Refinement and Separation Contexts

Ivana Mijajlović[1], Noah Torp-Smith[2], and Peter O'Hearn[1]

[1] Queen Mary, University of London {ivanam, ohearn}@dcs.qmul.ac.uk
[2] IT University of Copenhagen noah@itu.dk

Abstract. A separation context is a client program which does not dereference internals of a module with which it interacts. We use certain "precise" relations to unambiguously describe the storage of a module and prove that separation contexts preserve such relations. We also show that a *simulation* theorem holds for separation contexts, while this is not the case for arbitrary client programs.

1 Introduction

Pointers wreak havoc with data abstractions [1–4]. To see why, suppose that a data abstraction uses a linked list in its internal representation; for example, an implementation of resource manager will use a free list. If a client program dereferences or otherwise accesses a pointer into this representation, then it will be sensitive to changes to the internal representation of the module. In theoretical terms, this havoc is manifest in the failure of classical "abstraction, logical relation, simulation" theorems for data abstraction. For example, the client program will behave differently if, say, the first rather than the second field in a cons cell is used to link together elements of a free list.

Data refinement is a method where one starts with an abstract specification of a data type and derives its concrete representation. Hoare introduced a method of refinement for imperative programs [5, 6]. His treatment of refinement assumes a static-scope based separation between the abstract data type and variables of the client. Pointers break those assumptions, as described above.

Previous approaches to abstraction in the presence of pointers [1, 3, 4, 7, 8] typically work by restricting what can point across certain boundaries. These solutions are limited and complex, and have difficulty coping with situations where pointers transfer between program components or where pointers across boundaries do exist without being dereferenced at the wrong time.

Separation logic [9], on the other hand, enables us to check code of a client for safety, even if there are pointers into the internals of a module [12]. It just ensures that pointers not be dereferenced at the wrong time, without permission.

This paper takes a first step towards bringing the ideas from 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. Our conditions for abstraction, based on a notion of "separation context", are considerably simpler than ones developed by Banerjee et al [3] and Reddy et al [4], and can easily handle examples with dangling pointers and examples of dynamic ownership transfer. We illustrate this with the nastiest problem we know of – toy versions of `malloc` and `free`.

The paper is organized as follows: we give some basic ideas and motivation in Section 2. In Section 3, we give relevant definitions regarding the programming language and relations on states. This enables us to define *unary separation contexts* in Section 4, 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 5, we prove a *simulation theorem* which is a cousin of a classic logical relations or abstraction theorem, and which fails when a context is not a separation context. We also give a condition which ensures that a separation context for an abstract module is automatically a separation context for all its refinements.

2 Basic Ideas

We will discuss two simple examples in which we consider two different pieces of client code. In both programs we assume that the client code interacts with the memory manager module through two provided operations, $\mathsf{new}()$ and $\mathsf{dispose}()$, for allocating and disposing memory, respectively. Suppose the module keeps locations available for allocating to a client, in a singly linked list.

To begin with we regard the program state as being separated into two parts, one of which belongs to a client, and the other which belongs to the module. The module's part always contains the free list. The statement $\mathsf{new}(x)$; takes a location from the free list puts it into x; at this point we regard the boundary between the client and module states as shifting: the ownership of the cell has transferred from the module to the client, so that the separation between client and module states is dynamic rather than determined once-and-for-all before a program runs. Similarly, when a client disposes a location we regard the ownership of that location as transferring from the client to the module. The concept of "ownership" here is simple: at any program point we regard the state as being separated into two parts, and ownership is just membership in one or the other component of the separated states.

Now, some programs respect this concept of separation while others do not. Consider the following client code.

$$\mathsf{new}(x); \text{do something with } x; \mathsf{dispose}(x); \mathsf{dispose}(x)$$

This simple program behaves very badly – it disposes the same location twice. This is possible because after disposing the location pointed to by x the first time, x holds the value of the location. Depending on the implementation of $\mathsf{dispose}$, this code could destroy the structure of the free list, and might eventually cause a program crash. This program contradicts our assumption of separation: the second $\mathsf{dispose}(x)$ statement accesses a cell which the client does not own, since it was previously transferred to the module.

In fact, *any* attempt to use the location after first $\mathsf{dispose}$ will contradict separation, say if we replace the second $\mathsf{dispose}$ by a statement $[x] := 42$ that mutates x's location. And both cases contradict abstraction. For instance, if the manager uses the $[x]$ field as a pointer to the next node in the free list, then $[x] := 42$ will corrupt the free list, but if the manager uses a different representation of the free list, corruption might not occur: depends whether or not it is representation-dependent.

