Safe Modification of Pointer Programs in Refinement Calculus

Susumu NISHIMURA

Dept. of Mathematics, Graduate School of Science, Kyoto University
Sakyo-ku, Kyoto 606-8502, JAPAN
susumu@math.kyoto-u.ac.jp

Abstract. This paper discusses stepwise refinement of pointer programs in the framework of refinement calculus. We augment the underlying logic with formulas of separation logic and then introduce a pair of new predicate transformers, called *separating assertion* and *separating assumption*. The new predicate transformers are derived from separating conjunction and separating implication, which are fundamental logical connectives in separation logic. They represent primitive forms of heap allocation/deallocation operators and the basic pointer statements can be specified by means of them. We derive several refinement laws that are useful for stepwise refinement and demonstrate the use of the laws in the context of correctness preserving transformations that are intended for improved memory usage.
The formal development is carried out in the framework of higher-order logic and is based on Back and Preoteasa's axiomatization of state space and its extension to the heap storage [BP05, Pre06]. All the results have been implemented and verified in the theorem prover PVS.

1 Introduction

Pointers are a powerful tool that provides clean and efficient solutions to certain programming tasks. However, pointers are also notoriously hard to handle because of the problems caused by their effectful nature, e.g., pointer aliasing and dangling pointers. Thus correct implementation of pointers is a challenging issue in any stage of program development.

This paper studies safe modifications of pointer programs in the framework of refinement calculus: we want to safely modify a pointer program to another one in a way that the modification is guaranteed to preserve the correctness. For the moment we informally presume that a program T being a correct modification of a program S means that T executes gracefully in any program context that S does so. In other words, T can safely replace any occurrence of S in a program.

Modifying a pointer program correctly is often a delicate task that requires great care. (Throughout, we are particularly interested in correct transformations of pointer programs that are intended for improved memory usage and therefore we deal with mostly such examples of program transformations. The results of this paper are not limited to this particular variety of applications but they can be adopted to wider purposes, e.g., deriving programs from specifications.) Consider the following program S_0

that successively executes two pointer statements.

$$S_0 \quad = \quad x := \mathbf{alloc}(12); \mathbf{free}(p)$$

The first statement $x := \mathbf{alloc}(12)$ allocates an arbitrary fresh heap address, updates the value stored at that address to 12, and assigns variable x the fresh address; the second statement $\mathbf{free}(p)$ performs deallocation that reclaims the address p.

One might be tempted to rewrite this program into the following program U, in order to improve memory efficiency.

$$U \quad = \quad [p] := 12; x := p$$

The mutation statement $[p] := 12$ updates the value stored at the address p to 12; the subsequent assignment $x := p$ assigns p to the variable x. This modified program is intended to reuse the address p for the subsequent execution, instead of discarding it away and allocating a fresh address.

This rewrite is not correct, however. The former program S_0 assigns x a fresh address different from p (because the address p is already allocated at the moment of fresh allocation), while the latter program U sets p to x. This means that, the two programs will show different behaviors, if they are put in a larger context of a program that tests on the equality between p and x.

In contrast, it is safe to rewrite the following program S_1 into program U.

$$S_1 \quad = \quad \mathbf{free}(p); x := \mathbf{alloc}(12)$$

Note that this modification is indeed safe but the two programs are not exactly equal. Program U always assigns p to x, while program S_1 assigns either p or an arbitrary fresh address. This means that U can safely replace S_1 (since a graceful execution of S_1 in a context implies a graceful execution of U, which has a lesser variety of variable assignments, in the same context), but not vice versa.

The example above indicates that correct modifications of pointer programs can be a very delicate matter: even slight changes may unexpectedly disrupt the safety of programs. This implies that the correctness of a pointer program of reasonable size would be desperately intricate to be established by an informal argument.

Seeking for a rigorous way to derive safe modifications of pointer programs, we augment the framework of *refinement calculus* [BvW98, Mor94] so that it can handle pointer statements. Refinement calculus deals with concrete programs as well as abstract specifications universally as predicate transformers. Predicate transformers are ordered by *refinement relation* $S \sqsubseteq T$, which is defined by "$S(\varphi)$ implies $T(\varphi)$ for every postcondition φ". In other words, $S \sqsubseteq T$ iff Hoare triple assertion $\{\varphi\}T\{\psi\}$ holds whenever $\{\varphi\}S\{\psi\}$ holds for any pair of precondition φ and postcondition ψ.[1] Thus $S \sqsubseteq T$ implies that T can safely replace S. As for the above example, $S_1 \sqsubseteq U$ holds, but $S_0 \sqsubseteq U$ does not.

To deal with pointer programs in refinement calculus, we make use of formulas of separation logic [ORY01, Rey02] to qualify the properties of (a formalized) heap

[1] The triple $\{\varphi\}S\{\psi\}$ asserts the total correctness of program, i.e., when executed in any state satisfying precondition φ, program S terminates and establishes postcondition ψ.

storage. Then we can verify safety of program modifications by proving propositions expressed in separation logic formulas.

Developing a formal proof of the refinement of an entire program, however, can be a task of unmanageable size and complexity. Therefore a modular style development, called *stepwise refinement*, is effective in practice: we successively apply refinement laws to subcomponents of the program until we obtain the desired result.

In order to manage modifications of pointer programs in the paradigm of stepwise refinement, we identify two predicate transformers, called *separating assertion* and *separating assumption*, as fundamental units. They represent primitive forms of heap operations that work in complementary ways: separating assertion, written $\{\varphi\}^*$, works as a generic deallocation statement that reclaims a portion of the heap storage as specified by φ; separating assumption, written $[\varphi]^*$, works as a generic allocation statement that allocates a subheap as specified by φ.

The merits of formulating pointer programs using these two predicate transformers are twofold.

- Separating assertion and separating assumption enjoy several simple but useful refinement laws. They were inspired from predicate transformers called *assertion* and *assumption*, which are useful for handling context information in the stepwise refinement process [Bac88, Mor94, LvW97, BvW98, Gro00]. Assertions and assumptions are defined in terms of conjunction $\cap$ and implication $\Rightarrow$ in classical logic, while separating assertions and separating assumptions are defined in terms of logical operators of separating counterpart, namely, separating conjunction $*$ and separating implication $-\!\!*$ in separation logic. Both counterparts share certain logical properties. In particular, the adjunctive relationship of the classical counterpart:

$$\varphi \cap \psi \text{ entails } \gamma \quad \text{iff} \quad \varphi \text{ entails } \psi \Rightarrow \gamma$$

  is paralleled by that of the separating counterpart [Rey02]:

$$\varphi * \psi \text{ entails } \gamma \quad \text{iff} \quad \varphi \text{ entails } \psi -\!\!* \gamma.$$

  Due to the similarities in logical properties, both counterparts of predicate transformers enjoy similar sets of refinement laws. With these laws, separating assertions and assumptions can be used to handle context information about pointer programs together with assertions and assumptions.
- Basic pointer statements that are in common use in programming can be defined as composition of more primitive predicate transformers such as separating assertions and separating assumptions. This compound representation is useful for proving refinement laws about pointer statements, because finer logical granularity implies greater opportunities of stepwise refinement and lesser size and complexity of proof.

We have implemented and verified the results in this paper in the theorem prover PVS [SRSC01]. In the present paper, however, we will show the proofs for interesting cases only. Most of the remaining cases are an easy exercise, except for a few non-trivial cases whose proofs are too lengthy to write down on papers. Interested readers can take a look at the proof script that is available from the author's homepage `http://www.math.kyoto-u.ac.jp/~susumu/mpc08pvs/`.

Related work Extensions of refinement calculus by pointer manipulations and separation logic have been studied by a few researchers. Back, Fan, and Preoteasa [BFP03] extended refinement calculus with pointer statements by providing a model that allows explicit reference to the heap storage. Preoteasa [Pre06] presented, based on Back and Preoteasa's axiomatization of states [BP05], another extension of refinement calculus that employs separation logic formulas as a means to expressing heap properties.

The formalization in the present paper basically follows Preoteasa's but we add two substantial extensions to it. First, we introduce separating implication and give a formal semantics for it. Separating implication is a principal logical operation for describing the axiomatic semantics of heap allocation in separation logic, while Preoteasa gives the semantics in a different way without using it. Second, we introduce the new predicate transformers, separating assertions and separating assumptions, and define pointer statements by means of them.

The present paper provides a set of refinement laws for handling separating assertions and separating assumptions and applies them to derivations of pointer programs, in much the same way as is done for their classical counterpart [Bac88, Mor94, LvW97, BvW98, Gro00]. In contrast, Preoteasa's work aims at establishing the correctness of Hoare rules and the frame rule of separation logic; he did not give refinement laws explicitly. The refinement laws and examples in the present paper shall be successfully proved in his formalization. However, we believe that the present formalization enables a finer stepwise refinement process, as we have argued earlier. The most crucial consequence of the present work is that we identify separating assertions and separating assumptions as fundamental units of pointer programs and provide a set of refinement laws that are useful for stepwise refinement of pointer programs.

