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 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 contexts in connection with refinement.

1 Introduction

Pointers wreak havoc with data abstractions [2–5]. 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 [6, 7]. 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 [2, 4, 5, 8, 9] 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 [10], on the other hand, enables us to check code of a client for safety, even though there may exist pointers into the internals of a module [11]. 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 model is considerably simpler than ones given in [4, 5], 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 define *safe separation contexts* and prove their useful property. We illustrate refinement with a detailed example, and conclude by giving pointers to future work.

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.

Consider the following client code.

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

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.

From this example, we can intuitively see that we can regard the program state as containing two parts, one of which belongs to a client, and the other which belongs to the module. The module's part contains a free list, and after allocating a location and putting it into x, client's state contains variable x and the location to which x points. Once a client has disposed a location, that location is no longer in the client's part of the program state, but rather transfered to a module's state. This is usually refered to as "ownership transfer". The problem with this client code then, is that it contradicts the separation between the state portions belonging to the client and the module.

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

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

[*Aside*. It is important to stress that the issue here is not exclusive to low-level programs written in a language like C. For instance, if a web server were programmed in a garbage collected language we would still perhaps use thread and connection pools to avoid the overhead of creating and destroying threads and database connections. Then, a thread or connection id should not be used after it has been returned to a pool, until is is subsequently doled out again.]

Using the fact that the current program state may be split in a way described above, and having in hand tools, such as separation logic, which enable us express that and maintain the splitting in a dynamic environment, we identify the class of programs which "behave well", and call them separation contexts.

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, parametrized over the relations just defined.

Storage Model. We will 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 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 $(H, *, e)$. We assume that $*$ is injective in the following sense: for each h, $h * - : H \rightharpoonup H$ is a partial injective function. 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 by $h_1 \# h_2$, if $h_1 * h_2$ is defined.

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

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

but we will not restrict ourselves to this (RAM) model. In our examples, we will assume the RAM model, unless stated differently.

Separation logic. Separation logic uses the pointer model of [12] for **BI** [13] 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 laguage.

If $h \# h'$, we define the combined heap $h * h'$ simply as the set-theoretic union of h and h'.

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

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

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 judgment $s, h \models A$ which asserts that the assertion A holds in the state (s, h). More about separation logic can be found in [10].

Unary relations. Certain special properties are used to identify the heap portion owned by a module [11].

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 predciate $\mathsf{list}(\alpha, i, j)$ is defined inductively on the sequence α by

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

where ε represents the emtpy sequence and $\cdot$ conses an element p onto the front of a sequence α. Note that if $s, h \models \mathsf{list}(\alpha, i, j)$, then there are no elements occurring twice in α. Now, if ls is a pointer to the beginning of the list, then $M = \{(s, h) \mid s, h \models \mathsf{list}(< 7, 49, 17 >, ls, nil)\}$ is precise. Generally, a precise relation gives you a way to "pick out the relevant cells".

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

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

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

For a unary relation on states M, we write M_{wrong} to denote $M \cup \{wrong\}$.

Local Functions and Relations. 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 $f : (S \times H) \to (S \times H) \uplus \{wrong\} \uplus \{av\}$ and the denotation of module operations 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 function f (relation t) if for all $(s, h), (s', h')$, $(s, h) \in M$ and $f(s, h) = (s', h')$ $((s, h)[t](s', h'))$, imply $(s', h') \in M_{wrong}$.

We will consider functions and relations on states that access resources in a local way. More formally, we say that a function $f : (S \times H) \to (S \times H) \uplus \{wrong\} \uplus \{av\}$ (relation $t \subseteq (S \times H) \times (S \times H) \uplus \{wrong\}$) is *local* [11] if it satisfies the following properties

- **Safety Monotonicity**: For all states (s, h) and heaps h_1 such that $h \# h_1$, if $f(s, h) \neq wrong$ $(\neg(s, h)[t]wrong)$, then $f(s, h * h_1) \neq wrong$ $(\neg(s, h * h_1)[t]wrong)$

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

The properties are those needed to prove the important Frame Rule from [12]. We will only consider local functions and relations in this paper.

Programming Language The programming language is an extension of the simple while-language with a finite set of atomic client operations $f_j : S \times H \to S \times H \uplus \{av, wrong\}$ denoted by f_j ($j \in J$) and a finite set of module operations $oper_i : (S \times H) \times (S \times H) \uplus \{wrong\}$ denoted by oper_i, $i \in I$. The *user language* has the following syntax:

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

The expressions used in the language do not access heap storage. Note that intention is to make atomic operations of the language deterministic and to let operations of the module be nondeterministic. Commands such as $x := e$, $[e_1] := e_2$, $x := [e]$, etc. are examples of atomic operations.

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 client code illegally accesses the storage beyond the current heap. Access violation is denoted by av and is a state in which client code illegally accesses the heap storage owned by the module. We use precise relations to keep track of this. The operational semantics of the language is given in Table 1. K denotes an element of $(S \times H) \uplus \{av\} \uplus \{wrong\}$.