In contrast, the following code obeys separation: the client code reads and writes to its own part, and disposes only a location which belongs to it.

$$\mathsf{new}(x); [x] := 15; y := [x]; \mathsf{dispose}(x)$$

The issue here is not exclusive to low-level programming languages. In a garbage collected language thread and connection pools are sometimes used to avoid the overhead of creating and destroying threads and database connections (such as when in a web server). Then, a thread or connection id should not be used after it has been returned to a pool, until it has been doled out again.

In the formal development to follow a "separation contexts" will be a piece of client code together with a precondition which ensures respect for separation.

3 Preliminary Definitions

In this section, we give relevant definitions regarding the storage model and relations in it. We give a programming language and its semantics.

Storage Model. We describe our models in an abstract way, which will allow various realizations of "heaps". We assume a countably infinite set Var of variables given. Let $S : \mathsf{Var} \to \mathsf{Val}$ be the set of *stacks* (that is, finite, partial maps from variables to values), and let H be a set of *heaps*, where we just assume that we have a set with a partial commutative monoid structure $(H, *, e)$. In effect, our development is on the level of the abstract model theory of BI [10], rather than the single model used in separation logic [11, 9]. We assume that $*$ is injective in the sense that for each h, the partial function $h * - : H \rightharpoonup H$ is injective. The set of *states* is the set of stack-heap pairs.

The subheap order $\sqsubseteq$ is induced by $*$ in the following way

$$h_1 \sqsubseteq h_2 \iff \exists h_3 . h_1 * h_3 h_2 .$$

Two heaps h_1 and h_2 are disjoint, denoted $h_1 \# h_2$, if $h_1 * h_2$ is defined.

We will often take H to be a set of finite partial functions

$$H = \mathsf{Ptr} \rightharpoonup_{\mathit{fin}} \mathsf{Val}, \text{ where } \mathsf{Ptr} = \{0, 1, 2, \ldots\} \quad \mathsf{Val} = \{\ldots, -1, 0, 1, \ldots\}.$$

The combination $h * h'$ of two such heaps is defined only when they have disjoint domains, in which case it is the union of the graphs of the two functions. We will not restrict ourselves to this (RAM) model, but will assume it in examples unless stated differently.

Separation logic. Separation logic is an extension of Hoare logic, where *heaps* have been added to the storage model. The usual assertion language of Hoare logic is extended with assertions that express properties about heaps

$$A, B ::= \mathsf{emp} \mid e_1 \mapsto e_2 \mid A * B \mid \mathsf{T} \mid \forall_* p \in m.\ A \mid \ \cdots .$$

The first 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. The fourth is true for any state, and the last assertion form is an iterated separating conjunction over a finite set. The semantics of assertions is given by a judgement $s, h \models A$ which asserts that the assertion A holds in the state (s, h). More about separation logic can be found in [9].

Unary relations. Certain special properties are used to identify the heap portion owned by a module [12].

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 integers. The predicate $\mathsf{list}(\alpha, x)$ is defined inductively on the sequence α by

$$\mathsf{list}(\varepsilon, x) \overset{\text{def}}{=} x = \mathit{nil} \wedge \mathsf{emp}, \qquad \mathsf{list}(a \cdot \alpha, x) \overset{\text{def}}{=} x = a \wedge \exists y.\ x \mapsto y * \mathsf{list}(\alpha, y)$$

where ε represents the empty sequence and $\cdot$ conses an element a onto the front of a sequence α. This predicate says that x points to a non-circular singly-linked list whose addresses are the sequence α (this is called a "Bornat list" in [9]). For any given s, h, there can be at most one subheap which satisfies $\mathsf{list}(\alpha, x)$, consisting of the cells in α. Generally, a precise relation gives you a way to "pick out the relevant cells".

We define the *separating conjunction of unary relations* $M, M' \subseteq S \times H$ by

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

Taking into account that $*$ is injective, a precise relation M induces a unique splitting of a state (s, h). We write (s, h_M) for the substate of (s, h) uniquely described by M, if it exists. Otherwise, $(s, h_M) = e$, the unit.