Outline The rest of the paper is organized as follows. Section 2 gives a formal definition of refinement calculus and its extension with separation logic formulas. Section 3 introduces predicate transformers that represent primitive operations on the heap storage and defines basic pointer statements in terms of them. Section 4 lists refinement laws for these transformers. In Section 5, we examine and discuss how pointer statements interact with each other, and in Section 6 we show a transformation process for a larger pointer program. Finally Section 7 concludes the paper.

2 Refinement Calculus and Its Formalization

This section summarizes refinement calculus and its formalization in the higher-order logic, following the formalization techniques in [BvW98, BP05]. We then extend refinement calculus with separation logic formulas, in order to deal with pointer statements. This extension is formalized by a variant of the technique proposed in [Pre06].

We will use the following notations. Let A, B be types (sets) in the higher-order logic. We write either $a \in A$ or $a : A$ to mean that a is a member of A. We write $\mathcal{P}_{\mathit{fin}}(A)$ for the finite powerset of A, and $A \setminus B$ for the set difference. A function of type $A \to B$ is often written in λ-notation $\lambda x : A.M$, where M is a term that denotes a value of type B. The type annotations in λ-notation are often left implicit. A predicate over A is a function $A \to \mathsf{bool}$, where bool denotes the type of boolean values $\{\mathsf{true}, \mathsf{false}\}$. The type of predicates forms a complete boolean lattice. Given predicates φ, ψ over A, we write $\varphi \cup \psi$, $\varphi \cap \psi$, $\varphi \subseteq \psi$, and $\neg\varphi$ to denote predicate union, intersection, inclusion, and complement, respectively. The logical interpretations of $\cup$, $\cap$, $\subseteq$, $\neg$ are disjunction, conjunction, entailment, and negation, resp., and we also write implication $\varphi \Rightarrow \psi$ to abbreviate $\neg\varphi \cup \psi$. A relation R between types A and B is a subset of the product $A \times B$. For a relation $R \subseteq A \times B$, we define its inverse relation by $R^{-1} = \{(y,x) \mid (x,y) \in R\}$. For a function $f : A \rightarrow B$, we write $graph(f)$ for the graph of f, i.e., $\{(x, f(x)) | x \in A\}$.

2.1 Axiomatization of states

Let Value be the type representing the set of all values. Value includes the set of locations $\mathsf{Location}$ and the set of constants $\mathsf{Constant}$. We assume $\mathsf{Constant}$ at least contains boolean values and nil. For every location x, we write $\mathsf{T}(x)$ to denote the type of values assignable to x.

In order to formalize a state model that has a heap storage, Preoteasa [Pre06] defined $\mathsf{Location}$ to be the disjoint union of sets:

$$\mathsf{Location} \triangleq \mathsf{Variable} \uplus \mathsf{Address} \uplus \{\mathsf{alloc}\},$$

where $\mathsf{Variable}$ is the infinite set of variables, $\mathsf{Address}$ is the infinite set of addresses (of the heap storage), and alloc is a distinguished location denoting the finite set of allocation addresses. We assume that variables and addresses can be assigned arbitrary values and thus the types of locations are given by $\mathsf{T}(x) = \mathsf{Value}$ for any $x \in \mathsf{Variable} \cup \mathsf{Address}$ and $\mathsf{T}(\mathsf{alloc}) = \mathcal{P}_{fin}(\mathsf{Address})$.

Let State be the type of *states*. It is intended that a state is a pair of a mapping from locations to values and a LIFO queue (often called *stack*) that is used for saving/restoring values. Following Back and Preoteasa [BP05], we do not stick to a concrete definition of a state space but rather specify it by a set of state manipulation functions and its axiomatization.

We have three functions for state manipulation.

Lookup	$\mathsf{val}(x) : \mathsf{State} \rightarrow \mathsf{T}(x)$	$(x \in \mathsf{Location})$
Update	$\mathsf{set}(x) : \mathsf{T}(x) \rightarrow \mathsf{State} \rightarrow \mathsf{State}$	$(x \in \mathsf{Location})$
Restore	$\mathsf{del}(x) : \mathsf{State} \rightarrow \mathsf{State}$	$(x \in \mathsf{Location})$

Figure 1 gives the axioms[2] that characterize the intended meaning of these functions: $\mathsf{val}(x)(\sigma)$ returns the value that is assigned to the location x in the state σ. $\mathsf{set}(x)(v)(\sigma)$ returns a new state obtained by updating the value of the location x to v in the state σ. $\mathsf{del}(x)(\sigma)$ returns a new state obtained by "popping" the top most stack value and restoring the value to the location x in the state σ. The operation for saving values to the stack will be defined as the relational inverse of del.

In what follows, two states $\sigma, \sigma' \in \mathsf{State}$ are called val-*equivalent* if $\mathsf{val}(x)(\sigma) = \mathsf{val}(x)(\sigma')$ holds for any location $x \in \mathsf{Location}$ [BP05]. We notice that $\sigma = \sigma'$ implies

[2] Back and Preoteasa used these axioms to formalize local variable declarations and recursive procedure calls [BP05]. The present paper does not deal with recursive procedure calls but they should be incorporated without much difficulty.

$$\mathsf{val}(x)(\mathsf{set}(x)(v)(\sigma)) \;=\; v \tag{2.1}$$
$$x \neq y \;\Rightarrow\; \mathsf{val}(y)(\mathsf{set}(x)(v)(\sigma)) \;=\; \mathsf{val}(y)(\sigma) \tag{2.2}$$
$$\mathsf{set}(x)(v)(\mathsf{set}(x)(u)(\sigma)) \;=\; \mathsf{set}(x)(v)(\sigma) \tag{2.3}$$
$$x \neq y \;\Rightarrow\; \mathsf{set}(x)(v)(\mathsf{set}(y)(u)(\sigma)) \;=\; \mathsf{set}(y)(u)(\mathsf{set}(x)(v)(\sigma)) \tag{2.4}$$
$$\mathsf{set}(x)(\mathsf{val}(x)(\sigma))(\sigma) \;=\; \sigma \tag{2.5}$$
$$\forall\sigma.\exists\sigma'.\sigma \;=\; \mathsf{del}(x)(\sigma') \tag{2.6}$$
$$x \neq y \;\Rightarrow\; \mathsf{val}(y)(\mathsf{del}(x)(\sigma)) \;=\; \mathsf{val}(y)(\sigma) \tag{2.7}$$
$$\mathsf{del}(x)(\mathsf{set}(x)(v)(\sigma)) \;=\; \mathsf{del}(x)(\sigma) \tag{2.8}$$
$$x \neq y \;\Rightarrow\; \mathsf{del}(x)(\mathsf{set}(y)(u)(\sigma)) \;=\; \mathsf{set}(y)(u)(\mathsf{del}(x)(\sigma)) \tag{2.9}$$

Fig. 1. Axioms for states [BP05]

σ and σ' are val-equivalent but the converse does not hold in general, because two val-equivalent states may hold different values in the stack.

2.2 Expressions and predicates over states

An *expression* over State is a function $\mathsf{State} \to A$, written $\mathsf{Exp}(A)$, where A is the type of the value of the expression.

We remark that the bare variable x just denotes the *location* of the variable and that an expression that refers to the *value* of x is denoted by $\mathsf{val}(x)$. For example, an assignment statement `x:=y` is expressed as $x := \mathsf{val}(y)$ in our notation.

Let e be an expression of type $\mathsf{Exp}(A)$, x be a location, and e' be an expression of type $\mathsf{Exp}(\mathsf{T}(x))$. We define *substitution* of location x in e by e', written $e[e'/x]$, as follows.

$$e[e'/x] \;\triangleq\; \lambda\sigma.e(\mathsf{set}(x)(e'(\sigma))(\sigma))$$

An expression e is called *x-independent* if $e(\mathsf{set}(x)(v)(\sigma)) = e(\sigma)$ holds for any state σ and any value v of type $\mathsf{T}(x)$. For notational convention, we write $FV(e)$ to denote the set of "free" locations in e, i.e., $FV(e) \triangleq \{x \in \mathsf{Location} \mid e \text{ is not } x\text{-independent}\}$.

An expression e is called *finitely-dependent* if $FV(e)$ is finite. An expression e is called val-*determined*, if $e(\sigma) = e(\sigma')$ for any val-equivalent states σ, σ'. An expression e is called *pure* if its value does not depend on the state of the heap, i.e., it holds that

$$e(\mathsf{set}(\mathsf{alloc})(h)(\sigma)) \;=\; e(\sigma) \;\text{ and }\; e(\mathsf{set}(a)(v)(\sigma)) \;=\; e(\sigma)$$

for any $\sigma \in \mathsf{State}$, $h \in \mathcal{P}_{fin}(\mathsf{Address})$, $a \in \mathsf{Address}$, and $v \in \mathsf{Value}$.

We call an expression e is *nonalloc-independent* [Pre06] if it holds that

$$e(\mathsf{set}(a)(v)(\sigma)) \;=\; e(\sigma)$$

for any $\sigma \in \mathsf{State}$, $a \in \mathsf{Address} \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$, and $v \in \mathsf{Value}$. Apparently pure expressions are a subclass of nonalloc-independent expressions.

