CORE
Metadata, citation and similar papers at core.ac.uk
Provided by The IT University of Copenhagen's Repository

Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Modular Verification of Linked Lists with Views via Separation Logic

Jonas Braband Jensen[a] Lars Birkedal[a] Peter Sestoft[a]

a. IT University of Copenhagen

Abstract We present a separation logic specification and verification of *linked lists with views*, a data structure from the C5 collection library for .NET. A *view* is a generalization of the well-known concept of an iterator. Linked lists with views form an interesting case study for verification since they allow mutation through multiple, possibly overlapping, views of the same underlying list. For modularity, we build on a fragment of higher-order separation logic and use abstract predicates to give a specification with respect to which clients can be proved correct. We introduce a novel mathematical model of lists with views, and formulate succinct modular abstract specifications of the operations on the data structure. To show that the concrete implementation realizes the specification, we use fractional permissions in a novel way to capture the sharing of data between views and their underlying list.

We conclude by suggesting directions for future research that arose from conducting this case study.

Keywords Separation logic, formal verification, modularity

1 Introduction

Separation logic [Rey02] is a generalization of Hoare logic better suited for reasoning about heap data in imperative programming. In particular, the logic's separating conjunction connective directly supports reasoning about situations where heap-allocated data can be separated into non-overlapping regions. The challenging applications of separation logic are therefore those involving partially overlapping data structures. List iterators provide one example of such structures, and they have been studied extensively in connection with separation logic [KAB+09, BRZ07, HH08].

Here we investigate the *linked list with views* (LLWV) data structure from the C5 library [KS06] of collections for the .NET framework. Where an iterator can be thought of as marking a current *position* in a list, a view more generally marks a list *segment*, so an iterator is just a special case of a view (of length zero). A list may have multiple views, the views may overlap, and modifications to the underlying list show through the views and vice versa; for more details, see Section 2.

Jonas Braband Jensen, Lars Birkedal, Peter Sestoft. *Modular Verification of Linked Lists with Views via Separation Logic*. In Journal of Object Technology, vol. 10, 2011, pages 2:1–20.
doi:10.5381/jot.2011.10.1.a2

Hence views provide a much more powerful mechanism than iterators but also pose new challenges for verification. In particular, the co-dependencies between a list and its views are typically implemented by cyclic pointer structures, and there is no “obvious” mathematical model of a linked list with views. We find that this makes the data structure a challenging and compelling case study for specification and verification with separation logic and related approaches.

1.1 Related Work

Hoare pioneered the proof method of relating a concrete (object-oriented) implementation to an abstract (functional, mathematical) implementation [Hoa72]; we use the same technique here. We also use the concepts of precondition and postcondition, which are due to Dijkstra [Dij76] and which form the basis for Meyer’s design-by-contract methodology [Mey92]. However, we do not use class invariants, because they appear to fall short when, as in the case of linked lists with views, there is no hierarchical “ownership” relation among the objects making up a data structure [Par07]. Hence our formalization is not immediately expressible in contemporary frameworks such as the Java Modeling Language [CKLP06] or .NET Code Contracts [FBL10].

The formalization and proof of lists with views, presented in this paper, uses separation logic and has many similarities with separation logic formalizations of iterators. The iterators from the Java standard library seem to be the most popular objects of study [KAB+09, Par05, HH08, BRZ07]. In contrast to the list views discussed here, such iterators become invalid after structural modification to the underlying list, and so it becomes an important part of the specification to capture the protocol that constrains the permitted order of method calls.

Krishnaswami et al. [KAB+09] use higher-order separation logic to give an elegant specification of iterators. It allows multiple iterators at the same time, but iterators are read-only.

Parkinson [Par05] specifies iterators in first-order separation logic, instead using *counting permissions* to share the list between multiple iterators. Again, modification of the list through iterators is not considered.

Haack & Hurlin [HH08] use *fractional permissions* to give a specification that allows both multiple iterators and (limited) modification of the list through iterators. The techniques used to achieve this have similarities to what we present in Section 4.3.

In contrast to iterators, it is always well-defined how views behave after the underlying list is modified.

1.2 Significance for Object-Oriented Languages

The present work focuses on a particular aspect of object-oriented languages: *local update* (by assignment `x.f=e` to object fields) combined with *sharing* (by having multiple references `x` and `y` denoting the same object). This combination means that multiple surface “names” `x.f` and `y.f` denote the same updatable data structure, which makes object-oriented programs hard to reason about using the basically substitution-based approach of Hoare logic. This is not just a formal problem, but also a challenge to informal program understanding, as evidenced by the recent emphasis on the virtues of immutable data; see for instance Josh Bloch’s admonishment “Minimize Mutability” [Blo08, Item 15].