Table 1. Operational semantics

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

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

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

$$\frac{c_1, s, h \rightsquigarrow wrong}{c_1; c_2, s, h \rightsquigarrow wrong} \qquad \frac{[\![e]\!]s = 0}{\textbf{while } e \textbf{ do } c, s, h \rightsquigarrow s, h} \qquad \frac{[\![e]\!]s = 1 \quad c; \textbf{while } e \textbf{ do } c, s, h \rightsquigarrow K}{\textbf{while } e \textbf{ do } c, s, h \rightsquigarrow K}$$

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

Remark 1. We are interested only in starting states which are related by $M * \mathsf{T}$, but the operational semantics is valid for commands starting in arbitrary states.

4 Unary Separation Contexts

In this section, we first define a separation context, and then prove the "relation-preservation" property of separation contexts. We end this secton with several examples, which illustrate the essence of separation contexts. The operational semantics above works by substracting the module state when client operations are performed, in the rule for f_j. This lets us judge when an access violation has occurred, and a separation context is then a user program (with a precondition) that does not access violate.

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

The intuitive meaning of this definition is that M describes the heap storage owned by the module, and client code which is a separation context will never access that storage. For example, in a RAM model, a separation context is allowed to have pointers to the module's heap, but not to dereference them.

Unary separation contexts preserve precise unary relations.

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 $oper_i \subseteq \ S \times H \times (S \times H) \uplus \{wrong\}$ preserve $M * \mathsf{T}$, and let c be a separation context for M, P and $(oper_i)_{i \in I}$. 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 $(s',h') \in (M * \mathsf{T})_{wrong}$.*

This property of separation contexts is very important. It directly addresses the problems of a client being dependent on a particular implementation of the module, described in Section 1. Separation contexts preserve the resource invariant of the module and change storage owned by the module only through the operations provided by the module.

Examples. We consider three client programs which interact with a memory manager module.

First we define the memory manager module. We will use the list predicate defined in Section 3 to describe the resource invariant of the module. The sequence α in this case records what pointers are held in the list. By $set(\alpha)$ we will denote the set of pointers in the sequence α. We define the operations $\mathsf{new}()$ and $\mathsf{dispose}()$ of the memory manager module, using the resource invariant.

$$\{\mathsf{list}(ls, a \cdot \alpha, nil)\}\ \mathsf{new}_C(x)\ \{\mathsf{list}(ls, \alpha, nil) * x \mapsto a\}$$
$$\{\mathsf{list}(ls, \varepsilon, nil)\}\ \mathsf{new}_C(x)\ \{\mathsf{list}(ls, \varepsilon, nil) * x \mapsto -\}$$
$$\{\mathsf{list}(ls, \alpha, nil) * x \mapsto -\}\ \mathsf{dispose}_C(x)\ \{\mathsf{list}(ls, \alpha, nil)\}$$

$Program_1$ below is an example of a separation context for the precondition emp and the resource invariant of the memory management module with which it interacts. It dereferences only locations that are owned by itself. Clearly, not all

programs are separation contexts. Suppose that the precondition for $Program_2$ is $x \mapsto 22$ and the resource invariant is $\mathsf{list}(ls, < 18, 42, 37 >, nil)$. The program first disposes a location pointed to by variable x and then, after giving up the ownership of the location, tries to dereference it, commiting the access violation. Let the precondition and the resource invariant be as in the previous example. Note that $Program_3$, even though it dereferences a location which is not in the current heap, and so goes *wrong*, is still a separation context; it does not dereference module internals.

$Program_1$:	$Program_2$:	$Program_3$:
$x := \mathsf{new}();$	$\mathsf{dispose}(x);$	$[81] := x$
$[x] := 47;$	$[x] := 47;$	
$\mathsf{dispose}(x);$		

5 Refinement and Separation

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

Let $R \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ be a binary relation. We say that R is *precise*, if each of its two projections on the corresponding set of states is precise:

- for any state $(s_1, h_1) \subseteq (S_1 \times H_1)$ there is at most one $h_1' \sqsubseteq h_1$ such that there exists a state $(s_2, h_2) \subseteq (S_2 \times H_2)$ such that $(s_1, h_1')[R](s_2, h_2)$, and
- for any $(s_2, h_2) \subseteq (S_2 \times H_2)$, there is at most one $h_2' \sqsubseteq h_2$ such that there is a state $(s_1, h_1) \subseteq (S_1 \times H_1)$ with $(s_1, h_1)[R](s_2, h_2')$.

Recall that precise unary relations uniquely identify a portion of a heap. Similarly, given two states and a precise binary relation R, R uniquely determines the corresponding substates of the two states.