A predicate (over state) is an expression of boolean type, namely an expression of type $\mathsf{Exp}(\mathsf{bool})$.

2.3 Separation logic formulas

We denote by Exp the subtype of $\mathsf{Exp}(\mathsf{Value})$ that consists of expressions which are pure, finitely-dependent, and val-determined. In what follows we assume that every program expression e is a member of Exp. The usual (syntactically constructed) expressions that contain no access to the heap storage have type Exp.

Let us use the following shorthand notations.

$$\begin{aligned}\mathsf{add\_alloc}(a)(\sigma) &\triangleq \mathsf{set}(\mathsf{alloc})(\{a\} \cup \mathsf{val}(\mathsf{alloc})(\sigma))(\sigma)\\ \mathsf{diff\_alloc}(h)(\sigma) &\triangleq \mathsf{set}(\mathsf{alloc})(\mathsf{val}(\mathsf{alloc})(\sigma) \setminus h)(\sigma)\end{aligned}$$

We introduce a partial order relation $\preceq$ over states that denotes one state is an extension of the other. The formal definition is by induction on the size of allocation set: $\sigma \preceq \sigma'$ iff $\mathsf{val}(\mathsf{alloc})(\sigma) \subseteq \mathsf{val}(\mathsf{alloc})(\sigma')$ and either

- $\sigma = \sigma'$ (hence $\mathsf{val}(\mathsf{alloc})(\sigma) = \mathsf{val}(\mathsf{alloc})(\sigma')$) or
- $\mathsf{set}(a)(\mathsf{val}(a)(\sigma'))(\mathsf{add\_alloc}(a)(\sigma)) \preceq \sigma'$ holds for any $a \in \mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$.

This is intended that σ' allocates an equal or larger set of addresses than σ does and that σ' may assign different values for the extended set of addresses, i.e., $\mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$. There are no other differences between σ and σ': they have the same assignment of values to variables and the same values saved in the stack. A similar but more abstract notion of this partial ordering can be found in [COY07].

For expressions $e, e' \in \mathsf{Exp}$ and predicates $\varphi, \psi \in \mathsf{Exp}(\mathsf{bool})$, we define the semantics of separation logic formulas as follows.

$$\begin{aligned}\mathbf{emp} &\triangleq \lambda\sigma.(\mathsf{val}(\mathsf{alloc})(\sigma) = \emptyset)\\ e \mapsto e' &\triangleq \lambda\sigma.\big(\mathsf{val}(\mathsf{alloc})(\sigma) = \{e(\sigma)\} \text{ and } \mathsf{val}(e(\sigma))(\sigma) = e'(\sigma)\big)\\ \varphi * \psi &\triangleq \lambda\sigma.\Big(\exists h.\big(h \subseteq \mathsf{val}(\mathsf{alloc})(\sigma) \text{ and}\\ &\qquad \varphi(\mathsf{set}(\mathsf{alloc})(h)(\sigma)) \text{ and } \psi(\mathsf{diff\_alloc}(h)(\sigma))\big)\Big)\\ \varphi \mathrel{-\!\!*} \psi &\triangleq \lambda\sigma.\big(\forall\sigma'.\ [\sigma \preceq \sigma' \text{ and } \varphi(\mathsf{diff\_alloc}(\mathsf{val}(\mathsf{alloc})(\sigma))(\sigma'))] \text{ implies } \psi(\sigma')\big)\end{aligned}$$

The intended semantics is explained as below.

emp (empty heap) The heap allocates nothing.

$e \mapsto e'$ **(single allocation)** The heap allocates the value e' at the address e (and nothing else).

$\varphi * \psi$ **(separating conjunction)** The heap can be split into two separate heap domains (that is, the allocation set $\mathsf{val}(\mathsf{alloc})(\sigma)$ is divided into two disjoint address sets, h and $\mathsf{val}(\mathsf{alloc})(\sigma) \setminus h$ for some h) so that φ holds for one subheap and ψ holds for the other.

$\varphi \mathrel{-\!\!*} \psi$ **(separating implication)** ψ holds for any extension σ' of the current heap σ such that φ holds for the extended heap domain. That is, no matter how the current allocation set $\mathsf{val}(\mathsf{alloc})(\sigma)$ is extended with an additional address set, say h, that satisfies φ and is disjoint from $\mathsf{val}(\mathsf{alloc})(\sigma)$, ψ holds for the extended allocation set $h \cup \mathsf{val}(\mathsf{alloc})(\sigma)$. The partial order $\sigma \preceq \sigma'$ indicates that the additional address set is the difference $\mathsf{val}(\mathsf{alloc})(\sigma') \setminus \mathsf{val}(\mathsf{alloc})(\sigma)$ and that arbitrary values are stored at the extended heap addresses.

For notational convenience, we also make use of the following abbreviations.

– We write $e \mapsto -$ to mean that $e \mapsto v$ holds for some value v. The notation $e \mapsto -$ represents a heap that allocates a single address e but the stored value does not matter.
– $e \hookrightarrow e'$ abbreviates $e \mapsto e' * \mathsf{true}$; similarly, $e \hookrightarrow -$ abbreviates $e \mapsto - * \mathsf{true}$. They represent a heap that allocates the address e and possibly other addresses.

We can also formally specify some specific classes of separation logic formulas that advocate stronger logical properties. For example, we specify the class of *precise* predicates [OYR04] as those predicates φ satisfying:

$$\sigma \preceq \sigma_0 \text{ and } \varphi(\sigma) \text{ and } \sigma' \preceq \sigma_0 \text{ and } \varphi(\sigma') \ \text{ implies } \ \mathsf{val}(\mathsf{alloc})(\sigma) = \mathsf{val}(\mathsf{alloc})(\sigma') \ ,$$

for any states σ, σ', and σ_0.[3] Informally, a predicate φ is precise if, for any state σ_0, there exists at most one restriction to a subheap that satisfies φ: that is, there exists at most one subset h of the allocation set $\mathsf{val}(\mathsf{alloc})(\sigma_0)$ such that, for any σ satisfying φ, $\sigma \preceq \sigma_0$ implies $\mathsf{val}(\mathsf{alloc})(\sigma) = h$. We can formally show that, when φ and ψ are precise, so are **emp**, $e \mapsto e'$, $e \mapsto -$, $\varphi * \psi$, and $\varphi \cap \psi$.

2.4 Predicate transformers

We define pointer manipulating programs as predicate transformers that operate on the type of predicates:

$$\mathsf{Pred} \ \triangleq \ \{\varphi \in \mathsf{Exp}(\mathsf{bool}) \mid \varphi \text{ is nonalloc-independent}\}.$$

We restrict the type of predicates to those nonalloc-independent ones, since execution of programs should not be affected by the values of heap addresses that are not allocated. (Any attempt to access the heap storage at a non-allocated address immediately causes an error, e.g., segmentation fault.) The type Pred is closed under logical operators including those of separation logic.

Proposition 2.1. *Let $x \in \mathsf{Variable}$, $e, e' \in \mathsf{Exp}$, $\varphi, \psi \in \mathsf{Pred}$. Then the following predicates are all members of* Pred.

$$\varphi \cup \psi \quad \varphi \cap \psi \quad \neg\psi \quad \varphi \Rightarrow \psi \quad (\forall x)\varphi \quad (\exists x)\varphi \quad \mathbf{emp} \quad e \mapsto e' \quad \varphi * \psi \quad \varphi \mathrel{-\!\!*} \psi$$

where $(\forall x)$ and $(\exists x)$ are quantifications over Pred:

$$(\forall x)\varphi \ \triangleq \ \lambda\sigma.(\forall v \in \mathsf{Value}.\varphi(\mathsf{set}(x)(v)(\sigma)))$$
$$(\exists x)\varphi \ \triangleq \ \lambda\sigma.(\exists v \in \mathsf{Value}.\varphi(\mathsf{set}(x)(v)(\sigma))) \ .$$

[3] This definition is slightly more general than the usual definition in separation logic: "for any state σ, there is at most one address set h such that h is a subset of the current allocation set and the restriction of σ to h satisfies φ." The two definitions are indeed equivalent if we restrict the set of predicates to the class of nonalloc-independent ones, which we will consider in Section 2.4.

We define the type MTran of predicate transformers that are monotonic w.r.t. predicate inclusion, i.e.,

$$\mathsf{MTran} \;\triangleq\; \{S \in \mathsf{Pred}{\to}\mathsf{Pred} \mid \forall\varphi,\psi.\big(\varphi \subseteq \psi \text{ implies } S(\varphi) \subseteq S(\psi)\big)\}.$$

We write $S;T$ for the sequential composition of monotonic predicate transformers $S,T \in \mathsf{MTran}$ and define it simply by function composition, i.e., $S;T \triangleq S \circ T$. The type MTran is closed under sequential composition.

The refinement relation $\sqsubseteq$ over MTran is defined as pointwise extension of $\subseteq$, i.e.,

$$S \sqsubseteq T \quad \text{iff} \quad \forall\varphi \in \mathsf{Pred}.S(\varphi) \subseteq T(\varphi)\ .$$

We say S is refined to T, if $S \sqsubseteq T$ holds. The equality $S = T$ holds iff both $S \sqsubseteq T$ and $T \sqsubseteq S$ hold.