In this paper we use separation logic to handle the combination of field update and sharing. We also assume the object-oriented virtue of *encapsulation*: a client

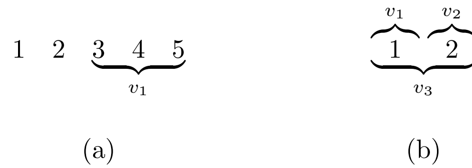

Figure 1 – (a) Linked list with one view. (b) Linked list with two 1-item views and one 2-item view.

cannot arbitrarily access the internals of objects on which it operates. This is of course essential for preservation of invariants and hence for correctness.

On the other hand, we do not address inheritance and virtual methods. Although these are important and challenging features, we believe they are rather orthogonal to the formalization here, and related work has devised one way in which to handle them formally in the context of separation logic [PB08].

1.3 Outline

Section 2 introduces linked lists with views as they are seen from the perspective of a client, and Section 3 describes how they were implemented in this case study.

We give our specification in Section 4 in a fragment of intuitionistic higher-order separation logic [BBTS05], using abstract predicates [PB05], such that clients can be verified without revealing information about the concrete implementation of the data structure. The overall idea in this approach is to use a predicate $\mathsf{L}(x, \alpha)$ that relates a data structure pointer x in the implementation to a mathematical object α that models the data structure abstractly. Partial-correctness specifications for each method f on x are then expressed in terms of this predicate; they are typically of the form $\{\mathsf{L}(x, \alpha)\}\, x.f(\ldots)\, \{\mathsf{L}(x, \alpha')\}$.

We present the concrete realization of the abstract predicates in Section 5, using fractional permissions.

Section 6 presents and discusses the alternative models we have considered.

An earlier version of the results in this paper was published at the FTfJP'10 workshop.

2 Linked Lists With Views

The linked list data structure is well known and is a standard example of separation logic specification and proof. Here we consider *linked lists with views*, a data structure designed as part of the C5 collection library [KS06] that provides several new verification challenges. A *view* is a window on a contiguous segment of a list; a list can have multiple, possibly overlapping, views; see Figure 1 for two examples. An update to a view affects the underlying list as well as overlapping views; and an update to the underlying list may affect multiple views. Finally, a view can be slid left and right along a list, and can be grown and shrunk. A list and its views are closely intertwined, and the update semantics means that there is no "obviously right" model in terms of standard mathematical structures such as sequences, trees and sets.

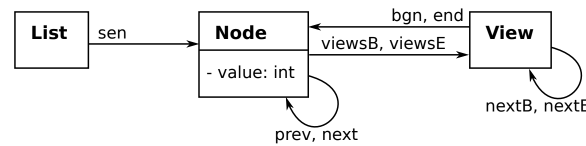

Figure 2 – Class diagram of the implementation.

But why consider the intricacies of these list views at all? Because views have several interesting applications. With views, one can give linked lists and array lists a single common interface, while avoiding the explicit manipulation of internal linked list nodes and hence raising questions of the list's structural integrity, yet provide efficient item access in linked lists, via views instead of item indices.

In fact, a zero-item view is a cursor that points *between* (or before or after) list items, and there are $n + 1$ distinct zero-item views on an n-item list; whereas a one-item view is a cursor that points *at* a list item, and there are n distinct one-item views on an n-item list. Just for this reason, views may be beneficial even for array list algorithms where it is often unclear whether an index i is meant to point *before*, *at*, or *after* the i'th item.

Moreover, a view implements the same interface and supports the same operations as linked lists and array lists, so "sort this particular list segment" can be decomposed into "create a view comprising this particular list segment" and "sort the view". This orthogonality considerably reduces the number of operations that the list interface must exhibit: a single "search view" operation replaces "search entire list", "search list starting at index i", "search list starting at index i and ending at item $i + n$". Furthermore, views (and lists) can be looked at "backwards" so "search view" actually represents $3 \cdot 2 = 6$ different search functions. The same holds for other kinds of list traversal, clearing, shuffling, and so on. Thus views lead to a considerably leaner and more regular list library design.

Apart from the use as between-item and at-item cursors, and to achieve orthogonality of list operations, our updatable slidable views enable elegant implementation of some algorithms such as Graham's point elimination scan when computing a 2D convex hull [KS06, section 11.3]; here three-item views are called for.