The Model. Our model will use a simple language with two kinds of atomic operations: the client operations and the module operations. The denotation of client commands will be given by functions $f : (S \times H) \to (S \times H) \uplus \{\mathit{wrong}\}$, and the denotation of module operations will be given by binary relations $t \subseteq (S \times H) \times (S \times H) \uplus \{\mathit{wrong}\}$. The special state *wrong* results when a program illegally accesses storage beyond the current heap. We presume there is a fixed set of module variables Var_M, which are never changed by the client:

$$\forall x \in \mathsf{Var}_M . \begin{array}{l} f(s, h) = \mathit{wrong} \Leftrightarrow \forall v . f(s \backslash \{x \mapsto v\}, h) = \mathit{wrong} \ \text{ and} \\ f(s, h) = (s', h') \Leftrightarrow \forall v . f(s \backslash \{x \mapsto v\}, h) = (s' \backslash \{x \mapsto v\}, h'). \end{array}$$

For a unary relation on states M, we write M_{wrong} to denote $M \cup \{wrong\}$. We will write $(s, h)[t](s', h')$ to denote that the states (s, h) and (s', h') are in the binary relation t.

The relation $M \subseteq S \times H$ is said to be *preserved* by a function f (respectively relation t) on states, if for all $(s, h), (s', h')$, such that state (s, h) is in M and $f(s, h) = (s', h')$ (respectively $(s, h)[t](s', h')$), imply $(s', h') \in M_{wrong}$.

The reader will have recognized an asymmetry in our model: client primitive operations are required to be deterministic, while in module operations non-determinism is allowed. One effect of this is that, when frame conditions are imposed later, the client operations will not be able to do any allocation; allocation will have to be viewed as a module operation. Technically, the determinism restriction is needed for our simple simulation theorem.

Local Functions and Relations. We will consider functions and relations on states that access resources in a local way. More formally, we say that a function $f : (S \times H) \to (S \times H) \uplus \{wrong\}$ (relation $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$) is *local* [12] if it satisfies the following properties

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

The properties are the ones needed for soundness of the Frame Rule of separation logic; see [13]. We will only consider local functions and relations.

Programming Language. The programming language is an extension of the simple while-language with a finite set of atomic client operations f_j $(j \in J)$ and a finite set of module operations oper_i, $i \in I$. The syntax of the *user language* is

$$\begin{aligned} c_{user} &::= \mathsf{f}_j,\ j \in J \mid \mathsf{oper}_i,\ i \in I \mid c_1; c_2 \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \mid \textbf{while } e \textbf{ do } c, \\ e &::= int \mid var \mid e + e \mid e \times e \mid e - e, \quad int \in \mathsf{Int},\ var \in \mathsf{Var}, \\ \mathsf{Int} &= \{\ldots -1, 0, 1, \ldots\},\ \mathsf{Var} = \{x, y, \ldots\},\ I, J - \text{finite indexing sets.} \end{aligned}$$

The expressions used in the language do not access heap storage. Commands such as $x := e$, $[e_1] := e_2$, $x := [e]$, etc. are examples of atomic operations.

The semantics of the language is parameterized by a precise relation M and a collection $(oper_i)_{i \in I}$ of binary relations that preserve $M * \mathsf{T}$. It defines a big-step transition relation $\rightsquigarrow \subseteq (c_{user} \times (S \times H)) \times ((S \times H) \uplus \{wrong, av\})$ on configurations, where av denotes a state in which client code illegally accesses the heap storage owned by the module, and will be referred to as "access violation". The operational semantics of the language is given in Table 1. State $(s, h_M) \in M$ denotes the substate of (s, h) uniquely determined by relation M in the second rule. What is left over, (s, h_U), is the client's state. K denotes an element of $(S \times H) \uplus \{av, wrong\}$.

Table 1. Operational semantics

$$(s,h) = (s,h_M) * (s,h_U)$$

$$\frac{f_j(s,h) = (s',h')}{\mathsf{f}_j, s, h \rightsquigarrow s', h'} \quad \frac{f_j(s,h) \neq \mathit{wrong} \quad f_j(s,h_U) = \mathit{wrong}}{\mathsf{f}_j, s, h \rightsquigarrow \mathit{av}} \quad \frac{f_j(s,h) = \mathit{wrong}}{\mathsf{f}_j, s, h \rightsquigarrow \mathit{wrong}}$$