We also define operations $\sqcup$ and $\sqcap$ over MTran as pointwise extensions of $\cup$ and $\cap$:

$$(S \sqcup T)(\varphi) \;\triangleq\; S(\varphi) \cup T(\varphi) \qquad\qquad (S \sqcap T)(\varphi) \;\triangleq\; S(\varphi) \cap T(\varphi)\ .$$

Proposition 2.2 ([BvW98]). MTran *forms a complete lattice with* $\sqsubseteq$, $\sqcap$, *and* $\sqcup$ *being the partial order, meet, and join, respectively. The least element of* MTran *is* **abort** $\triangleq \lambda\varphi.\mathsf{false}$ *and the greatest element is* **magic** $\triangleq \lambda\varphi.\mathsf{true}$.

The least element **abort** represents a program whose execution may not terminate normally (because of non-termination or errors). The greatest element **magic** is a miraculous (unimplementable) program that can establish arbitrary postcondition for any precondition.

A predicate transformer $S \in \mathsf{MTran}$ is called *conjunctive* if it holds that $S(\bigsqcap\{\varphi \mid \varphi \in \Gamma\}) = \bigsqcap\{S(\varphi) \mid \varphi \in \Gamma\}$ for any nonempty subset Γ of Pred; dually, S is called *disjunctive* if it holds that $S(\bigsqcup\{\varphi \mid \varphi \in \Gamma\}) = \bigsqcup\{S(\varphi) \mid \varphi \in \Gamma\}$. We also call $S \in \mathsf{MTran}$ *terminating* if $S;$**magic** $=$ **magic**; Dually we call S *feasible* if $S;$**abort** $=$ **abort**.

3 Statements as Predicate Transformers

We introduce several predicate transformers of type MTran. In what follows, let us write $x,y,\cdots$ for variables, $e,e',\cdots$ for expressions of type Exp, and $\varphi,\psi,\cdots$ for predicates of type Pred. We also write B for *pure* predicates.

3.1 Basic program statements

The idle statement **skip** and the assignment statement $x := e$ are defined as follows.

$$\textbf{skip} \;\triangleq\; \lambda\varphi.\varphi \qquad\qquad x := e \;\triangleq\; \lambda\varphi.(\varphi[e/x])$$

Back and Preoteasa [BP05] defined statements for saving/restoring the value of variable. Let $[R] \triangleq \lambda\varphi.\lambda\sigma.(\forall\sigma'.\sigma\, R\, \sigma' \text{ implies } \varphi(\sigma'))$ be the relational demonic update for arbitrary relation $R \subseteq \mathsf{State} \times \mathsf{State}$. The definition of the save statement $\mathsf{Add}(x)$ and the restore statement $\mathsf{Del}(x)$ is given as below.

$$\mathsf{Add}(x) \triangleq \left[graph(\mathsf{del}(x))^{-1}\right] \qquad \mathsf{Del}(x) \triangleq \left[graph(\mathsf{del}(x))\right]$$

Using the save/restore statements in pair, we can put a predicate transformer S into a scope of a local variable x by

$$\mathsf{Add}(x); S; \mathsf{Del}(x) \ .$$

Note that, since $\mathsf{Add}(x)$ is the inverse of $\mathsf{Del}(x)$, it saves the current value of x onto the stack and assigns an *arbitrary* value to x; the saved value is restored on exit by $\mathsf{Del}(x)$. This indicates that the local scope might be alternatively interpreted by means of universal quantification as Morgan [Mor94] did, i.e.,

$$\big(\mathsf{Add}(x); S; \mathsf{Del}(x)\big)(\varphi) = \forall x.(S(\varphi)) \ ,$$

where the local variable x is assumed not to occur free in φ. In the present formalization, this is not provable for arbitrary S but it actually holds when S represents a “normal” program where any Add and Del always appear in pair and the value saved in the stack is never accessed until the program execution leaves the corresponding local scope. Throughout the paper, we consider such normal programs only and thus the above alternative interpretation of local scoping is valid, although we have to justify it for each concrete instance of S in a formal development.

3.2 Abstract statements

The logical quantifications $(\forall x)$ and $(\exists x)$ can be as well regarded as predicate transformers and we call them *demonic assignment* and *angelic assignment* [BvW98], respectively. Both non-deterministically assign a value to the variable, but the angelic assignment chooses a value that establishes the postcondition, if possible.

We can also define predicate transformers that are related to logical conjunction and implication as follows.

$$\{\psi\} \triangleq \lambda\varphi.(\psi \cap \varphi) \qquad [\psi] \triangleq \lambda\varphi.(\psi \Rightarrow \varphi)$$

The predicate transformers $\{\psi\}$ is called *assertion* and $[\psi]$ is called *assumption*. They work as primitive forms of conditional statements: if ψ is satisfied, both of them act like **skip**; otherwise, $\{\psi\}$ acts like **abort**, while $[\psi]$ acts like **magic**.

Some programming constructs can be defined in terms of the above abstract predicate transformers. The conditional statement can be expressed using $\sqcap$:

$$\textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ fi} \triangleq \big([B]; S\big) \sqcap \big([\neg B]; T\big) \ .$$

Let $\mathcal{F}$ be a monotonic function from MTran to MTran. By proposition 2.2 and Knaster-Tarski theorem, $\mathcal{F}$ has least fixpoint, written $\mu.\mathcal{F}$. This allows us to define recursive program constructs as least fixpoints in MTran. We give below the least fixpoint definition of the while loop.

$$\textbf{while } B \textbf{ do } S \textbf{ od} \triangleq \mu.(\lambda U.\textbf{if } B \textbf{ then } S; U \textbf{ else skip fi})$$

3.3 Pointer statements

Let us consider four basic pointer statements: *lookup* $x := [e]$, *mutation* $[e] := e'$, *allocation* $x := \mathbf{alloc}(e)$, and *deallocation* $\mathbf{free}(e)$. (The *lookup* statement $x := [e]$ updates the variable x to the value stored at the address e of the heap. The other statements have been explained in Introduction.)

The weakest preconditions for these basic pointer statements can be expressed in terms of separation logic formulas [Rey02]. Accordingly they can be recognized as predicate transformers of the following forms:

$$\begin{aligned} x := [e] &= \lambda\varphi.\exists y.(e \hookrightarrow y \cap \varphi[y/x]) && (3.1)\\ [e] := e' &= \lambda\varphi.(e \mapsto -) * (e \mapsto e' -\!\!* \varphi) && (3.2)\\ x := \mathbf{alloc}(e) &= \lambda\varphi.\forall y.(y \mapsto e -\!\!* \varphi[y/x]) && (3.3)\\ \mathbf{free}(e) &= \lambda\varphi.(e \mapsto - * \varphi) && (3.4)\end{aligned}$$

where x and y are distinct and $y \notin FV(e) \cup FV(\varphi)$.

Here we can observe that these predicate transformers calculate a compound formula that combines logical connectives such as $*$ and $-\!\!*$. This suggests that the definitions above could be expressed by combining simpler predicate transformers, each of which corresponds to a single logical connective.

Let us define a pair of new predicate transformers as follows.

$$\{\psi\}^* \triangleq \lambda\varphi.(\psi * \varphi) \qquad\qquad [\psi]^* \triangleq \lambda\varphi.(\psi -\!\!* \varphi)$$

These are a separating counterpart of assertion and assumption in Section 3.2 and therefore called *separating assertion* and *separating assumption*, respectively. They represent the complementary pair of operations over the heap storage, namely, heap deallocation and allocation. Separating assertion $\{\psi\}^*$ reclaims a part of the current heap as mentioned by ψ; if no subheap establishes ψ, it acts like **abort**. Separating assumption $\{\psi\}^*$ extends the heap with a set of fresh allocations as mentioned by ψ; if no fresh allocation that satisfies ψ is available, it acts like **magic**.

Combining separating assertion, separating assumption, local variable scoping, etc., we give alternative definitions of pointer statements as below:

$$\begin{aligned} x := [e] &\triangleq \mathsf{Add}(y); (\exists y); \{e \hookrightarrow \mathsf{val}(y)\}; x := \mathsf{val}(y); \mathsf{Del}(y) && (3.5)\\ [e] := e' &\triangleq \{e \mapsto -\}^*; [e \mapsto e']^* && (3.6)\\ x := \mathbf{alloc}(e) &\triangleq \mathsf{Add}(y); [\mathsf{val}(y) \mapsto e]^*; x := \mathsf{val}(y); \mathsf{Del}(y) && (3.7)\\ \mathbf{free}(e) &\triangleq \{e \mapsto -\}^* && (3.8)\end{aligned}$$

where x and y are distinct and $y \notin FV(e)$.