The actual C5 data structures are generic, or parametrically polymorphic, in the item type. In this paper we assume for simplicity that list items are just integers, but our proofs do not rely on this fact, and the specification and verification can be extended using higher-order verification techniques for generics [SBP10].

3 Implementation

A class diagram of the implementation data structures is shown in Figure 2. An LLWV has class List; it uses a circular doubly-linked list of Node objects internally to hold the list items. Each node n has a field value that holds the item value; fields prev and next for the doubly-linked list representation; and fields viewsB and viewsE that hold references to two singly-linked lists of View objects: a list of those views that begin just after n and a list of those views that end just before n.

It is an invariant of the data structure that if v.bgn points to n, then n.viewsB points to a list with next-pointers nextB in which v occurs exactly once; and similarly for end/viewsE/nextE.

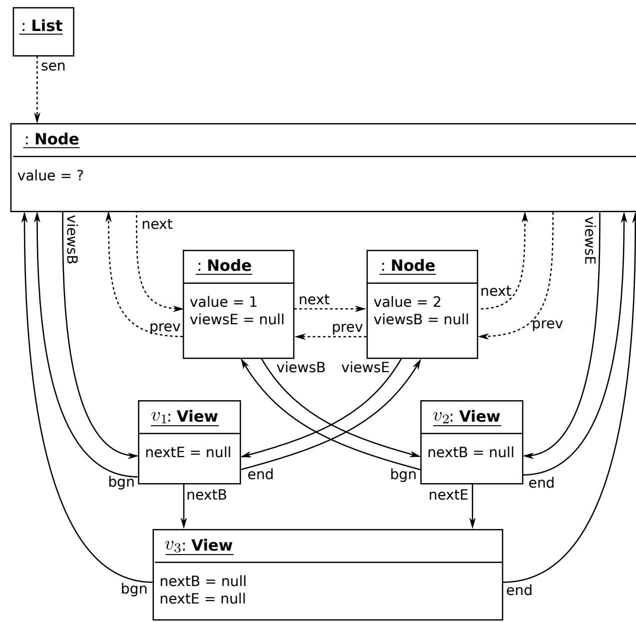

Figure 3 – Object diagram showing the heap layout of the example LLWV from Figure 1(b), with two items and three partially-overlapping views. Dashed arrows make up the list of items, while solid arrows are pointers maintained to support views.

There is a *sentinel* node, whose value we ignore, at the beginning and end of the list. In fact, a single Node object can be used as both start sentinel and end sentinel since the sets of fields used in these two roles are disjoint.

As an example of such a data structure, Figure 3 depicts a possible heap representation of the LLWV from Figure 1(b).

The actual code we have verified is not the original C# code from the C5 library but a Java implementation that has been written from scratch for verification purposes. It captures the essence of what makes LLWV interesting to verify without containing all the bells and whistles that would make it pleasant to use in an engineering context. The most important differences are discussed in Section 5.1.

4 Abstract Specification

This section presents a mathematical model of LLWV and specifications of its methods using abstract predicates. Verification of clients will rely only on these, not on the actual implementation of LLWV.

4.1 Model

Our specifications will revolve around a predicate $\mathsf{L}(l, \alpha)$ that relates a LLWV l (a pointer in the implementation) to a model "bex-list" α. A *bex-list* describes the list items along with all views defined on the list. It seems necessary to join these objects together in one monolithic model because the behaviour of views is defined such that a list and its views can affect each other to a great extent.

The actual definition of the predicate $\mathsf{L}(l, \alpha)$ is shown in Section 5 and is used only when proving an implementation correct. The definition is hidden from clients to prevent them from depending on implementation details [PB05].

Let View be the class of views. Then bex-lists α, β, γ are defined as follows, where "::" denotes list construction:

$$\alpha, \beta, \gamma ::= \epsilon \mid \mathsf{B}\, b :: \alpha \mid \mathsf{E}\, e :: \alpha \mid \mathsf{X}\, x :: \alpha$$
$$b, e \subset_{\mathsf{fin}} \text{pointers to } \mathsf{View}$$
$$x \in \mathbb{Z}$$

Intuitively, $\mathsf{B}\, b$ means that the views in set b begin at this position in the list. Similarly, $\mathsf{E}\, e$ means that the views in e end at this position in the list, and $\mathsf{X}\, x$ means that the item x is stored at this position in the list. Such a list element $\mathsf{B}\, b$, $\mathsf{E}\, e$ or $\mathsf{X}\, x$ is called a *bex*.