$$\frac{(s,h)[\mathit{oper}_i](s',h')}{\mathsf{oper}_i, s, h \rightsquigarrow s', h'} \quad \frac{(s,h)[\mathit{oper}_i]\mathit{wrong}}{\mathsf{oper}_i, s, h \rightsquigarrow \mathit{wrong}} \quad \frac{c_1, s, h \rightsquigarrow s', h' \quad c_2, s', h' \rightsquigarrow K}{c_1; c_2, s, h \rightsquigarrow K}$$

$$\frac{c_1, s, h \rightsquigarrow \mathit{wrong}}{c_1; c_2, s, h \rightsquigarrow \mathit{wrong}} \quad \frac{\llbracket e \rrbracket s = 0}{\textbf{while } e \textbf{ do } c, s, h \rightsquigarrow s, h} \quad \frac{\llbracket e \rrbracket s \neq 0 \quad c; \textbf{while } e \textbf{ do } c, s, h \rightsquigarrow K}{\textbf{while } e \textbf{ do } c, s, h \rightsquigarrow K}$$

$$\frac{c_1, s, h \rightsquigarrow \mathit{av}}{c_1; c_2, s, h \rightsquigarrow \mathit{av}} \quad \frac{\llbracket e \rrbracket s \neq 0 \quad c_1, s, h \rightsquigarrow K}{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, s, h \rightsquigarrow K} \quad \frac{\llbracket e \rrbracket s = 0 \quad c_2, s, h \rightsquigarrow K}{\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, s, h \rightsquigarrow K}$$

4 Unary Separation Contexts

An essential point in the semantics in Table 1 is the way that module state is subtracted when client operations f_j are performed. If a client operation does not go wrong in a global state, but goes wrong when the module state is subtracted, we judge that this was due to an attempt to access the module's state; in the semantics this is rendered as an access violation, and a separation context is then a program (with a precondition) that does not lead to access violation.

Definition 2. *Let $M \subseteq S \times H$ be a precise unary relation, let P be a unary predicate on states, and for $i \in I$ let $\mathit{oper}_i \subseteq (S \times H) \times (S \times H) \uplus \{\mathit{wrong}\}$ preserve relation $M * \top$. A program c is a* unary separation context *for M, P and $(\mathit{oper}_i)_{i \in I}$ if for all executions and all $(s,h) \in M * P$ $c, s, h \not\rightsquigarrow \mathit{av}$.*

The idea is that M describes the heap storage owned by the module, and a separation context will never access that storage. Separation contexts preserve the resource invariant of a module because they change storage owned by the module only through the provided operations.

Theorem 1. *Let $M \subseteq S \times H$ be a precise relation, let P be a unary predicate on states, and for $(i \in I)$ let $\mathit{oper}_i \subseteq S \times H \times (S \times H) \uplus \{\mathit{wrong}\}$ preserve $M * \top$, and let c be a separation context for M, P and $(\mathit{oper}_i)_{i \in I}$. Then for all such P and all states (s,h) and (s',h'), if $(s,h) \in M * P$, and $c, s, h \rightsquigarrow s', h'$, then $(s', h') \in (M * \top)_{\mathit{wrong}}$.*

Separation Context Examples. We now revisit the ideas discussed in Section 2 in our more formal setting. In order to specify the operations of the memory manager module, we make use of the "greatest relation" for the specification $\{P\}\mathsf{oper}\{Q\}[X]$, which is the largest local relation satisfying a triple $\{P\} - \{Q\}$ and changing only the variables in the set X. It is similar to the "generic commands" introduced by Schwarz [14] and the "specification statements" studied in the refinement literature, but adapted to work with locality conditions in [12].

The predicate $\exists\alpha.\mathsf{list}(\alpha, ls)$ describes the free list, and we choose it as the M component in the definition of a separation context. The operations $\mathsf{new}(x)$ and $\mathsf{dispose}(x)$ are the greatest relations satisfying the following specifications.

$$\begin{array}{ll}\mathsf{new}_C(x): & \{\mathsf{list}(a\cdot\alpha, ls)\} - \{\mathsf{list}(\alpha, ls) * x \mapsto a\}[x, ls] \\ & \{\mathsf{list}(\varepsilon, ls)\} - \{\mathsf{list}(\varepsilon, ls) * x \mapsto -\}[x, ls] \\ \mathsf{dispose}_C(x): & \{\mathsf{list}(\alpha, ls) * x \mapsto a\} - \{\mathsf{list}(a\cdot\alpha, ls)\}[ls]\end{array}$$

For future reference, we will call this the *concrete* interpretation of the memory manager module. With these definitions we can judge whether a program (together with a precondition) is a separation context.