We justify these compound definitions as follows. By a simple calculation we have that the compound definition $\{e \mapsto -\}^*; [e \mapsto e']^*$ of the mutation statement is equivalent to the predicate transformer that maps a postcondition φ to $(e \mapsto -) * (e \mapsto e' -\!\!* \varphi)$; as for the allocation, by the discussion in Section 3.1, we have that $[\mathsf{val}(y) \mapsto e]^*; x :=$

$\mathsf{val}(y)$ maps φ to $\forall y.(\mathsf{val}(y) \mapsto e \mathrel{-\!\!*} \varphi[y/x])$, where the universal quantification is introduced by the local variable scoping. Here the local variable scoping is needed for another reason: it guarantees that y is a fresh variable (as required by the side condition for the definition (3.3)). The local variable y can be recognized as fresh because a possible occurrence of y in φ, though being syntactically identical, denotes a different entity: the former denotes the value local to the scope, while the latter denotes the value outside the scope. Similar arguments apply to the lookup and deallocation statements.

The definition of lookup statement (3.5) implicitly contains separating assertion and separating assumption: from the fact that $e \hookrightarrow e' \cap \varphi$ is equivalent to $e \mapsto e' * (e \mapsto e' \mathrel{-\!\!*} \varphi)$ for any φ [Rey02], we have

$$\{e \hookrightarrow e'\} \ = \ \{e \mapsto e'\}^*; [e \mapsto e']^* \ . \tag{3.9}$$

Thus the definition is alternatively expressed as follows.

$$x := [e] \ \triangleq \ \mathsf{Add}(y); (\exists y); \{e \mapsto \mathsf{val}(y)\}^*; [e \mapsto \mathsf{val}(y)]^*; x := \mathsf{val}(y); \mathsf{Del}(y) \tag{3.10}$$

4 Refinement Laws

We will show refinement laws for predicate transformers defined in the previous section. We denote predicate transformers in MTran by $S, T, U, \cdots$, program expressions in Exp by $e, e', \cdots$, and predicates in Pred by $P, Q, \cdots$. We also assume that program expressions and predicates are val-determined and finitely-dependent, unless otherwise stated. Any usual (syntactically constructed) expressions and predicates satisfy these properties.

The compound statements, which combine the basic statements given in the previous section by sequential composition ; , meet $\sqcap$, join $\sqcup$, and the least fixpoint operator, are conventionally called *programs*. The next proposition indicates that a refinement of any substatement of a program gives a refinement of the entire program. (In the subsequent development, we will exploit this fact without explicitly mentioning it.)

Proposition 4.1 (monotonicity[BvW98]). *The sequential composition, meet, and join of predicate transformers preserve monotonicity. That is, for any S, T, S', T' such that $S \sqsubseteq S'$ and $T \sqsubseteq T'$ it holds that $S; T \sqsubseteq S'; T'$, $S \sqcap T \sqsubseteq S' \sqcap T'$, and $S \sqcup T \sqsubseteq S' \sqcup T'$. Furthermore, the least fixpoint operator preserves monotonicity, in the sense that $\mu.\mathcal{F}$ is monotonic if $\mathcal{F}$ is a monotonic function that preserves monotonicity.*

We list below several simple refinement laws.

$$S; \mathbf{skip} = \mathbf{skip}; S = S \tag{4.1}$$

$$(S \sqcap T); U = (S; U) \sqcap (T; U) \tag{4.2}$$

$$S; (T \sqcap U) \sqsubseteq (S; T) \sqcap (S; U) \tag{4.3}$$

$$(S \sqcup T); U = (S; U) \sqcup (T; U) \tag{4.4}$$

$$(S; T) \sqcup (S; U) \sqsubseteq S; (T \sqcup U) \tag{4.5}$$

4.1 Laws for local variable scoping

The next law indicates that one can freely introduce or eliminate an empty local variable scope.

Proposition 4.2. *[BP05]*

$$\mathsf{Add}(x);\mathsf{Del}(x) \;=\; \textbf{skip} \tag{4.6}$$

In the course of program refinement, we often need to enlarge or shrink the scope of local variables. The next proposition shows that this is possible, provided that there is no collision in variable names.

Proposition 4.3. *It holds that $S;\mathsf{Add}(x) = \mathsf{Add}(x);S$ and $S;\mathsf{Del}(x) = \mathsf{Del}(x);S$, in either of the following cases.*

- *S is either* **abort**, **magic**, *$\{P\}$, $[P]$, $[P]^*$, $y := e$, $y := [e]$, $[e] := e'$, $y :=$ **alloc**(e), or* **free**(e)*, where x and y are distinct and $x \notin FV(e) \cup FV(e') \cup FV(P)$.*
- *S is a separating assertion $\{P\}^*$, where P is precise and $x \notin FV(P)$.*

On exit from a local scope $\mathsf{Add}(x);S;\mathsf{Del}(x)$, the original value of the variable x is restored and hence any assignment to x in the local scope is overridden (the state axiom (2.8)). Thus we can prove the following refinement laws.

$$x := e;\mathsf{Del}(x) \;=\; \mathsf{Del}(x) \quad (4.7) \qquad x := e;\mathsf{Del}(y);\mathsf{Del}(x) \;=\; \mathsf{Del}(y);\mathsf{Del}(x) \quad (4.8)$$

If x and y are distinct and $x \notin FV(e')$, it follows from the state axioms (2.3) and (2.9) that

$$x := \mathsf{val}(y);\mathsf{Del}(y);x := e' \;=\; \mathsf{Del}(y);x := e'\ . \tag{4.9}$$

4.2 Laws for assertions and assumptions and their separating counterpart

We list below some of the refinement laws of assertions and assumptions and contrast them with the laws of the separating counterpart.

Proposition 4.4.

$$\{P\} \sqsubseteq \{Q\} \quad \textit{if } P \subseteq Q \quad (4.10) \qquad [P] \sqsubseteq [Q] \quad \textit{if } Q \subseteq P \quad (4.14)$$
$$\{P\};\{Q\} = \{Q\};\{P\} = \{P \cap Q\} \quad (4.11) \qquad [P];[Q] = [Q];[P] = [P \cap Q] \quad (4.15)$$
$$\{P\}^* \sqsubseteq \{Q\}^* \quad \textit{if } P \subseteq Q \quad (4.12) \qquad [P]^* \sqsubseteq [Q]^* \quad \textit{if } Q \subseteq P \quad (4.16)$$
$$\{P\}^*;\{Q\}^* = \{Q\}^*;\{P\}^* = \{P * Q\}^* \quad (4.13) \qquad [P]^*;[Q]^* = [Q]^*;[P]^* = [P * Q]^* \quad (4.17)$$

These laws show that assertion (assumption, resp.) is monotonic (antimonotonic, resp.) w.r.t. predicate inclusion and also assertions and assumptions are commutative for sequential composition; similarly for the separating counterpart.

Some of the laws for the interaction between assertions and assumptions and the corresponding laws for the separating counterpart are given below.

Proposition 4.5.

$$\textbf{skip} \sqsubseteq [P];\{Q\} \quad \textit{if } P \subseteq Q \quad (4.18) \qquad \{P\};[Q] \sqsubseteq \textbf{skip} \quad \textit{if } P \subseteq Q \quad (4.22)$$
$$\{P\};S \sqsubseteq T \;\; \textit{iff} \;\; S \sqsubseteq [P];T \quad (4.19) \qquad \{P\};[Q] \sqsubseteq [Q];\{P\} \quad (4.23)$$
$$\{P\}^*;[Q]^* \sqsubseteq \textbf{skip} \quad \textit{if } P \subseteq Q \quad (4.20) \qquad \textbf{skip} \sqsubseteq [P]^*;\{Q\}^* \quad \textit{if } P \subseteq Q \quad (4.24)$$
$$\{P\}^*;S \sqsubseteq T \;\; \textit{iff} \;\; S \sqsubseteq [P]^*;T \quad (4.21) \qquad \{P\}^*;[Q]^* \sqsubseteq [Q]^*;\{P\}^* \quad (4.25)$$

We can observe that the refinement laws for both counterparts run completely in parallel. This is because both counterparts satisfy some significant logical properties such as monotonicity, commutativity, and the adjunctive relationship in parallel. For example, given the proof for the law (4.25):

$$
\begin{array}{llll}
& \{P\}^*;[Q]^* \sqsubseteq [Q]^*;\{P\}^* & & \\
\text{iff} & P*(Q \mathrel{-\!\!*} \varphi) \subseteq Q \mathrel{-\!\!*} (P*\varphi) & \text{for any } \varphi & \\
\text{iff} & Q*P*(Q \mathrel{-\!\!*} \varphi) \subseteq P*\varphi & \text{for any } \varphi & \text{(adjunction)} \\
\text{iff} & P*Q*(Q \mathrel{-\!\!*} \varphi) \subseteq P*\varphi & \text{for any } \varphi & \text{(commutativity)} \\
\text{if} & Q*(Q \mathrel{-\!\!*} \varphi) \subseteq \varphi & \text{for any } \varphi & \text{(monotonicity)} \\
\text{iff} & Q \mathrel{-\!\!*} \varphi \subseteq Q \mathrel{-\!\!*} \varphi & \text{for any } \varphi & \text{(adjunction)} \\
\text{iff} & \mathit{True}\,, & &
\end{array}
$$

we instantly obtain the proof for the other counterpart (4.23) simply by replacing the symbols $\{-\}^*$, $[-]^*$, and $*$ by $\{-\}$, $[-]$, and $\cap$, respectively.[4]

We can also show that some stronger refinement laws hold for the separating counterpart, provided that the predicate P belongs to a particular class of predicates.