We will always want the bexes to appear in the order $\mathsf{B}, \mathsf{E}, \mathsf{X}, \mathsf{B}, \mathsf{E}, \mathsf{X}, \ldots$. To enforce this, we define a predicate $\mathsf{ord}(\alpha, t, t')$, where $t, t' \in \{\mathsf{B}, \mathsf{E}, \mathsf{X}\}$, expressing that α is an ordered bex-list starting with a bex of constructor t and ending just before a bex of constructor t'. Formally, let ord be the least predicate satisfying

$$\mathsf{ord}(\epsilon, t, t)$$
$$\mathsf{ord}(\mathsf{B}\, b :: \alpha, \mathsf{B}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{E}, t)$$
$$\mathsf{ord}(\mathsf{E}\, e :: \alpha, \mathsf{E}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{X}, t)$$
$$\mathsf{ord}(\mathsf{X}\, x :: \alpha, \mathsf{X}, t) \Longleftarrow \mathsf{ord}(\alpha, \mathsf{B}, t)$$

Note that we here used the symbols $\mathsf{B}, \mathsf{E}, \mathsf{X}$ both as (unary) constructors and as (nullary) tags.

Concatenation is defined as usual for cons-based lists and is written $\alpha\beta$. It can be shown by induction on α that

$$\exists t'.\ \mathsf{ord}(\alpha, t, t') \wedge \mathsf{ord}(\beta, t', t'') \iff \mathsf{ord}(\alpha\beta, t, t'')$$

An empty LLWV is modelled by a bex-list $\mathsf{B}\, b :: \mathsf{E}\, e :: \epsilon$, abbreviated *be*. A singleton LLWV is modelled by a bex-list $b_1 e_1 x_1 b_2 e_2$. In general, a LLWV is modelled by an ordered bex-list that begins with a B and ends with an E (i.e. just before an X). We call such lists *well-formed*. We will also need the notion of the length of a bex-list α, written $|\alpha|$. In summary,

$$\mathsf{wf}(\alpha) \triangleq \mathsf{ord}(\alpha, \mathsf{B}, \mathsf{X})$$
$$|\alpha| \triangleq \text{number of X's in } \alpha$$

The bex-list may not seem like the most intuitive or obvious construction, but it will turn out that specification of the public methods on linked lists with views becomes very simple when using it. Some other models we tried before choosing the bex-list model are discussed in Section 6.

4.2 Operations on Lists

The List class has methods for the list operations one would expect: insertion, removal, subscripting, size query, etc. We now discuss the specifications of the most important operations; Figure 4 gives a summary of all the specifications.

A simple and typical specification is that of $\mathsf{setValue}(i, x')$, which replaces the item at index $i \geq 0$ in list l by x':

$$\{\, \mathsf{L}(l, \alpha x \gamma) \,\}\ l.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{L}(l, \alpha x' \gamma) \,\}$$

This specification says that, provided list l is described by the bex-list $\alpha x \gamma$ before the call, then after the call $l.\mathsf{setValue}(|\alpha|, x')$, list l is described by the bex-list $\alpha x' \gamma$. That is, all list items and views remain the same, except that the item at index $i = |\alpha|$ has been replaced by x'.

Note how part of the precondition is made implicit by restricting the first method argument to have the form $|\alpha|$, instead of an arbitary integer i. Together with the assertion $\mathsf{L}(l, \alpha x \gamma)$ about the shape of the list, this restriction ensures that the item index $|\alpha|$ is legal for the list.

The client should not know the exact definition of L, but it is part of the specification that $\mathsf{L}(l, \alpha)$ implies $\mathsf{wf}(\alpha)$.

It is an important detail that pre- and postconditions are both expressed in terms of equations in bex-lists and their lengths. This makes it easy for sequential client code to establish that the postcondition of one call implies the precondition of the next.

Removal and insertion can also be defined just in terms of bex-list equalities and operations on finite sets. Here, the $\uplus$ operator is a partial version of set union $\cup$ that is defined only for disjoint sets.

$$\{\, \mathsf{L}(l, \alpha b'\, e x b\, e' \gamma) \,\}\ l.\mathsf{remove}(|\alpha|)\ \{\, \mathsf{L}(l, \alpha (b' \uplus b)(e' \uplus e) \gamma) \,\}$$

$$\{\, \mathsf{L}(l, \alpha b e \gamma) \,\}\ l.\mathsf{insert}(|\alpha|, x)\ \{\, \mathsf{L}(l, \alpha (\mathsf{B}(b \cap e)) e x (\mathsf{B}(b \setminus e))(\mathsf{E}\emptyset) \gamma) \,\}$$

Note that they both preserve well-formedness (and therefore ordering) of the bex-list. The complicated-looking postcondition of insert captures exactly the rules of how views that begin or end around the point of insertion are affected [KS06, Jen10].