We 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 $\mathsf{list}(\alpha, ls, \mathsf{end})$, defined in Section 3. Now, a precise binary relation R relating these two implementations is given by

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

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 $R, R' \subseteq (S_1 \times H_1) \times (S_2 \times H_2)$ on states, we define their separating conjunction [5] as

$$R * R' = \{((s_1,h_1),(s_2,h_2)) \mid \exists h_1', h_1'', h_2', h_2''.\ h_1 = h_1' * h_1'' \ \wedge h_2 = h_2' * h_2'' \ \wedge \\ (s_1,h_1')[R](s_2,h_2') \ \wedge \ (s_1,h_1'')[R'](s_2,h_2'')\}$$

As in 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 will formally express what it means for one module to be a *refinement* (or an implementation) 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 [7], our definition of refinement is called 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 $(H_1, *, e)$, $(H_2, *, e)$ and $(H_3, *, e)$ 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 $(s_1, h_1)[oper^1]wrong$.*

Using notation from [14] we let $\{Z\}\, \mathsf{oper}_2^1\, \{Z\}$ denote the conditions of Definition 3, i.e. we let this quadruple state that $oper^2$ is a refinement of $oper^1$ with respect to relation Z.

Proposition 1. *Let $oper^i \subseteq (S_i \times H_i) \times (S_i \times H_i) \uplus \{wrong\}$, $i = 1, 2, 3$, be such that $oper^2$ is a refinement of $oper^1$ with respect to binary relation V, and $oper^3$ is a refinement of $oper^2$ with respect to binary relation W. Then, $oper^3$ is a refinement of $oper^1$ with respect to a relation $V \circ W$.*

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

- R is a precise binary relation
- Q is such that for any two states (s_1, h_1), (s_1, h_1) related by Q and a guard (condition of **if** and **while** statements) b, $s_1(b) \Leftrightarrow s_2(b)$
- We assume that $oper^2 \subseteq (S_2 \times H_2) \times (S_2 \times H_2) \uplus \{wrong\}$ is a refinement of $oper^1 \subseteq (S_1 \times H_1) \times (S_1 \times H_1) \uplus \{wrong\}$ with respect to $R * Q$
- We denote a pair (f_j, f_j) by $\mathbf{f}_j$. Pair $\mathbf{f}_j : S \times H \to S \times H$, $j \in J$ is such that it maps Q-related states to Q_{wrong}-related states, and each of its components has the property that for all states (s, h) and heaps $h''\#h$ if $f_j(s, h) \neq wrong$ and $f_j(s, h) = (s', h')$ then $h''\#h'$ (Minimum resource property).

Note that 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.

We will also use the following notation. Let c be a program, and let $c_i \subseteq (S_i \times H_i) \times (S_i \times H_i) \uplus \{wrong\}$ 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 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 $(s_1', h_1')[R * Q](s_2', h_2')$.*

Note that the crucial assumption is that c_1 and c_2 are separation contexts for given modules and preconditions, and without this condition, the theorem fails. The theorem states that for any separation context c, the concrete denotation c_2 of c refines the abstract denotation c_1 of c.

We now introduce a *safe* separation context - a client which does not touch any storage not in its posession.

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

The following property gives us a criterion for "safe refinement". Namely, we specify the conditions under which, once we start with the abstract separation context for some abstract module, we may conclude that a concrete program is also a separation context with respect to a refinement of the module.

Lemma 1. *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. 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 $oper^2$.*

To illustrate why is it important to assume that the client program is a *safe* separation context in order to have the previous lemma work, take a look at the memory manager module. Suppose that on the abstract level we have magical malloc, which does not own any storage, and on the concrete a "free list" of locations owned by the module[3]. If the abstract client goes *wrong*, its refinement - the concrete client, may either go *wrong* or it may commit access violation. The possibility of commiting access violation forces us to conclude that the concrete program is not neccessarily a separation context. The extra condition that the client code may not go *wrong*, solves this problem.

[3] These two implementations are explained in more detail in subsection **Example**

Example. The example involves three implementations of `malloc`. One is completely abstract, one is more concrete, and the last uses a free-list, so is closest to a realistic implementation. 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 define the operations new and dispose using the specifications.

Before we proceed, we give a definition of the "greatest relation" [11], which we will use to specify each of the operations of the module.

For each specification $\{P\} - \{Q\}$, define $\mathsf{great}(P, Q)$

$$(s,h)[\mathsf{great}(P,Q)]wrong \iff^{def} (s,h) \notin P * \mathsf{T}$$
$$(s,h)[\mathsf{great}(P,Q)](s',h') \iff^{def} \forall h_P, h_1.(h_P * h_1 = h \ \wedge \ (s,h_P) \in P) \Longrightarrow \exists h'_Q.h'_Q \# h_1 \ \wedge \ h'_Q * h_1 = h' \ \wedge \ (s',h'_Q) \in Q.$$

For the abstract implementation (the "magical malloc"), we assume usual RAM model. The intention is that the abstract module does not own any of the locations, but when the client asks for a new location, the module also asks for a location from the system. When the client gives up a location, it is immediately returned to the system. Therefore, the resource invariant of the module is emp. Now, we define the abstract operations $\mathsf{new}_A(x)$ and $\mathsf{dispose}_A(x)$ as the greatest relations satisfying the following specifications.