Proposition 4.6. *The followings hold for any pure predicate P.*

$$
\begin{array}{llll}
\{P\} \sqsubseteq \{P\}^* & (4.26) & \{Q\}^*;\{P\} = \{P\};\{Q\}^* = \{Q \cap P\}^* & (4.28) \\
[P]^* \sqsubseteq [P] & (4.27) & \{Q\}^*;[P] \sqsubseteq [P];\{Q\}^* & (4.29)
\end{array}
$$

Proposition 4.7. *The followings hold for any precise predicate P.*

$$
\{P\}^*;\{Q\} = \{P*Q\};\{P\}^* \quad (4.30) \qquad \{Q\};[P]^* \sqsubseteq [P]^*;\{Q*P\} \quad (4.31)
$$

4.3 Commutativity laws for statements

The next proposition gives several commutativity laws for pairs of statements that have no collision in variable names.

Proposition 4.8. *The following refinement laws hold, if x and y are distinct, $x \notin FV(e') \cup FV(P)$, and $y \notin FV(e)$.*

$$
\begin{array}{llll}
x := e;\{P\} \;=\; \{P\};x := e & (4.32) & x := e; y := e' \;=\; y := e'; x := e & (4.36) \\
x := e;[P] \;=\; [P];x := e & (4.33) & (\exists x); y := e' \;=\; y := e'; (\exists x) & (4.37) \\
x := e;\{P\}^* \;=\; \{P\}^*;x := e & (4.34) & (\exists x);\{P\} \;=\; \{P\};(\exists x) & (4.38) \\
x := e;[P]^* \;=\; [P]^*;x := e & (4.35) & (\exists x);[P]^* \;\sqsubseteq\; [P]^*;(\exists x) & (4.39)
\end{array}
$$

When two sequentially composed statements have a collision in variable names, we may use refinement laws that exchange the execution order subject to modifications of the original statements. We list below a few such laws for assignment statements.

[4] Note that not every refinement law in one counterpart has a corresponding law in the other. For instance, $\{P\};S \sqsubseteq S$ holds for arbitrary P but $\{P\}^*;S \sqsubseteq S$ does not.

Proposition 4.9.

$$x := e; \{P\} = \{P[e/x]\}; x := e \quad (4.40) \qquad x := e; \{P\}^* = \{P[e/x]\}^*; x := e \quad (4.42)$$

$$x := e; [P] = [P[e/x]]; x := e \quad (4.41) \qquad x := e; [P]^* = [P[e/x]]^*; x := e \quad (4.43)$$

5 Refinement of Pointer Statements

Now we are ready to discuss refinement laws for pointer statements. First we show that pointer statements commute with assignment, provided that there is no collision in variable names.

Lemma 5.1. *The following refinement laws hold, if x and y are distinct, $x \notin FV(e')$, and $y \notin FV(e)$.*

$$x := e; y := [e'] \;=\; y := [e']; x := e \qquad (5.1)$$

$$x := e; [y] := e' \;=\; [y] := e'; x := e \qquad (5.2)$$

$$x := e; y := \mathbf{alloc}(e') \;=\; y := \mathbf{alloc}(e'); x := e \qquad (5.3)$$

$$x := e; \mathbf{free}(e') \;=\; \mathbf{free}(e'); x := e \qquad (5.4)$$

Proof. These refinement laws follow from the definition of the pointer statements and the commutativity laws in proposition 4.3 and 4.8. □

The next lemma shows that any pointer statement that attempts to access the heap storage via a dangling pointer is equal to **abort**.

Lemma 5.2.

$$\{\neg e \hookrightarrow -\}; x := [e] \;=\; \{\neg e \hookrightarrow -\}; [e] := e' \;=\; \{\neg e \hookrightarrow -\}; \mathbf{free}(e) \;=\; \mathbf{abort}$$

Proof. We first show the proof for the deallocation statement. By the definition, we have $\{\neg e \hookrightarrow -\}; \mathbf{free}(e) = \{\neg e \hookrightarrow -\}; \{e \mapsto -\}^*$. This is equivalent to **abort**, since for any postcondition φ, $(\neg e \hookrightarrow -) \cap (e \mapsto - * \varphi)$ implies $(\neg e \hookrightarrow -) \cap (e \hookrightarrow -)$, which is unsatisfiable. The proof for the mutation statement is similar.

For the lookup statement, we have $\{\neg e \hookrightarrow -\}; x := [e] = \mathsf{Add}(y); (\exists y); \{\neg e \hookrightarrow -\}; \{e \hookrightarrow \mathsf{val}(y)\}; x := \mathsf{val}(y); \mathsf{Del}(y)$ by proposition 4.3 and law (4.38). This is equivalent to **abort**, because so is the subcomponent $\{\neg e \hookrightarrow -\}; \{e \hookrightarrow \mathsf{val}(y)\}$. □

The things get more complicated and delicate when we deal with interaction between pointer statements, as we have seen in Introduction. In what follows, we demonstrate that some refinement laws for such delicate interactions can be verified by means of stepwise refinement.

We first show the following refinement law that we have argued in Introduction.

Proposition 5.1. *If $y \notin FV(e)$, then it holds that*

$$\mathbf{free}(e); y := \mathbf{alloc}(e') \;\sqsubseteq\; [e] := e'; y := e\ . \qquad (5.5)$$

Proof. By the definition of allocation and deallocation statements and proposition 4.3, the lhs is equal to:

$$\mathsf{Add}(z); \{e \mapsto -\}^*; [\mathsf{val}(z) \mapsto e']^*; y := \mathsf{val}(z); \mathsf{Del}(z)$$

for some fresh variable z. As we have observed in Section 3, the scope of local variable z constitutes a universal quantification over z and hence (by instantiating z to e) the above compound statement can be refined to[5]

$$\{e \mapsto -\}^*; [e \mapsto e']^*; y := e\ ,$$

which is equal to the rhs of (5.5). □

Let us examine another refinement law, whose proof is more involved.

Proposition 5.2. *If x and y are distinct, $x \notin FV(e')$, and $y \notin FV(e)$, then it holds that*

$$x := [e]; y := \mathbf{alloc}(e') \quad = \quad y := \mathbf{alloc}(e'); x := [e]\ . \tag{5.6}$$

Proof. We prove this by showing that both sides of the equation refine each other.

The following derivation steps prove that the rhs is a refinement of the lhs. (Throughout, we put references to the laws, propositions, etc. that justify the derivation, on the left of each derivation step within a pair of angle brackets. A symbol like ♠ refers to a subsidiary law that will be justified later on.)

$$\begin{array}{rcl} & & x := [e]; y := \mathbf{alloc}(e') \\ \langle\text{Def.\&Prop. 4.3}\rangle & = & \mathsf{Add}(z); x := [e]; [\mathsf{val}(z) \mapsto e']^*; y := \mathsf{val}(z); \mathsf{Del}(z) \\ \langle\spadesuit\rangle & \sqsubseteq & \mathsf{Add}(z); [\mathsf{val}(z) \mapsto e']^*; x := [e]; y := \mathsf{val}(z); \mathsf{Del}(z) \\ \langle\text{(5.1), Prop. 4.3}\rangle & = & y := \mathbf{alloc}(e'); x := [e] \end{array}$$

In the derivation step ♠, we applied the following law.

$$x := [e]; [\mathsf{val}(y) \mapsto e']^* \sqsubseteq [\mathsf{val}(y) \mapsto e']^*; x := [e]$$

This subsidiary law is derived as follows. By the definition of lookup statement, proposition 4.3, and the refinement laws (4.35), (4.39), this is equivalent to showing:

$$\begin{array}{rl} & \mathsf{Add}(z); (\exists z); \{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^*; x := \mathsf{val}(z); \mathsf{Del}(z) \\ \sqsubseteq & \mathsf{Add}(z); (\exists z); [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\}; x := \mathsf{val}(z); \mathsf{Del}(z)\ . \end{array}$$

Hence it is enough to show that $\{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^* \sqsubseteq [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\}$. This is derived by the following stepwise refinement.

$$\begin{array}{rcl} & & \{e \hookrightarrow \mathsf{val}(z)\}; [\mathsf{val}(y) \mapsto e']^* \\ \langle(3.9)\rangle & = & \{e \mapsto \mathsf{val}(z)\}^*; [e \mapsto \mathsf{val}(z)]^*; [\mathsf{val}(y) \mapsto e']^* \\ \langle(4.17)\rangle & \sqsubseteq & \{e \mapsto \mathsf{val}(z)\}^*; [\mathsf{val}(y) \mapsto e']^*; [e \mapsto \mathsf{val}(z)]^* \\ \langle(4.25)\rangle & \sqsubseteq & [\mathsf{val}(y) \mapsto e']^*; \{e \mapsto \mathsf{val}(z)\}^*; [e \mapsto \mathsf{val}(z)]^* \\ \langle(3.9)\rangle & = & [\mathsf{val}(y) \mapsto e']^*; \{e \hookrightarrow \mathsf{val}(z)\} \end{array}$$

[5] In the formal proof, we need to boil down the compound statements into single predicate transformers and prove that they are equal up to extensionality, appealing to the state axioms given in Figure 1. The proof procedure is cumbersome but a routine.