4.3 Operations on Views

A new view is created on a list l by calling $l.\mathsf{view}$, specified in Figure 4. We use the special variable ret for the return value and use the notation b^v to mean the partial operation $b \cup \{v\}$ where $v \notin b$.

There is an abstract predicate $\mathsf{V}(v, \alpha)$ that is like L for most purposes; it says that view v is described by the bex-list α. As with L, clients are guaranteed that $\mathsf{V}(v, \alpha)$ implies $\mathsf{wf}(\alpha)$. As stated in Figure 4, the methods that work on both lists and views have identical specifications except that l and L are replaced by v and V in the case of views. For example, the specification of $\mathsf{setValue}$ on a view would be

$$\{\, \mathsf{V}(v, \alpha x \gamma) \,\}\ v.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{V}(v, \alpha x' \gamma) \,\}$$

The V predicate is not given directly in any method postcondition. Instead, the client is given a guarantee that the following implication is valid:

$$\mathsf{L}(l, \alpha b^v \beta e^v \gamma) \implies \mathsf{V}(v, b \beta e) \ast \forall b', \beta', e'. \left[\mathsf{V}(v, b' \beta' e') \mathrel{-\!\!\ast} \mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma) \right] \quad (1)$$

In words, this expresses that a heap containing a LLWV l with a view v can be separated into two parts, say, h and h'. Heap h satisfies the V predicate and can

$$\{\,\mathsf{true}\,\}\ \mathsf{new\ List}()\ \{\,\mathsf{L}(ret, (\mathsf{B}\,\emptyset)\,(\mathsf{E}\,\emptyset))\,\}$$

The following methods are also available on views, with the same specification except that l and L are replaced by v and V.

$\{\,\mathsf{L}(l,\alpha)\,\}$	$l.\mathsf{count}()$	$\{\,\mathsf{L}(l,\alpha) \land ret = \lvert\alpha\rvert\,\}$
$\{\,\mathsf{L}(l,\alpha x\gamma)\,\}$	$l.\mathsf{getValue}(\lvert\alpha\rvert)$	$\{\,\mathsf{L}(l,\alpha x\gamma) \land ret = x\,\}$
$\{\,\mathsf{L}(l,\alpha x\gamma)\,\}$	$l.\mathsf{setValue}(\lvert\alpha\rvert, x')$	$\{\,\mathsf{L}(l,\alpha x'\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b\beta e\gamma)\,\}$	$l.\mathsf{view}(\lvert\alpha\rvert, \lvert\beta\rvert)$	$\{\,\mathsf{L}(l,\alpha b^{ret}\beta e^{ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b'\,exb\,e'\gamma)\,\}$	$l.\mathsf{remove}(\lvert\alpha\rvert)$	$\{\,\mathsf{L}(l,\alpha(b' \uplus b)(e' \uplus e)\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha be\gamma)\,\}$	$l.\mathsf{insert}(\lvert\alpha\rvert, x)$	$\{\,\mathsf{L}(l,\alpha(\mathsf{B}\,(b\cap e))ex(\mathsf{B}\,(b\setminus e))(\mathsf{E}\,\emptyset)\gamma)\,\}$

Methods specific to views:

$$\begin{gathered}\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \land \alpha b\beta e\gamma = \alpha' b'\beta' e'\gamma'\,\}\\ v.\mathsf{slide}(\lvert\alpha'\rvert - \lvert\alpha\rvert,\ \lvert\beta'\rvert)\\ \{\,\mathsf{L}(l,\alpha' b'^v\beta' e'^v\gamma')\,\}\end{gathered}$$

$$\begin{gathered}\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \land \alpha b\beta e\gamma = \alpha' b'\beta' e'\gamma' \land \lvert\beta'\rvert = \lvert\beta\rvert\,\}\\ v.\mathsf{slide}(\lvert\alpha'\rvert - \lvert\alpha\rvert)\\ \{\,\mathsf{L}(l,\alpha' b'^v\beta' e'^v\gamma')\,\}\end{gathered}$$

