp} \wedge s', h' \Vdash \forall_* p \in f.\ p \mapsto -, -\}
\end{array}$$

Suppose $(s_1, h_1)[R_1 * \mathbf{Id}](s_2, h_2)$, and $(s_2, h_2)[\mathsf{new}_2(x)](s_2', h_2')$. We must argue that there exists a state (s_1', h_1') such that $(s_1, h_1)[\mathsf{new}_1(x)](s_1', h_1')$ and (s_1', h_1') $[R_1 * \mathbf{Id}](s_2', h_2')$. From $(s_2, h_2)[\mathsf{new}_2(x)](s_2', h_2')$, we have that $n \in \mathit{dom}(h_2)$, and $h_2' = h_2[n \mapsto \mathit{nil}, \mathit{nil}]$, which means that location n is in the current heap, but not in the user's part of the heap, or $n \notin$

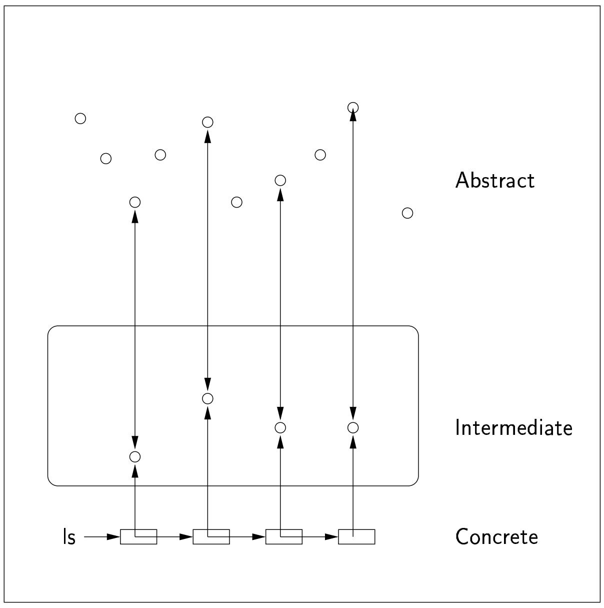

Figure 3: An illustration of the three representations of the memory management module.

$dom(h_2)$ and $h'_2 = h_2 * n \mapsto nil, nil$, in which case, n does not belong to the current heap. In both cases, n evidently does not belong to the domain of the user's part of the heap h_2, and because of the assumption $(s_1, h_1)[R * \mathbf{Id}](s_2, h_2)$, n can not be in the user's part of the heap h_1 either. So, n must be owned by the system, and we construct a state (s'_1, h'_1) such that $h'_1 = h_1 * n \mapsto nil, nil$ and $s'_1 = s_1[x \mapsto n]$. It is not hard to see that $(s_1, h_1)[\mathsf{new}_1(x)](s'_1, h'_1)$ and $(s'_1, h'_1)[R * \mathbf{Id}](s'_2, h'_2)$, as desired.

We omit the proof for dispose.

Next, we show that the concrete implementation is a refinement of the intermediate one. This will imply that the concrete implementation is a refinement of the abstract, since the refinement relation is easily seen to be transitive. We show the claim in the same way as before, namely by giving a binary relation on states and show that the concrete implementation is a refinement of the intermediate one with respect to this relation.

The binary relation R_2 is given by

$$\begin{aligned} R_2 \quad = \quad & \{((s,h),(s',h')) \mid (s,h \Vdash \forall_* p \in f.\ p \mapsto -,-) \wedge \\ & (s',h' \Vdash \mathsf{list}(\alpha, \mathsf{ls}, nil)) \wedge set(\alpha) = s(f)\}, \end{aligned}$$

Suppose $(s_2, h_2)[R_2 * \mathbf{Id}](s_3, h_3)$, and $(s_3, h_3)[\mathsf{new}(x)](s'_3, h'_3)$. There are two cases

If $s'_3 = s_3[x \mapsto s_3(\mathsf{ls}), ls \mapsto h_3(s_3(\mathsf{ls})).2, \alpha \mapsto \mathsf{tl}(s_3(\alpha))]$, $h'_3 = h_3[s_3(\mathsf{ls}) \mapsto nil, nil]$, and $s'_3(\alpha) \neq \varepsilon$, we have $s_3(\mathsf{ls}) = \mathsf{hd}(s_3(\alpha)) \in s_2(f)$ because of the definition of R_2. We can therefore construct the state

$$\begin{aligned} &(s'_2, h'_2) = \\ &\quad (s_2[x \mapsto s_3(\mathsf{ls}), f \mapsto s_2(f) \setminus \{s_3(\mathsf{ls})\}], h_2[s_3(\mathsf{ls}) \mapsto nil, nil]), \end{aligned}$$

and we see that this state has the desired properties, namely $(s'_2, h'_2)[R_2 * \mathbf{Id}](s'_3, h'_3)$ and $(s_2, h_2)[\mathsf{new}_2(x)](s'_2, h'_2)$.