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

For the intermediate implementation, we assume the following heap model. Let Loc be an infinite set of locations. We define a heap as a Cartesian product $H = \mathcal{P}_{fin}(Loc) \times H_1$, where $(H_1, *_1, e_1)$ is a partial commutative monoid of a RAM model. For a heap (M, h) we say it is *well defined* if $M \cap dom(h) = \emptyset$. Two heaps (M_1, h_1) and (M_2, h_2) are disjoint, $(M_1, h_1)\#_1(M_2, h_2)$, whenever $M_1 \cap M_2 = \emptyset$ and $M_1 \cap dom(h_2) = \emptyset$ and $M_2 \cap dom(h_1)$ and $dom(h_1) \cap dom(h_2) = \emptyset$. We define $*$ between two heaps by

$$(M_1,h_1) * (M_2,h_2) = \begin{cases} (M_1 \cup M_2, h_1 *_1 h_2), & \text{if } (M_1,h_1)\#_1(M_2,h_2) \text{ and} \\ & (M_1,h_1),(M_2,h_2) \text{ well defined} \\ \mathsf{undefined}, & \text{otherwise} \end{cases}$$

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.

We say that $s, (M, h) \models act(p)$ if and only if $p \in M$. The resource invariant can be described with $\forall_* p \in f.\ act(p)$. Let Y denote a set of actual locations owned by the module. We now define operations $\mathsf{new}_I()$ and $\mathsf{dispose}_I()$ as the greatest relations satisfying specifications

$$\begin{array}{ll} \mathsf{new}_I(x): & \{\forall_* p \in f.\ act(p) \wedge f = Y \neq \emptyset\} - \{(\forall_* p \in f.\ act(p) \wedge f = Y \setminus \{x\}) \\ & * x \mapsto -\} \\ & \{\forall_* p \in f.\ act(p) \wedge f = \emptyset\} - \{(\forall_* p \in f.\ act(p) \wedge f = \emptyset) * x \mapsto -\} \\ \mathsf{dispose}_I(x): & \{(\forall_* p \in f.\ act(p) \wedge f = Y) * x \mapsto -\} - \{\forall_* p \in f.\ act(p) \wedge \\ & f = Y \cup \{x\}\} \end{array}$$

Note that $\mathsf{new}_I()$ is the greatest relation satisfying both stated specifications.

The concrete implementation is the more realistic one. The module keeps a *free-list* of the cells that have been disposed by the client. When the client calls new, it returns the first element in the list, if it is not empty. If it is empty, the module calls a system routine like in the intermediate implementation. The resource invariant for this implementation is the variant of the well-known list predicate defined in Section 3.

The operations $\mathsf{new}_C()$ and $\mathsf{dispose}_C$ are given as greatest relations satisfying the following specifications. As in the intermediate representation, $\mathsf{new}()$ has to satisfy both stated specifications.

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

Before we define refinement relations, we introduce the notation that will be used in further presentation[4]. 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 be predicates. We will let

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

As all the resource invariants in our example are precise, they induce a unique splitting of the heap.

Now, we give the refinement relations. First we define the refinement relation between the intermediate and abstract implementations. The binary relations R_{AI} and Q_{AI} which relate modules' and clients' heaps, respectively, are given by

$$R_{AI} = \binom{\mathsf{emp}}{f} \qquad Q_{AI} = \mathbf{Id}$$

The concrete implementation is a refinement of the intermediate one with respect to the following relation. Denote by $val(f)$ the value of set variable f. The binary relations R_{IC} and Q_{IC} are given by

$$R_{IC} = \binom{f}{\mathsf{list}(\alpha, ls, nil)} \wedge set(\alpha) = val(f) \qquad Q_{IC} = \mathbf{Id}$$

In this example we have not exercised the possibility of using a non-identity relation to relate the abstract and concrete client states. A good such example of this 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.

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, we would like to

[4] We were influenced by [14].

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 [11] can be formulated, borrowing from Yang's relational separation logic [14].

Finally, it would be worthwhile to provide a formal comparison between the work here and that on confinement [4, 5, 8]. The idea would be to use a very small language in which confinement and separation could both be formulated. Our (biased) view is that confinement is a *mechanism* for ensuring abstraction in the presence of pointers, but separation is the *reason*. One way to probe this question would be to attempt to show a result to the effect that confinement provides sufficient but not necessary conditions for ensuring separation.

Acknowledgements We would like to thank Hongseok Yang, Josh Berdine, Richard Bornat and Cristiano Calcagno for invaluable 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.