To show the converse refinement, since it holds that $S = (\{P\};S) \sqcup (\{\neg P\};S)$, it is enough to show the following two cases:

$$\begin{aligned}&\{\neg e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \sqsubseteq x := [e]; y := \mathbf{alloc}(e') \text{ and}\\&\{e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \sqsubseteq x := [e]; y := \mathbf{alloc}(e')\ .\end{aligned}$$

For the first case, by the definition of allocation statement, Proposition 4.3, laws (4.31), (4.32), and lemma 5.2, we have $\{\neg e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e] \sqsubseteq \mathsf{Add}(z);$ $[\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -]^*; \mathbf{abort}$. This is equivalent to the least element **abort**, since $\mathsf{Add}(z); [\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -]^*$ is feasible. The feasibility follows from the fact that $\mathsf{val}(z) \mapsto e' * \neg e \hookrightarrow -$ can be satisfied by assigning z a non-allocated heap address other than the one denoted by e.

To establish the other case, we derive as follows.

$$\begin{aligned}
& && \{e \hookrightarrow -\}; y := \mathbf{alloc}(e'); x := [e]\\
\langle\text{Prop. 4.3, (5.1)}\rangle\ & = && \mathsf{Add}(z); \{e \hookrightarrow -\}; [\mathsf{val}(z) \mapsto e']^*; x := [e]; y := z; \mathsf{Del}(z)\\
\langle\clubsuit\rangle\ & \sqsubseteq && \mathsf{Add}(z); x := [e]; [\mathsf{val}(z) \mapsto e']^*; y := z; \mathsf{Del}(z)\\
\langle\text{Prop. 4.3}\rangle\ & \sqsubseteq && x := [e]; y := \mathbf{alloc}(e').
\end{aligned}$$

To justify the derivation step $\clubsuit$, we need a subsidiary law:

$$\{e \hookrightarrow -\}; [\mathsf{val}(z) \mapsto e']^*; x := [e] \sqsubseteq x := [e]; [\mathsf{val}(z) \mapsto e']^*.$$

By a similar calculation as above, we can see that this is established by showing

$$\{e \hookrightarrow -\}; [\mathsf{val}(z) \mapsto e']^*; (\exists w); \{e \hookrightarrow w\} \sqsubseteq (\exists w); \{e \hookrightarrow w\}; [\mathsf{val}(z) \mapsto e']^*.$$

This refinement law holds, since

$$e \hookrightarrow - \cap (\mathsf{val}(z) \mapsto e' \mathrel{-\!\!*} (\exists w)(e \hookrightarrow w \cap \varphi)) \Rightarrow (\exists w)(e \hookrightarrow w \cap (\mathsf{val}(z) \mapsto e' \mathrel{-\!\!*} \varphi))$$

is a valid separation logic formula for any (postcondition) φ. □

6 Example: Recycling Heap Beyond Loop Boundaries

We apply our refinement laws to a larger program. Let us consider the following program.

$$S_0 \triangleq \begin{array}{l}\mathsf{Add}(j); \mathbf{while}\ i \neq \mathsf{nil}\ \mathbf{do}\\ \qquad j := \mathsf{val}(i); i := [\mathsf{val}(i)]; o := \mathbf{alloc}(\mathsf{val}(o)); \mathbf{free}(\mathsf{val}(j))\\ \quad \mathbf{od}; \mathsf{Del}(j)\end{array}$$

In the program, variables i and o are initially assigned a pointer to a chain of pointers terminated by the constant nil (Fig. 2-(0)). Every single iteration of the loop body is intended to update the pointer structure in the following steps. First the value of i is saved in j and then the pointer i is dereferenced to follow the link one step forward

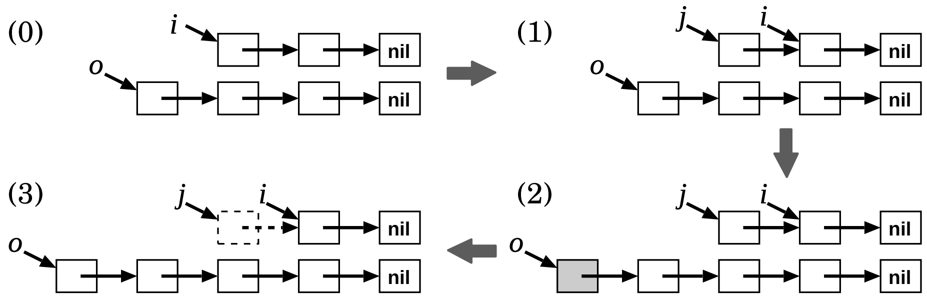

Fig. 2. Updating pointer structure by a single iteration of program S_0. (Dotted lines represent reclaimed data and shaded box represent a freshly allocated heap cell.)

(Fig. 2-(1)). Next, a fresh address is allocated to extend the chain of o by one (Fig. 2-(2)). Finally the address that has been saved in j is reclaimed (Fig. 2-(3)). Repeating this iteration until i reaches to nil, the program "appends" the two chains of pointers; the variable o is updated to refer to the resulting chain of pointers, whose length is the sum of the lengths of the initial chains.

One might be tempted to improve memory usage by rewriting the substatement $o := \textbf{alloc}(\mathsf{val}(o)); \textbf{free}(\mathsf{val}(j))$ into $[\mathsf{val}(j)] := \mathsf{val}(o); o := \mathsf{val}(j)$, which is intended to reuse the address $\mathsf{val}(j)$ in place of a freshly allocated one, instead of deallocating it. However, this is not a safe rewrite, as we have observed in Introduction.

We will show another safe way to improving memory usage. Refining the deallocation statement $\textbf{free}(\mathsf{val}(j))$ together with its subsequent allocation statement $o := \textbf{alloc}(\mathsf{val}(o))$ beyond the loop boundary, we can obtain the following program T_0 as a refinement of S_0.

$$T_0 \triangleq \begin{array}{l} \mathsf{Add}(j); j := \textbf{alloc}(\mathsf{nil}); \\ \quad \textbf{while } i \neq \mathsf{nil} \textbf{ do} \\ \quad\quad [\mathsf{val}(j)] := \mathsf{val}(o); o := \mathsf{val}(j); j := \mathsf{val}(i); i := [\mathsf{val}(i)] \\ \quad \textbf{od}; \\ \quad \textbf{free}(\mathsf{val}(j)); \mathsf{Del}(j) \end{array}$$

To show this formally, we need some lemmas regarding the loop construct.

Lemma 6.1.

(a) If x and y are distinct, $x, y \notin FV(B)$, and $y \notin FV(e)$, we have

$$\begin{array}{l} \mathsf{Del}(x); \textbf{while } B \textbf{ do } y := e; S \textbf{ od}; \mathsf{Del}(y) \\ \sqsubseteq\ y := \mathsf{val}(x); \mathsf{Del}(x); \textbf{while } B \textbf{ do } y := e; S \textbf{ od}; \mathsf{Del}(y)\ . \end{array} \tag{6.1}$$

(b) If S is disjunctive, and the refinement relations $S; [B] \sqsubseteq [B]; S$, $S; [\neg B] \sqsubseteq [\neg B]; S$, and $S; T \sqsubseteq T'$ hold, we have

$$S; \textbf{while } B \textbf{ do } T; S \textbf{ od} \sqsubseteq \textbf{while } B \textbf{ do } T' \textbf{ od}; S\ . \tag{6.2}$$

The first law (6.1) holds, since the effect of assignment $y := \mathsf{val}(x)$ in the rhs is either (i) canceled by the subsequent $\mathsf{Del}(y)$ if the loop condition B is never established or (ii) overridden by the assignment $y := e$ in the first iteration of the loop body. The second law (6.2) holds because $S;(T;S)^n$ $(n \geq 0)$ is refined to $(S;T)^n;S$, which is further refined to $T'^n;S$. We can formally prove these properties by exploiting the fixpoint property of the loop statement and also applying transfinite induction over ordinals [BvW98]. See Appendix for the detail of the proof.