The other case is when $s'_3 = s_3[x \mapsto n]$, $s(\mathsf{ls}) = nil$, $h'_3 = h_3 * n \mapsto nil, nil$, $n \notin dom(h_3)$, and $s_3(\alpha) = \varepsilon$. In this case, we have $s_2(f) = \varnothing$, so the state

$$(s'_2, h'_2) = (s[x \mapsto n], h_2 * n \mapsto nil, nil)$$

has the desired properties.

5. FUTURE WORK

We already mentioned that our approach is non-modular, in the sense that we have only dealt with one module and have not considered resources that are shared between modules, nor explored the situation where there is more than one module in play at the same time.

We defined c to be a unary separation context if under certain conditions, starting from any state, c does not access violate and does not go wrong. This actually means that c has a precondition **true** and that c has to build a heap for itself before dereferencing any heap locations, or in other words, to always have a valid initialization.

A similar result would be achieved if we required a certain precondition to hold before executing c. The corresponding definition might be:

Definition 6. Let $M, P \subseteq S \times H$ be unary relations, and let M be precise. Let $oper_i \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$ preserve relation $M * \mathsf{T}$. A program c is a *unary separation context* for (M, P) if for all $(s, h) \in M * P$ $c, s, h \not\rightsquigarrow av$ and $c, s, h \not\rightsquigarrow wrong$.

This means that c can be executed in a state in which resources available to c already exist.

O'Hearn *et al.* have given criteria for proving specifications of programs under the assumption of already proven specifications for modules, by introducing and proving sound the Hypothetical Frame Rule [11]. For practical purposes, it would be desirable to have a similar rule for showing properties about refinement. Letting

$$\{R * \mathbf{Id}\} \begin{array}{c} \mathsf{oper}_1 \\ \mathsf{oper}_2 \end{array} \{R * \mathbf{Id}\}$$

denote the condition in Def. 4, it seems plausible that a rule of the following form should hold.

$$\frac{\{\Delta_P * R\} \begin{array}{c} c_1 \\ c_2 \end{array} \{\Delta_Q * R\} \quad \{P\}\ k\ \{Q\} \vdash \{A\}\ c\ \{B\}}{\{\Delta_A * R\} \begin{array}{c} c[c_1/k] \\ c[c_2/k] \end{array} \{\Delta_B * R\}} \ ,$$

where Δ_P denotes the binary relation $\mathbf{Id} \cap (P \times P)$, and $c[c_1/k]$ is c with all occurences of k replaced by c_1. We will investigate under which conditions a rule like the above is valid.[1] A rule like the above could also give syntactic criteria to decide when a program is a separation context for a module. This would be expressible using judgments of the form

$$\{P\}\ k\ \{Q\} \vdash \{A\}\ c\ \{B\}.$$

In our present setting we assume that we have the same heap model at abstract and concrete level. We will explore refinement where heap model at the concrete and abstract level are not necessarily the same.

[1]We are aware that the rule, as it is defined here is *not* valid.

6. CONCLUSIONS

There are at least two reasons why separation contexts are interesting. The first reason is that separation contexts do not dereference implementation modules' internals. This is formalized using $*$ and *precise relations* without any restrictions to reachability. The other reason is that once it is proved that a user program is a separation context with respect to some data structure, it is a separation context for all its refinements.

The work in this paper is a first step towards a theory of refinement in the setting of separation logic. We have mentioned a desirable rule that would pave the way for formal development of refinement results and correct programs. This also could be used for reasoning about equivalence of different implementations of data structures, that involve complex pointer manipulations.

Acknowledgements

This work was initiated by ideas of Prof. Peter O'Hearn and his influence was invaluable throughout the development of the paper. We would also like to thank Richard Bornat, Hongseok Yang and Cristiano Calcagno for insightful discussions. Noah Torp-Smith's research was partially supported by Danish Natural Science Research Council Grant 51–00–0315 and Danish Technical Research Council Grant 56–00–0309.

7. REFERENCES

[1] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control [extended abstract]. In *POPL 02*, 2002.

[2] L. S. C. Boyapati, B. Liskov. Ownership types for object encapsulation. In *POPL'03*.

[3] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *Proc. European Conference on Object-Oriented Programming*, June 2001.

[4] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined (resume). In B. Robinet and R. Wilhelm, editors, *ESOP 86, European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187 – 196. Springer Verlag, 1986.

[5] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12(583):576 – 580, 1969.

[6] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[7] J. Hogg. Islands: Aliasing protection in object-oriented languages.

[8] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The geneva convention on the treatment of object aliasing. *OOPS Messenger*, 1992.

[9] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of Programming Languages*, volume 28, London, 2001. ACM - SIGPLAN.

[10] P. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), June 1999.

[11] P. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, 2003.

[12] U. S. Reddy and H. Yang. Correctness of data representations involving heap data structures. In P. Degano, editor, *Proc. of the 12th European Symposium on Programming, ESOP 2003*, pages 223 – 237. Springer Verlag, 2003.

[13] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of Logic in Computer Science*, volume 17, pages 55 – 74, Copenhagen, July 2002. IEEE.