References

1. O'Hearn, P., Yang, H., Reynolds, J. C.: Separation and information hiding. Unpublished Manuscript. Extended version of the POPL'04 paper with the same title
2. Hogg, J.: Islands: Aliasing Protection In Object-Oriented Languages. OOPSLA'91
3. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The Geneva Convention On The Treatment of Object Aliasing. OOPS Messenger (1992)
4. Banerjee, A., Naumann, D. A.: Representation Independence, Confinement and Access Control [extended abstract]. POPL. (2002)
5. Reddy, U. S., Yang, H.: Correctness of Data Representations involving Heap Data Structures. In Degano, P. (ed.): Proceedings of ESOP. Springer Verlag (2003) 223–237
6. Hoare, C. A. R.: Proof of Correctness of Data Representations. Acta Informatica. Vol. 1. (1972) 271–281
7. He, J., Hoare, C. A. R., Sanders, J. W.: Data Refinement Refined (Resume). In: Robinet, B., Wilhelm, R. (eds.): ESOP. LNCS. Vol. 213. Springer Verlag (1986) 187–196
8. Clarke, D. G., Noble, J., Potter, J. M.: Simple Ownership Types for Object Containment. Proceedings of ECOOP. (2001)
9. Boyapati, C., Liskov, B., Shrira, L.: Ownership Types for Object Encapsulation. POPL (2003)
10. Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures. Proceedings of LICS. Vol. 17. IEEE, Copenhagen (2002) 55–74
11. O'Hearn, P., Yang, H., Reynolds, J. C.: Separation and information hiding. POPL (2004)
12. Ishtiaq, S., O'Hearn, P. W.: BI as an Assertion Language for Mutable Data Structures. POPL. Vol. 28. ACM - SIGPLAN, London (2001)
13. O'Hearn, P., Pym, D. J.: The Logic of Bunched Implications. Bulletin of Symbolic Logic. Vol. 5 (1999)
14. Yang, H.: Relational Separation Logic. Theoretical Computer Science (2004)

Appendix: Proofs of theorems

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 $oper_i \subseteq \ S \times H \times (S \times H) \uplus \{wrong\}$ preserve $M * \mathsf{T}$, and let c be a separation context for M, P and $(oper_i)_{i \in I}$. 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 $(s',h') \in (M * \mathsf{T})_{wrong}$.*

Proof: The proof is by induction on the command c.

Let P satisfy the conditions of the theorem,and let $(s,h) \in M * P$.

Let $c \equiv \mathsf{f}_j$. Suppose that $f_j, s, h \rightsquigarrow s', h'$, then we want to prove that $(s',h') \in M * \mathsf{T}$. Relation M is precise, which means that there is at most one $h_M \sqsubseteq h$ such that $(s, h_M) \in M$. Then there exists h_U such that $h = h_M * h_U$ and $(s, h_U) \in P$. Since f_j is a separation context, $\mathsf{f}_j, s, h \not\rightsquigarrow av$, which means $f_j(s, h_U) \neq wrong$. By assumption $\mathsf{f}_j, s, h \rightsquigarrow s', h'$, i.e. $\mathsf{f}_j, s, h_M * h_U \rightsquigarrow s', h'$. Then, by frame property which f_j obeys, there exists $h'_U \sqsubseteq h'$ and $h' = h_M * h'_U$ and $f_j(s, h_U) = (s', h'_U)$. This means that $h_M \sqsubseteq h'$ and since $(s', h_M) \in M$, it follows that $(s', h_M * h'_U) \in M * \mathsf{T}$.

If, on the other hand, $f_j, s, h \rightsquigarrow wrong$, then since $wrong \in (M * \mathsf{T})_{wrong}$, this case is proved.

Consider the case when $c \equiv \mathsf{oper}_i$. Then obviously $(s,h) \in M * \mathsf{T}$, and by the assumption of the theorem oper_i preserves $M * \mathsf{T}$, which gives us the desired conclusion.

Suppose $c \equiv c_1; c_2$. From assumption that c is a separation context for M, P and $(oper_i)_{i \in I}$, we conclude that c_1 is also a separation context for M, P and $(oper_i)_{i \in I}$. Therefore, command c_1 satisfies the condition of the theorem and we can apply induction hypothesis to c_1. c_1 can produce $wrong$ starting from state (s,h) (in which case the theorem is proved), or it can produce a state (s',h'), in which case $(s',h') \in sp(c_1, M * P)$. From induction hypothesis, we know that (s',h') is also in $M * T$. Since this holds for all states $(s,h) \in M * P$ and $(s',h') \in sp(c_1, M * P)$, it follows that $sp(c_1, M * P)$ can be written as $M * Q$ for some predicate Q. Command c_2 is a separation context for M, Q and $(oper_i)_{i \in I}$, (if c_2 is not a separation context for $M * Q = sp(c_1, M * P)$, then c is not a separation context for $M * P$, which contradicts assumption) so by applying the induction hypothesis again to c_2 and $M * Q$, we conclude that the resulting state of c_2, which is also the resulting state of c is in $(M * T)_{wrong}$.

Let $c \equiv$ **if** b **then** c_1 **else** c_2. This case is straightforward if we notice that c_1 is a separation context for all the states $(s,h) \in M * P$ for which $s(b) = true$, and dually, c_2 is a separation context for all the states $(s,h) \in M * P$ for which $s(b) = false$.

For the **while** loop, we do an inner induction on the smallest derivation of **while** b **do** c, $s, h \rightsquigarrow s', h'$. □

Proposition 1. *Let* $oper^i \subseteq (S_i \times H_i) \times (S_i \times H_i) \uplus \{wrong\}$, $i = 1, 2, 3$, *be such that* $oper^2$ *is a refinement of* $oper^1$ *with respect to binary relation* V, *and* $oper^3$ *is a refinement of* $oper^2$ *with respect to binary relation* W. *Then,* $oper^3$ *is a refinement of* $oper^1$ *with respect to a relation* $V \circ W$.