$\{\,\mathsf{L}(l,\alpha b^v\beta e^v\gamma)\,\}$	$v.\mathsf{dispose}()$	$\{\,\mathsf{L}(l,\alpha b\beta e\gamma)\,\}$
$\{\,\mathsf{V}(u,\alpha b^v\beta e^v\gamma)\,\}$	$v.\mathsf{dispose}()$	$\{\,\mathsf{V}(u,\alpha b\beta e\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha e^v\gamma)\,\}$	$v.\mathsf{atEnd}(l)$	$\{\,\mathsf{L}(l,\alpha e^v\gamma) \land ret = (\gamma = \epsilon)\,\}$
$\{\,\mathsf{L}(l,\alpha b^u\beta e^v\gamma)\,\}$	$u.\mathsf{span}(v)$	$\{\,\mathsf{L}(l,\alpha b^{u,ret}\beta e^{v,ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,b\alpha e^v\gamma)\,\}$	$l.\mathsf{span}(v)$	$\{\,\mathsf{L}(l,b^{ret}\alpha e^{v,ret}\gamma)\,\}$
$\{\,\mathsf{L}(l,\alpha b^v\gamma e)\,\}$	$v.\mathsf{span}(l)$	$\{\,\mathsf{L}(l,\alpha b^{v,ret}\gamma e^{ret}\,\}$

Guarantees about predicates:

$$\mathsf{L}(l,\alpha) \implies \mathsf{wf}(\alpha), \qquad \mathsf{V}(v,\alpha) \implies \mathsf{wf}(\alpha),$$
$$\mathsf{L}(l,\alpha b^v\beta e^v\gamma) \implies \mathsf{V}(v,b\beta e) * \forall b',\beta',e'.\ \Big[\mathsf{V}(v,b'\beta' e') \mathrel{-\!\!*} \mathsf{L}(l,\alpha b'^v\beta' e'^v\gamma)\Big]$$

Figure 4 – Summary of specifications. The notation b^v means $b \cup \{v\}$ where $v \notin b$.

thus be used for calling the various methods on views, such as $\mathsf{setValue}$ in Section 4.2, leading to a V-assertion for the same v but with a different bex-list. Heap h' satisfies that given any such modified bex-list $b'\beta'e'$, if h' is extended with a heap in which view v is described by $b'\beta'e'$, then this extended heap describes the original list l except that the sublist delimited by view v has been modified.

The verifier, i.e. the person or heuristic attempting to verify the program, can use (1) to convert from L to V at any convenient time. To get back to L again, he can use the *separating modus ponens* rule: $P * (P \mathrel{-\!\!*} Q) \implies Q$. This is not too different from how the frame rule is used in separation logic in general; in fact, the following specification-logic rule follows from (1).

$$\frac{\{\,\mathsf{V}(v, b\beta e)\,\}\ c\ \{\,\mathsf{V}(v, b'\beta'e')\,\}}{\{\,\mathsf{L}(l, \alpha b^v \beta e^v \gamma)\,\}\ c\ \{\,\mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma)\,\}} \tag{2}$$

In words, if command c changes a view v from $b\beta e$ to $b'\beta'e'$, then if v is a view on some underlying list l described by $\alpha b^v \beta e^v \gamma$, then c will also change list l to $\alpha b'^v \beta' e'^v \gamma$. In particular, the list "tails" α and γ are unaffected by c.

Hence (2) reads as a kind of frame rule, where α and γ constitute the frame that is disregarded while verifying c. Like the frame rule, its application happens at the discretion of the verifier rather than being driven by the program.

Note that (1) is more general than (2) since the conversions between L and V do not have to follow the nesting discipline of a tree in (1).

5 Verification of Implementation

We saw the implementation of the LLWV data structure in Section 3 and the specifications and guarantees involving the L and V predicates in Section 4. To tie these together, we must give the definitions of L and V. The I predicate (I as in Items) will be a key ingredient in this.

We define L as asserting the existence of a sentinel node n_s, which marks both the beginning and ending of the list:

$$\mathsf{L}(l, \alpha) \triangleq \mathsf{wf}(\alpha) * \exists n_s.\ l.\mathsf{sen} \mapsto n_s * \mathsf{I}(n_s, n_s, \alpha)$$

For an ordered bex-list α and nodes n and n', $\mathsf{I}(\alpha, n, n')$ asserts what must hold of a heap that spans α between n and n'. It does so by a case analysis on whether α is empty or starts with B, E or X. It is a convenient property of the bex-list model that no indirection is needed here: the bex-list as seen by the client corresponds so closely with the heap layout that I can be syntax-directed on α.

Another convenient property is that I admits an excellent correspondence between separation on the heap and concatenation of bex-lists:

$$\mathsf{I}(n, n'', \alpha\beta) \iff \exists n'.\ \mathsf{I}(n, n', \alpha) * \mathsf{I}(n', n'', \beta) \tag{3}$$

The definition of I and all predicates required by it is shown in Figure 5.