Let us write S'_0 for the loop statement in S_0. We derive the refinement relation $S_0 \sqsubseteq T_0$ as follows.

$$\begin{array}{rcl}
S_0 & = & \mathsf{Add}(j);S'_0;\mathsf{Del}(j) \\
\langle(4.6)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\mathsf{Del}(k);S'_0;\mathsf{Del}(j) \\
\langle\text{Lemma 6.1(a)}\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);j := \mathsf{val}(k);\mathsf{Del}(k);S'_0;\mathsf{Del}(j) \\
\langle(4.24)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\big[\mathsf{val}(k) \mapsto \mathsf{nil}\big]^*;\big\{\mathsf{val}(k) \mapsto \mathsf{nil}\big\}^*;j := \mathsf{val}(k); \\
 & & \qquad \mathsf{Del}(k);S'_0;\mathsf{Del}(j) \\
\langle(4.12)\rangle & \sqsubseteq & \mathsf{Add}(j);\mathsf{Add}(k);\big[\mathsf{val}(k) \mapsto \mathsf{nil}\big]^*;\big\{\mathsf{val}(k) \mapsto -\big\}^*;j := \mathsf{val}(k); \\
 & & \qquad \mathsf{Del}(k);S'_0;\mathsf{Del}(j) \\
\langle(4.42),\text{ Prop. 4.3}\rangle & = & \mathsf{Add}(j);\mathsf{Add}(k);\big[\mathsf{val}(k) \mapsto \mathsf{nil}\big]^*;j := \mathsf{val}(k); \\
 & & \qquad \mathsf{Del}(k);\big\{\mathsf{val}(j) \mapsto -\big\}^*;S'_0;\mathsf{Del}(j) \\
\langle\text{Defs.}\rangle & \sqsubseteq & \mathsf{Add}(j);j := \mathbf{alloc}(\mathsf{nil});\mathbf{free}(\mathsf{val}(j));S'_0;\mathsf{Del}(j) \\
\langle\text{Lemma 6.1(b)}\rangle & \sqsubseteq & T_0
\end{array}$$

To justify the last step of derivation, we need to check if the premises of lemma 6.1(b) hold. The disjunctivity of $\mathbf{free}(\mathsf{val}(j))$ follows from the fact that separating conjunction distributes over disjunction [Rey02]. The first and second prerequisite refinement relations hold by law (4.29); The remaining prerequisite refinement relation is verified as follows.

$$\begin{array}{rcl}
 & & \mathbf{free}(\mathsf{val}(j));j := \mathsf{val}(i);i := [\mathsf{val}(i)];o := \mathbf{alloc}(\mathsf{val}(o)) \\
\langle(5.6)\rangle & = & \mathbf{free}(\mathsf{val}(j));j := \mathsf{val}(i);o := \mathbf{alloc}(\mathsf{val}(o));i := [\mathsf{val}(i)] \\
\langle(5.3)\rangle & = & \mathbf{free}(\mathsf{val}(j));o := \mathbf{alloc}(\mathsf{val}(o));j := \mathsf{val}(i);i := [\mathsf{val}(i)] \\
\langle(5.5)\rangle & \sqsubseteq & [\mathsf{val}(j)] := \mathsf{val}(o);o := \mathsf{val}(j) j := \mathsf{val}(i);i := [\mathsf{val}(i)]
\end{array}$$

7 Conclusion and Future Work

We have introduced two new predicate transformers, called separating assertion and separating assumption, into refinement calculus as primitive forms of deallocating and allocating the heap storage. These primitives are defined by means of separating conjunction and separating implication that are fundamental adjunctive logical operators in separation logic. We have shown that they satisfy several refinement laws that are useful for developing safe modification of pointer programs.

There are a few topics that will merit further investigations. The refinement laws and program modifications discussed in this paper are mostly due to simple logical facts and stepwise refinement, but a cumbersome proof task is required in the proof of Proposition 5.1 to show that a particular instance of local variable scoping constitutes a universal quantification. We would be able to simplify this proof step by identifying a class of predicate transformers for which a general refinement law on local variable scoping holds. Another future work would be to take into account of the local reasoning principle (so called frame rule) of separation logic [ORY01, YO02] and to investigate refinement laws that hold under a richer separation context on the heap storage.

Acknowledgment I thank anonymous reviewers for their helpful comments.

References

[Bac88] Ralph-Johan Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, 1988.

[BFP03] Ralph-Johan Back, Xiaocong Fan, and Viorel Preoteasa. Reasoning about pointers in refinement calculus. In *10th Asia-Pacific Software Engineering Conference (APSEC 2003)*, pages 425–434, 2003.

[BP05] Ralph-Johan Back and Viorel Preoteasa. An algebraic treatment of procedure refinement to support mechanical verification. *Formal Aspects of Computing*, 17(1):69–90, 2005.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 366–378. IEEE Computer Society, 2007.

[Gro00] Lindsay John Groves. *Evolutionary Software Development in the Refinement Calculus*. PhD thesis, Victoria University of Wellington, 2000.

[LvW97] Linas Laibinis and Joakim von Wright. Context handling in the refinement calculus framework. Technical Report 118, TUCS Technical Report, 1997.

[Mor94] Carroll Morgan. *Programming from specifications*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 2nd edition edition, 1994.

[ORY01] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[OYR04] Peter O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 268–280. ACM Press, 2004.

[Pre06] Viorel Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 508–523. Springer, 2006.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[SRSC01] N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, November 2001. `http://pvs.csl.sri.com/`.

[YO02] Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.

Appendix

Proof of lemma 6.1(a). Let us define $\mathcal{F}(U) \triangleq \left[\neg B\right] \sqcap \left[B\right];y := e;S;U$. The lemma is shown by the following derivation.

$$\begin{array}{rcl}
\mathsf{Del}(x);\mu.\mathcal{F};\mathsf{Del}(y) & = & \mathsf{Del}(x);\left(\left[\neg B\right] \sqcap \left[B\right];y := e;S;\mu.\mathcal{F}\right);\mathsf{Del}(y) \\
\langle(4.3),(4.2)\rangle & \sqsubseteq & \mathsf{Del}(x);\left[\neg B\right];\mathsf{Del}(y) \sqcap \mathsf{Del}(x);\left[B\right];y := e;S;\mu.\mathcal{F};\mathsf{Del}(y) \\
\langle\text{Prop. } 4.3\rangle & \sqsubseteq & \left[\neg B\right];\mathsf{Del}(x);\mathsf{Del}(y) \sqcap \left[B\right];\mathsf{Del}(x);y := e;S;\mu.\mathcal{F};\mathsf{Del}(y) \\
\langle(4.8),\ (4.9)\rangle & = & \left[\neg B\right];y := x;\mathsf{Del}(x);\mathsf{Del}(y) \sqcap \\
 & & \quad \left[B\right];y := x;\mathsf{Del}(x);y := e;S;\mu.\mathcal{F};\mathsf{Del}(y) \\
\langle(4.33),\ \text{Prop. } 4.3\rangle & = & y := x;\mathsf{Del}(x);\left[\neg B\right];\mathsf{Del}(y) \sqcap \\
 & & y := x;\mathsf{Del}(x);\left[B\right];y := e;S;\mu.\mathcal{F};\mathsf{Del}(y) \\
\langle\text{conjunctivity\&}(4.2)\rangle & = & y := x;\mathsf{Del}(x);\left(\left[\neg B\right] \sqcap \left[B\right];y := e;S;\mu.\mathcal{F}\right);\mathsf{Del}(y) \\
 & = & y := x;\mathsf{Del}(x);\mu.\mathcal{F};\mathsf{Del}(y)
\end{array}$$

□

Proof of lemma 6.1(b). Defining $\mathcal{F}(U) \triangleq \left[\neg B\right] \sqcap \left[B\right];T;S;U$ and $\mathcal{G}(U) \triangleq \left[\neg B\right] \sqcap \left[B\right];T';U$, we will show that $S;\mathcal{F}^{\alpha}(\mathbf{abort}) \sqsubseteq \mu.\mathcal{G};S$ holds for any ordinal α by transfinite induction. For the case that α is a non-limit ordinal, we derive as follows.

$$\begin{array}{rcl}
S;\mathcal{F}^{\alpha+1}(\mathbf{abort}) & = & S;\left(\left[\neg B\right] \sqcap \left[B\right];T;S;\mathcal{F}^{\alpha}(\mathbf{abort})\right) \\
\langle(4.3)\rangle & \sqsubseteq & S;\left[\neg B\right] \sqcap S;\left[B\right];T;S;\mathcal{F}^{\alpha}(\mathbf{abort}) \\
\langle\text{premises}\rangle & \sqsubseteq & \left[\neg B\right];S \sqcap \left[B\right];T';S;\mathcal{F}^{\alpha}(\mathbf{abort}) \\
\langle\text{induction}\rangle & \sqsubseteq & \left[\neg B\right];S \sqcap \left[B\right];T';\mu.\mathcal{G};S \\
\langle(4.2)\rangle & = & \left(\left[\neg B\right] \sqcap \left[B\right];T';\mu.\mathcal{G}\right);S \\
 & = & \mu.\mathcal{G};S
\end{array}$$

The case for α being a limit ordinal follows as below.

$$\begin{array}{rcl}
S;\mathcal{F}^{\alpha}(\mathbf{abort}) & = & S;\Big(\bigsqcup_{\beta<\alpha} \mathcal{F}^{\beta}(\mathbf{abort})\Big) \\
\langle S\text{: disjunctive}\rangle & = & \bigsqcup_{\beta<\alpha} \left(S;\mathcal{F}^{\beta}(\mathbf{abort})\right) \\
\langle\text{induction}\rangle & \sqsubseteq & \bigsqcup_{\beta<\alpha} \left(\mu.\mathcal{G};S\right) \ = \ \mu.\mathcal{G};S
\end{array}$$

□