Proof: Let (s_1, h_1), (s_2, h_2), (s_3, h_3) and (s_3', h_3') be such that $(s_1, h_1)[V](s_2, h_2)$, (s_2, h_2) $[W](s_3, h_3)$ and $(s_3, h_3)[oper^3](s_3', h_3')$. Then, by the definition of refinement, there exists a state (s_2', h_2') such that $(s_2, h_2)[oper^2](s_2', h_2')$ and $(s_2', h_2')[W]$ (s_3', h_3'). Similary, there exists (s_1', h_1') such that $(s_1, h_1)[oper^1](s_1', h_1')$ and (s_1', h_1') $[V](s_2', h_2')$. If, on the other hand, $(s_3, h_3)[oper^3]wrong$, again looking into definition of refinement we can conclude that $(s_1, h_1)[oper^1]wrong$, which concludes the proof. □

Theorem 2 (Simulation Theorem). *Let* R, Q, $oper^i$, c, c_i *for* $i = 1, 2$, *be as defined in Section 5.1, 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 such* P *and all states* $(s_1, h_1), (s_2, h_2)$, (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* $(s_1', h_1')[R * Q](s_2', h_2')$.

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 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 f_j, for some $j \in J$ and suppose $(s_1, h_1)[R * Q_P](s_2, h_2)$. Since c_2 is a separation context for $R_2, Q(P)$ and $oper^2$, then for any state $(s_2, h_2) \in R_2 * Q(P)$, $f_j, s_2, h_2 \not\rightsquigarrow av$. If we look at the operational semantics rules for f_j, we see that this means $(s_2, h_2) = (s_2, h_{R_2}) * (s_2, h_{U_2})$, and either $f_j(s_2, h_{U_2}) \neq wrong$ or $f_j(s_2, h_2) = wrong$.

If $f_j(s_2, h_{U_2}) \neq wrong$, let s_2', h_2' be the resulting state of execution of f_j in state (s_2, h_2). Since $(s_2, h_{R_2}) \# (s_2, h_{U_2})$, by the frame property we have that there exists (s_2', h_{U_2}') such that $(s_2', h_{U_2}') * (s_2, h_{R_2}) = (s_2', h_2')$ and $f_j(s_2, h_{U_2}) = (s_2', h_{U_2}')$. Now, since f_j maps Q-related states to Q-related states, there exists a state (s_1', h_{U_1}'), such that $f_j(s_1, h_{U_1}) = (s_1', h_{U_1}')$ and $(s_1', h_{U_1}')[Q](s_2', h_{U_2}')$. Having in mind that functions f_j have Minimum resource property, we can conclude that $f(s_1, h_1) = (s_1, h_{R_1}) * (s_1', h_{U_1}')$, and construct the following diagram.

$$\begin{array}{ccc} (s_1, h_1) = (s_1, h_{R_1}) * (s_1, h_{U_1}) & \xleftrightarrow{f_j} & (s_1, h_{R_1}) * (s_1', h_{U_1}') \\ \Big\uparrow{\scriptstyle R*Q_P} \Big\downarrow & & \Big\uparrow \Big\downarrow{\scriptstyle R*Q} \\ (s_2, h_2) = (s_2, h_{R_2}) * (s_2, h_{U_2}) & \xleftrightarrow[f_j]{} & (s_2, h_{R_2}) * (s_2', h_{U_2}') \end{array}$$

If, on the other hand $f_j(s_2, h_2) = wrong$, then also $f_j(s_2, h_{U_2}) = wrong$, and from the assumption that $\mathbf{f}_j$ maps Q-related states to Q_{wrong}-related states, it follows that $f_j(s_1, h_{U_1}) = wrong$. But, from the assumption, c_1 is a separation

context for R_1, P and $oper^1$, so it must be that $f_j(s_1, h_1) = wrong$, which proves the claim of the theorem in this case.

For the induction step, the case for **if**-branches is easy when we use the fact that boolean guards have equivalent meanings in Q related states.

For the case of composition, let $c = c'; c''$ and let (s_1, h_1), (s_2, h_2) and (s_2', h_2') be such that $(s_1, h_1)[R * Q_P](s_2, h_2)$ and $(s_2, h_2)[c_2](s_2', h_2')$. Programs c_1 and c_2 are separation contexts for R_1, P, $oper^1$ and $R_2, Q(P)$ and $oper^2$, respectively, so are c_1' and c_2', and we can apply the induction hypothesis. Let (s_2'', h_2'') be the resulting state of c_2'. Then by induction hypothesis there exists a state (s_1'', h_1'') such that $c_1', s_1, h_1 \rightsquigarrow s_1'', h_1''$ and $(s_1'', h_1'')[R * Q](s_2'', h_2'')$. We also know that $(s_1'', h_1'') \in sp(c_1', R_1 * P)$ and $(s_2'', h_2'') \in sp(c_1', R_2 * Q(P))$. Now let S be the set of all such (s_1'', h_1''). Namely,

$$S = \{(s_1'', h_1'') \mid (s_1, h_1)[R * Q_P](s_2, h_2) \ \wedge \ (s_2, h_2)[c_2'](s_2'', h_2'') \ \wedge \\ (s_2'', h_2'')[R * Q](s_1'', h_1'') \ \wedge \ (s_1, h_1)[c_1'](s_1'', h_1'')$$