Consider the following three programs

$\mathit{Program}_1$:	$\mathit{Program}_2$:	$\mathit{Program}_3$:
$\mathsf{new}(x)$;	$\mathsf{dispose}(x)$;	$[81] := 42$
$[x] := 47$;	$[x] := 47$;	
$\mathsf{dispose}(x)$;		

We indicate whether a program, together with a precondition, is a separation context in the following table.

Context	Separation context?
$\{\mathsf{emp}\}$ $\mathit{Program}_1$	$\surd$
$\{x \mapsto -\}$ $\mathit{Program}_2$	$\surd$
$\{\mathsf{emp}\}$ $\mathit{Program}_2$	$\times$
$\{81 \mapsto -\}$ $\mathit{Program}_3$	$\surd$
$\{\mathsf{emp}\}$ $\mathit{Program}_3$	$\times$

Most of the entries are easy to explain, and correspond to our informal discussion from earlier. The last one, though, requires some care. For, how do we know that $[81] := 42$ interferes with the free list? The answer is that we do not. It might or might not be the case that location 81 is in the free list, at any given point in time. But, the notion of separation context is fail-safe: if there is *any* possibility that 81 is in the free list, on any run, then the program is judged not to be a separation context. And we can easily construct an example state where 81 is indeed in the free list. On the other hand in the second-last entry the precondition $81 \mapsto -$ ensures that 81 cannot be in the free list. This is because of the use of $*$ to separate the module and client states.

5 Refinement and Separation

In this section we first introduce precise binary relations and the separating conjunction of binary relations. We give a definition of refinement and prove a binary relation-preservation theorem.

Let $R \subseteq (S_0 \times H_0) \times (S_1 \times H_1)$ be a binary relation. We say that R is *precise*, if each of its two projections on the corresponding set of states is precise. Formally, for any state $(s_i, h_i) \in (S_i \times H_i)$ there is at most one $h_i' \sqsubseteq h_i$ such that there exists a state $(s_{1-i}, h_{1-i}) \in (S_{1-i} \times H_{1-i})$ such that $(s_i, h_i')[R](s_{1-i}, h_{1-i})$, for i=0,1.

We illustrate precise binary relations with an example. Suppose we have two different implementations of a memory 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 the list predicate $\mathsf{list}(\alpha, ls)$, defined in Section 3. Now, a precise binary relation

$$R = \left\{ ((s,h),(s',h')) \middle| \begin{array}{l} (s,h \models \forall_* p \in f.\ p \mapsto -) \wedge (s',h' \models \mathsf{list}(\alpha, ls)) \wedge \\ set(\alpha) = s(f) \end{array} \right\},$$

where $set(\alpha)$ is defined as the set of pointers in the sequence α, relates these two implementations. Relation R relates pairs of states, such that one state can be described as a set of different pointers, while the other is determined by the list of exactly the pointers that appear in the mentioned set.

For two binary relations $R, R' \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ on states, we define their separating conjunction [4] as

$$R * R' = \left\{ ((s_1,h_1),(s_2,h_2)) \middle| \begin{array}{l} \exists h_1', h_1'', h_2', h_2''.\ h_1 = h_1' * h_1'' \ \wedge h_2 = h_2' * h_2'' \wedge \\ (s_1,h_1')[R](s_2,h_2') \ \wedge \ (s_1,h_1'')[R'](s_2,h_2'') \end{array} \right\}$$

Similarly to the unary case, for a binary relation on states R we will write R_{wrong} to denote $R \cup \{(wrong, wrong)\}$.

5.1 Refinement and Separation Contexts

In this section, we formally express what it means for one module to be a *refinement* (or an implementation) of another. For simplicity, we 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 previous work on refinement [6], our definition of refinement is called an upward simulation.

In the following, we will take H_1, H_2 and H_3 to be three (in general different, but possibly equal) heap models, assuming that $(H_1, *_1, e_1)$, $(H_2, *_2, e_2)$ and $(H_3, *_3, e_3)$ have partial commutative monoid structure.