One might easily be tempted to define $\mathsf{V}(v, \alpha)$ as

$$\mathsf{wf}(\alpha) * \exists b, \beta, e.\ \alpha = b\beta e * \exists n_b, n_e.\ \mathsf{I}(n_b, n_e, b^v \beta e^v) \tag{4}$$

Expanding the I predicate will lead to the assertions $v.\mathsf{bgn} \mapsto n_b * v.\mathsf{end} \mapsto n_e$ needed by methods on the view as a starting points for accessing its items.

$$
\begin{aligned}
\mathsf{L}(l,\alpha) &\triangleq \mathsf{wf}(\alpha) * \exists n_\mathrm{s}.\ l.\mathsf{sen} \mapsto n_\mathrm{s} * \mathsf{I}(n_\mathrm{s}, n_\mathrm{s}, \alpha)\\
\mathsf{I}(n,n,\epsilon) &\triangleq \mathsf{true}\\
\mathsf{I}(n,n'',\mathsf{B}\,b :: \alpha) &\triangleq \mathsf{I_B}(n,b) * \mathsf{I}(n,n'',\alpha)\\
\mathsf{I}(n,n'',\mathsf{E}\,e :: \alpha) &\triangleq \exists n'.\ \mathsf{N}(n,n') * \mathsf{I_E}(n',e) * \mathsf{I}(n',n'',\alpha)\\
\mathsf{I}(n,n'',\mathsf{X}\,x :: \alpha) &\triangleq \mathsf{I_X}(n,x) * \mathsf{I}(n,n'',\alpha)\\
\mathsf{N}(n,n') &\triangleq n.\mathsf{next} \mapsto n' * n'.\mathsf{prev} \mapsto n\\
\mathsf{I_B}(n,b) &\triangleq \mathsf{BList}(n,b) * \mathop{\circledast}_{v\in b} v.\mathsf{bgn} \mapsto n\\
\mathsf{I_E}(n,e) &\triangleq \mathsf{EList}(n,e) * \mathop{\circledast}_{v\in e} v.\mathsf{end} \mapsto n\\
\mathsf{I_X}(n,x) &\triangleq n.\mathsf{value} \mapsto x\\
\mathsf{BList}(n,b) &\triangleq \exists v_\mathrm{h}.\ n.\mathsf{viewsB} \mapsto v_\mathrm{h} * \mathsf{BSeg}(v_\mathrm{h}, \mathsf{null}, b)\\
\mathsf{BSeg}(v_\mathrm{t}, v_\mathrm{t}, \emptyset) &\triangleq \mathsf{true}\\
\mathsf{BSeg}(v_1, v_\mathrm{t}, b^v) &\triangleq v_1 \in b^v * \exists v_2.\ v_1.\mathsf{nextB} \mapsto v_2 * \mathsf{BSeg}(v_2, v_\mathrm{t}, b^v \setminus \{v_1\})
\end{aligned}
$$

$\mathsf{EList}, \mathsf{ESeg}$ are defined like $\mathsf{BList}, \mathsf{BSeg}$.

Figure 5 – Definition of L and related predicates. The $\circledast$ operator is *iterated separating conjunction* [Rey02].

But for such a definition of V, Equation (1) will not hold. The issue is that it permits the asserter of V to write to bgn and end, allowing him to "unhook" the view from its underlying list and place it somewhere else in memory. The challenge is to somehow ensure that the sentinel nodes have not changed when it is time to convert the V predicate back to L.

One way to solve this problem could be to provide clients with the weaker but often sufficient Equation (2) instead of (1). That could be be proved by induction over the command c, showing that c will not modify $v.\mathsf{bgn}$ or $v.\mathsf{end}$ because any write to these fields would either be denied because the fields are private, or it would happen through one of the methods of View, whose specifications would have to be strengthened to guarantee that they do not modify to those fields either.

Clearly, this approach is problematic. It requires reasoning about field access modifiers in the logic, which has not been formalized in the separation logics found in the literature to date. It also provides a weaker guarantee to the client, and it lacks modularity because we cannot add or change methods without invalidating the proof.

To find a definition of V that validates (1), look back to the original issue: the asserter of V must be able to read the bgn and end fields but not write them. A popular approach to expressing this is to amend the assertion logic with *fractional permissions* [BCOP05, BRZ07, HH08], a technique borrowed from concurrent programming that turns out to be useful for sequential programs as well [BRZ07, HH08].