Then, obviously $S[R * Q]sp(R_2 * Q(P), c_2')$, so S is of the form $R_1 * Q_1$, and $sp(R_2 * Q(P), c_2')$ is of the form $R_2 * Q_2$. Since R is precise and having in mind how S is constructed, there exist A such that $S = R_1 * A$ and $sp(R_2 * Q(P), c_2') = R_2 * Q(A)$. Clearly, c_1'' is a separation context for R_1, A and $oper^1$ (if it were not, since $R_1 * A = S \subseteq sp(R_1 * P, c_1')$, c_1 would not be a separation context for R_1, P and $oper^1$). Similarly c_2'' is a separation context for $R_2, Q(A)$ and $oper^2$, so we can once again apply the induction hypothesis and construct the following diagram.

$$\begin{array}{ccccc} (s_1, h_1) & \xleftarrow{c_1'} & (s_1'', h_1'') & \xleftarrow{c_1''} & (s_1', h_1') \\ {\scriptstyle R*Q_P}\updownarrow & & \updownarrow{\scriptstyle (R*Q_A)_{wrong}} & & \updownarrow{\scriptstyle (R*Q)_{wrong}} \\ (s_2, h_2) & \xleftarrow[c_2']{} & (s_2'', h_2'') & \xleftarrow[c_2'']{} & (s_2', h_2') \end{array}$$

In case when either c_2' or c_2'' goes $wrong$ in corresponding states, the reasoning is similar.

When c is **while** b **do** c', and c_1 and c_2 are separation contexts for $R_1, P, oper^1$ and $R_2, Q(P), oper^2$, respectively, we prove by induction that for all $(s_1, h_1)[R * Q_P](s_2, h_2)$ and derivations $c_2, s_2, h_2 \rightsquigarrow s_2', h_2'$ of depth n, there is a state (s_1', h_1') such that $c_1, s_1, h_1 \rightsquigarrow s_1', h_1'$ and $(s_1', h_1')[(R * Q)_{wrong}](s_2', h_2')$. When $n = 0$, we use the rule

$$\frac{[\![b]\!]s_2 = 0}{c_2, s_2, h_2 \rightsquigarrow s_2, h_2,}$$

and it is easy to see that nothing happens for the program c_1 (because of the assumption on Q), so the conditions for the theorem are clearly fulfilled. Now, suppose that the last step in the derivation is an application of the rule

$$\frac{[\![b]\!]s_2 = 1 \quad c_2', s_2, h_2 \rightsquigarrow s_2'', h_2'' \quad c_2, s_2'', h_2'' \rightsquigarrow s_2', h_2'}{c_2, s_2, h_2 \rightsquigarrow s_2', h_2'}$$

By the outer induction hypothesis, c_1' and c_2' are separation contexts for $R_1, P, oper^1$ and $R_2, Q(P), oper^2$, respectively, and for the resulting state of c_2' there is a state (s_1'', h_1'') such that the left part of the diagram below "commutes".

$$\begin{array}{ccccc} (s_1,h_1) & \xleftarrow{c_1'} & (s_1'',h_1'') & \xleftarrow{c_1} & (s_1',h_1') \\ {\scriptstyle R*Q_P}\updownarrow & & \updownarrow{\scriptstyle (R*Q_A)_{wrong}} & & \updownarrow{\scriptstyle (R*Q)_{wrong}} \\ (s_2,h_2) & \xleftarrow[c_2']{} & (s_2'',h_2'') & \xleftarrow[c_2]{} & (s_2',h_2') \end{array}$$

As in previous case, we construct the set S, and find the unary predicate A such that $S = R_1 * A$ and $sp(R_2 * P, c_2') = R_2 * Q(A)$. 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_1', h_1') with $c_1, s_1'', h_1'' \rightsquigarrow s_1', h_1'$ and $(s_1', h_1')[(R*Q)_{wrong}](s_2', h_2')$. So the diagram gives us states (s_1', h_1'), (s_2', h_2') with the desired properties. This completes the proof of the lemma. □

Lemma 1. *Let R, Q, $oper^i$, c, c_i for $i = 1, 2$, be as defined in Section 5.1, 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 $oper^2$.*

Proof: We will look at few cases that are the most interesting ones.

The case when $c = oper$ relies only on a definition of refinement for opers.

Let $c \equiv f_j$ and (s_1, h_1) and (s_2, h_2) be such that $(s_1, h_1)[R * Q_P](s_2, h_2)$. Suppose $f_j, s_2, h_2 \rightsquigarrow av$, i.e. $(s_2, h_2) = (s_2, h_{R_2}) * (s_2, h_{U_2})$ and $f_j, s_2, h_{U_2} \rightsquigarrow wrong$. Pair $\mathbf{f}_j$ maps Q-related states to Q_{wrong}-related states, so it means that $f_j, s_1, h_{U_1} \rightsquigarrow wrong$, where $(s_1, h_1) = (s_1, h_{R_1}) * (s_1, h_{U_1})$, which is a contradiction to the assumption that c_1 is a safe separation context. We reason similarly when $f_j, s_2, h_2 \rightsquigarrow wrong$.