Definition 3. *Let $Z \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ be a binary relation. We define $oper^2 \subseteq (S_2 \times H_2) \times (S_2 \times H_2) \uplus \{wrong\}$ to be a* refinement *of $oper^1 \subseteq (S_1 \times H_1) \times (S_1 \times H_1) \uplus \{wrong\}$ with respect to Z, if*

- *for all states $(s_1,h_1),(s_2,h_2),(s_2',h_2')$, such that $(s_1,h_1)[Z](s_2,h_2)$ 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 $(s_1',h_1')[Z](s_2',h_2')$, and*
- *for all states $(s_1,h_1),(s_2,h_2)$, such that $(s_1,h_1)[Z](s_2,h_2)$ if $(s_2,h_2)[oper^2]wrong$ then $(s_1,h_1)[oper^1]wrong$.*

In order to prove the relation preservation theorem, we need to instantiate the refinement relation to a separating conjunction of binary relations, $R, Q \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$. We assume that the following properties hold:

- R is precise

- Q is such that for any two states (s_1, h_1), (s_2, h_2) related by Q and a guard (condition of **if** and **while** statements) b, $s_1(b) \Leftrightarrow s_2(b)$
- $\mathit{oper}^2 \subseteq (S_2 \times H_2) \times (S_2 \times H_2) \uplus \{\mathit{wrong}\}$ is a refinement of $\mathit{oper}^1 \subseteq (S_1 \times H_1) \times (S_1 \times H_1) \uplus \{\mathit{wrong}\}$ with respect to $R * Q$
- We denote a pair (f_j^1, f_j^2) by $\mathbf{f}_j$. Pair $\mathbf{f}_j$,is such that it maps Q-related states to Q_{wrong}-related states.

The role of R is to relate abstract and concrete subheaps which belong to the module, while Q relates the clients' parts of the heaps.

> **Simulation Theorem (Informally):** Suppose we have two instantiations of a client program, which use calls to concrete and abstract module operations respectively, related by a refinement relation. Then, provided both of these two instantiations are separation contexts with respect to the corresponding modules, the effect of the concrete computation can be tracked by the abstract.

Stating this more formally requires some notation. For a program c, let $c_i \subseteq (S_i \times H_i) \times (S_i \times H_i) \uplus \{\mathit{wrong}\}$ 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 projection of R onto $(S_i \times H_i)$. Q_P denotes $Q \cap (P \times Q(P))$, where Q is a binary relation on states, P is a unary relation on states, and $Q(P)$ is their composition.

Theorem 2 (Simulation Theorem). *Let R, Q, oper^i, c, c_i for $i = 1, 2$, be as above, and let $P \subseteq Q_1$ be a unary relation on states. Let c_1 be a separation context for R_1, P and oper^1, and let c_2 be a separation context for $R_2, Q(P)$ and oper^2. Then for all such P and all $(s_1, h_1), (s_2, h_2)$, (s_2', h_2') if (s_1, h_1) $[R * Q_P]$ (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 $(s_1', h_1')[R * Q](s_2', h_2')$.*

The crucial assumption is that c_1 and c_2 are separation contexts for the given modules and preconditions, and without this condition the theorem fails.

One shortcoming is that we have to check whether both c_1 and c_2 are separation contexts to apply Theorem 2. From the point of view of program development it would be better if we knew that when we had a separation context for an abstract module then it would automatically remain a separation context for all its refinements. Then the check could be done once and for all. In order to realize this aim, an extra concept is needed: safety. A safe separation context is a client which does not touch any storage not in its possession.

Definition 4 (Safe Separation Context). *Let c be a separation context for the precise relation M, precondition P and family of operations $(\mathit{oper}^i)_{i \in I}$. Program c is a* safe separation context *for M, P, $(\mathit{oper}^i)_{i \in I}$ if for all executions and all states $(s, h) \in M * P$, $c, s, h \not\rightsquigarrow \mathit{wrong}$.*

Theorem 3. *Let R, Q, oper^i, c, c_i for $i = 1, 2$ be as in Theorem 2, and let $P \subseteq Q_1$ be a unary relation on states. If c_1 is a safe separation context for R_1, P and oper^1, then c_2 is a safe separation context for $R_2, Q(P)$ and oper^2.*

Safe Separation Context Example. To see the role of the concept of safety, consider an *abstract* version of the memory manager procedures, the"magical malloc module". It is magical in that the module does not own any locations at all, producing them as if out of thin air. (In implementation terms, the thin air is like a call to a system routine such as `sbrk`.) Therefore, the resource invariant of the module, M in our formal setup, is the predicate emp. Now, we define the abstract operations $\mathsf{new}_A(x)$ and $\mathsf{dispose}_A(x)$ as the greatest relations satisfying the following specifications.

$$\mathsf{new}_A(x) : \{\mathsf{emp}\} - \{x \mapsto -\}[x], \quad \mathsf{dispose}_A(x) : \{x \mapsto -\} - \{\mathsf{emp}\}[\,]$$

This is the meaning of allocation and disposal that is usually presumed in separation logic. Because the manager owns no storage whatsoever, there is no way for a client to trample on it. As a result, *every* client program is a separation context for this abstract module.