For the case of composition, assume that c is $c'; c''$ and $(s_1, h_1)[R*Q_P](s_2, h_2)$. Since c_1 is safe separation context for R_1, Q_1 and $oper^1$, c_1' is too. By the induction hypothesis, this implies that c_2' is also safe separation context with respect to $R_2, Q(P)$ and $oper^2$. Now, the conditions of theorem 2 are fulfilled, and we can apply it. Namely, for any output state (s_2'', h_2'') of c_2', there exists an output state (s_1'', h_1'') of c_1', and we can in the same manner as in the proof of 2 construct the set S, and find the predicate A, such that $S = R_1 * A$ and $sp(R_1 * Q(P), c_2') = R_1 * Q(A)$ and repeat the reasoning principle, which leads us to applying the induction hypothesis once again for c_1'' and c_2''. □

Lemma 2. *Intermediate implementation of memory manager module defined in Section 5.1, (***Example***), refines its abstract implementation with respect to relation $R_{AI} * Q_{AI}$, where*

$$R_{AI} = \begin{pmatrix} \mathsf{emp} \\ f \end{pmatrix} \qquad Q_{AI} = \mathbf{Id}$$

Proof: Suppose $(s_A, h_A)[R_{AI} * Q_{AI}](s_I, (M_I, h_I))$, and $(s_I, (M_I, h_I))[\mathsf{new}_I(x)]$ $(s'_I, (M'_I, h'_I))$. We must argue that there exists a state (s'_A, h'_A) such that (s_A, h_A) $[\mathsf{new}_A(x)](s'_A, h'_A)$ and (s'_A, h'_A) $[R_{AI} * Q_{AI}](s'_I, h'_I)$. From $(s_I, (M_I, h_I))[\mathsf{new}_I(x)]$ $(s'_I, (M'_I, h'_I))$, we have that there is some $n \in M_I$ and $s'_I(x) = n$ and $(M'_I, h'_I) = (M_I - \{n\}, h) * (\emptyset, x \mapsto nil)$. On the other hand, if the set of locations owned by the module is empty, then there exists $n \notin dom(h_I)$ and $s'_I(x) = n$ and $(M'_I, h'_I) = (\emptyset, h) * (\emptyset, x \mapsto nil)$. In both cases, n evidently does not belong to the client's part of the heap ($n \notin dom(h_I)$), and because of the assumption $(s_A, h_A)[R_{AI} * Q_{AI}](s_I, (M_I, h_I))$, n cannot be in the clients's part of the heap $dom(h_A)$ either. So, n must be owned by the system, and we construct a state (s'_A, h'_A) such that $h'_A = h_A * n \mapsto nil$ and $s'_A = s_A[x \mapsto n]$. It is not hard to see that $(s_A, h_A)[\mathsf{new}_A(x)](s'_A, h'_A)$ and $(s'_A, h'_A)[R_{AI} * Q_{AI}](s'_I, (M'_I, h'_I))$, as desired.

We omit the proof for dispose. □

Lemma 3. *Concrete implementation of memory manager module defined in Section 5.1, (***Example***), refines its intermediate implementation with respect to relation $R_{IC} * Q_{IC}$, where*

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

Proof: Suppose $(s_I, (M_I, h_I))[R_{IC} * Q_{IC}](s_C, h_C)$, and $(s_C, h_C)[\mathsf{new}_C(x)](s'_C, h'_C)$. There are two cases.

If $s'_C = s_C[x \mapsto s_C(ls), ls \mapsto h_C(s_C(ls)).2, \alpha \mapsto \mathsf{tl}(s_C(\alpha))]$, $h'_C = h_C[s_C(ls) \mapsto nil, nil]$, and $s'_C(\alpha) \neq \varepsilon$, we have $s_C(ls) = \mathsf{hd}(s_C(\alpha)) \in s_I(f)$ because of the definition of R_{IC}. We can therefore construct the state

$$(s'_I, (M'_I, h'_I)) = (s_I[x \mapsto s_C(ls)], (M_I \setminus s_C(ls), h_I * s_C(ls) \mapsto nil, nil)),$$

and we see that this state has the desired properties, namely $(s'_I, M'_I)[R_{IC} * Q_{IC}](s'_C, h'_C)$ and $(s_I, (M_I, h_I))[\mathsf{new}_I(x)](s'_I, (M'_I, h'_I))$.

The other case is when $s'_C = s_C[x \mapsto n]$, $s_C(ls) = nil$, $h'_C = h_C * n \mapsto nil, nil$, $n \notin dom(h_C)$, and $s_C(\alpha) = \varepsilon$. In this case, we have $s_I(f) = \emptyset$, so the state

$$(s'_I, (M'_I, h'I)) = (s_I[x \mapsto n], (M_I, h_I * n \mapsto nil, nil))$$

has the desired properties. □