But, not every context is safe. Consider the context

$$\{\mathsf{emp}\}\ [81] := 42$$

from the Separation Context Examples in Section 4. It immediately goes wrong, and so is not safe. Recall also that in the more concrete semantics, from the same section, this is not even an ordinary separation context.

This shows the import of Theorem 3. If we know that our context is safe in the abstract setting, then this ensures that module internals will not be tampered with in refinements. Put another way, module tampering in a concrete implementation can show up as going wrong in the abstract, and the concept of safe separation context protects against this.

Refinement Examples. Here, we illustrate refinement relations between different interpretations of the memory manager module with two examples.

To define the refinement relations we borrow some notation from relational separation logic [15]. Let $S_1 \times H_1$ and $S_2 \times H_2$ be two state spaces. Let $P \subseteq S_1 \times H_1$, $Q \subseteq S_2 \times H_2$ and $R \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ be predicates. We let

$$\begin{pmatrix} P \\ Q \end{pmatrix} \wedge R \qquad \text{denote} \qquad \{(s_1, h_1), (s_2, h_2) \mid (s_1, h_1 \models P \ \wedge \ s_2, h_2 \models Q) \ \wedge \ R\}.$$

The first example involves refinement between the *abstract* and the *concrete* interpretations of the memory manager module. We have already specified both interpretations, the abstract – in the Safe Separation Context Example above, and the concrete – in the ordinary Separation Context Example from Section 4.

The refinement relation Z_{AC} between these two interpretations is a separating conjunction of binary relations R_{AC} and Q_{AC}. These are given by

$$R_{AI} = \begin{pmatrix} \mathsf{emp} \\ \exists \alpha.\ \mathsf{list}(\alpha, ls) \end{pmatrix} \qquad Q_{AI} = \mathbf{Id}.$$

Relation R_{AI} relates modules' states of the two interpretations and is basically the relation between their resource invariants. Relation Q_{AC} relates the clients' states and is the identity relation.

In the second example, we introduce the *intermediate* version of the memory manager module. We do this for two reasons. First, this illustrates the use of two different heap models, as allowed in our formal setting. Second, considering refinement between the intermediate and the concrete interpretations requires a subtler refinement relation.

On the intermediate level, the intention is to keep locations owned by the module in a set, without committing to the representation of the set. If this set becomes empty, we call a "system routine" (like `sbrk`) to get a new location.

For this interpretation, we assume the following heap model. Let Loc be an infinite set of locations. A heap will be an element of the Cartesian product $\mathcal{P}_{fin}(Loc) \times H_1$, where $(H_1, *_1, e_1)$ is the partial commutative monoid of the RAM model. We say that a pair (N, h) from this product is *well-defined* if $N \cap dom(h) = \emptyset$. The intermediate heap model H consists of these well-defined elements. Two intermediate heaps (N_1, h_1) and (N_2, h_2) are disjoint, $(N_1, h_1)\#_1(N_2, h_2)$, whenever $N_1 \cap N_2 = \emptyset$ and $N_1 \cap dom(h_2) = \emptyset$ and $N_2 \cap dom(h_1) = \emptyset$ and $dom(h_1) \cap dom(h_2) = \emptyset$. We define $*$ between two heaps by

$$(N_1, h_1) * (N_2, h_2) = \begin{cases} (N_1 \cup N_2, h_1 *_1 h_2), & \text{if } (N_1, h_1)\#_1(N_2, h_2) \text{ and} \\ & (N_1, h_1), (N_2, h_2) \text{ well defined} \\ \mathsf{undefined}, & \text{otherwise} \end{cases}$$

We say that $s, (N, h) \models act(p)$ if and only if $p \in N$. The resource invariant can be described with $\forall_* p \in f.\ act(p)$, where f is a set variable. We now define operations $\mathsf{new}_I(x)$ and $\mathsf{dispose}_I(x)$ as the greatest relations satisfying the specifications

$\mathsf{new}_I(x):$ $\{\forall_* p \in f.\ act(p) \land f = Y \neq \emptyset\} - \{(\forall_* p \in f.\ act(p) \land f = Y \setminus \{x\})*$ $x \mapsto -\}[x, f]$
$\{\forall_* p \in f.\ act(p) \land f = \emptyset\} - \{(\forall_* p \in f.\ act(p) \land f = \emptyset) * x \mapsto -\}[x]$
$\mathsf{dispose}_I(x):$ $\{(\forall_* p \in f.\ act(p) \land f = Y) * x \mapsto -\} - \{\forall_* p \in f.\ act(p) \land$ $f = Y \cup \{x\}\}[f]$

The variable Y is used to keep track of the initial contents of f, similarly to how α was used in the concrete interpretation. Note that it is not altered because it is not in the modifies set, a set of actual locations owned by the module. We intend that $\mathsf{new}_I(x)$ is the greatest relation satisfying both stated specifications.

Now, the refinement relation Z_{IC} between intermediate and concrete relations is a separating conjunction of binary relations R_{IC} and Q_{IC} given by

$$R_{IC} = \begin{pmatrix} f \\ \mathsf{list}(\alpha, ls) \end{pmatrix} \land set(\alpha)val(f) \qquad Q_{IC} = \mathbf{Id},$$

where $val(f)$ is the value of set variable f. It can be verified that the operations preserve these relations as required in the definition of refinement.

In these two examples we have not exercised the possibility of using a non-identity relation to relate the abstract and concrete client states. A good such example compares two implementations of a buffer, one of which copies two values where the other passes a single pointer to the two values. It is omitted here for space reasons.

Directions for future work. There are several directions for further work. First, we have, for simplicity, considered the interaction between a client and a single module; in the future we plan on investigating independence between modules. Second, it would be worthwhile to consider multiple-instance classes (e.g. [3]); here we have, in effect, a single single-instance class. It would also be important to remove the restriction of determinism, imposed to the client operations. Finally, we would like to use the model to make the connection back to logic. Perhaps a relational version of the hypothetical frame rule, or the modular procedure rule, from [12] can be formulated, borrowing from Yang's relational separation logic [15].

Acknowledgements We would like to thank Hongseok Yang, Josh Berdine, Richard Bornat and Cristiano Calcagno for invaluable discussions and anonymous referees for their careful comments. 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. Mijajlović and O'Hearn were supported by the EPSRC.

References

1. Hogg, J.: Islands: Aliasing Protection In Object-Oriented Languages. *OOPSLA'91*
2. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The Geneva Convention On The Treatment of Object Aliasing. *OOPS Messenger* (1992)
3. Banerjee, A., Naumann, D. A.: Representation Independence, Confinement and Access Control [extended abstract]. *29th POPL.* (2002)
4. Reddy, U. S., Yang, H.: Correctness of Data Representations involving Heap Data Structures. *Proceedings of ESOP* .Springer Verlag (2003) 223–237
5. Hoare, C. A. R.: Proof of Correctness of Data Representations. *Acta Informatica.* Vol. 1. (1972) 271–281
6. He, J., Hoare, C. A. R., Sanders, J. W.: Data Refinement Refined (Resume). *Proceedings of ESOP*. LNCS. Vol. 213. Springer Verlag (1986) 187–196
7. Clarke, D. G., Noble, J., Potter, J. M.: Simple Ownership Types for Object Containment. *Proceedings of ECOOP*. (2001)
8. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. *30th POPL.* (2003)
9. Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures. *Proceedings of 17th LICS.* (2002) 55–74
10. D. Pym, P. O'Hearn, H. Yang.: Possible Worlds and Resources: The Semantics of BI. *Theoretical Computer Science* 313(1) (2004) 257-305
11. Ishtiaq, S., O'Hearn, P. W.: BI as an Assertion Language for Mutable Data Structures. *28th POPL* (2001) 14-26
12. O'Hearn, P., Yang, H., Reynolds, J. C.: Separation and information hiding. *31st POPL.* (2004) 268–280
13. H. Yang, P. O'Hearn.: A semantic basis for local reasoning. In *Proceedings of FOSSACS'02* (2002) 402–416
14. J. Schwarz.: Generic Commands - A Tool for Partial Correctness Formalisms. *Comput. J.* 20(2) (1977) 151-155
15. Yang, H.: Relational Separation Logic. *Theoretical Computer Science* (to appear)