In separation logic with fractional permissions, the *points-to* assertion $x.f \mapsto a$ is extended to read $x.f \overset{z}{\mapsto} a$, where $0 < z \leq 1$. A permission $z = 1$ gives read/write access to the field $x.f$, while any smaller permission gives read-only access. The assertion logic is then defined such that the following is valid: $x.f \overset{z}{\mapsto} a * x.f \overset{z'}{\mapsto} a \iff x.f \xmapsto{z+z'} a$.

Now we can give a definition of V that works by modifying (4) to take away half

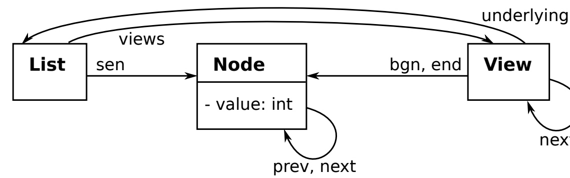

Figure 6 – A class diagram that, compared to Figure 2, more closely resembles the data structure found in the original C5 library [KS06].

of the permissions to the bgn and end fields:

$$\mathsf{V}(v,\alpha) \triangleq \mathsf{wf}(\alpha) * \exists b,\beta,e.\ \alpha = b\beta e\ * \exists n_\mathrm{b}, n_\mathrm{e}. \\ \left[v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e} \multimap\!\circledast \mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, b^v \beta e^v)\right]$$

Here, septraction [CPV07] ($\multimap\!\circledast$) is used to "subtract" the permissions on its left side from those on its right side.

Septraction is not essential for this definition, but it does make it much more elegant than it would have been otherwise. Since we are using an intuitionistic separation logic, the standard definitions of septraction developed for classical versions of the logic do not work. In Appendix A, we describe a definition of septraction that works in intuitionistic separation logic.

With the above definition of V, Equation (1) can be proved valid – see Appendix B. It is an important point that the client does not need to know that fractional permissions are being used behind the scenes; clients may reason entirely as if there were no fractions.

5.1 Discussion

It turns out that (1) and (2) can be generalized to treat more than one view. For example, given a list with two non-overlapping views v_1 and v_2, one can mutate those views independently ($\mathsf{V}(v_1, \beta_1) * \mathsf{V}(v_2, \beta_2)$) and later establish that L holds for their underlying list and a suitably-modified bex-list.

With the implementation we have discussed so far, this generalization is straightforward to prove; see Appendix B. The data structure used in the original C5 code is different, however: it has a *global* list of views for the whole LLWV rather than for each node. This is illustrated in Figure 6. The two implementations are observationally indistinguishable through their public interface, but the implementation with a global view-list does not seem to allow defining a V-predicate that admits the assertion $\mathsf{V}(v_1, \beta_1) * \mathsf{V}(v_2, \beta_2)$ for non-overlapping views.

This is because operations on v_1 and v_2 such as item insertion and removal will have to traverse the same global list of views; operations to create and dispose views are even going to modify this list, so it cannot just be shared read-only using fractional permissions.

Since a separation logic assertion describes concrete heap contents, the validity of assertions such as (1) depends only on the choice of data structure in the implementation, not on the code in methods. On the other hand, in object-oriented languages there is a difference between (a) objects that cannot interfere with each other in any observable way and (b) objects that have disjoint heap footprints. Clearly, (b) implies (a), but the converse does not hold because of encapsulation: interference can be

Method	S	I	V	F
$+$List::init	✓	✓		
$+l$.count	✓	✓		
$+l$.getValue	✓	✓	✓	
$+l$.isEmpty	✓	✓		
$+l$.insert	✓	✓	✓[2]	
$+l$.insertFirst		✓		
$+l$.insertLast		✓		
$+l$.remove	✓	✓	✓	
$+l$.removeFirst		✓		
$+l$.removeLast		✓		
$+l$.setValue	✓			
$+l$.span	✓			✓
$+l$.view	✓	✓	✓	
$-l$.getNode	✓	✓	✓	
$-l$.insertNode	✓	✓		✓
$-$View::init/0	✓	✓		
$-$View::init/2	✓	✓	✓[2]	
$+v$.atEnd	✓			
$+v$.count	✓	✓		
$+v$.dispose	✓			
$+v$.getValue	✓	✓		✓
$+v$.isEmpty	✓	✓		
$+v$.insert	✓	✓		✓

Method	S	I	V	F
$+v$.insertFirst		✓		
$+v$.insertLast		✓		
$+v$.remove	✓	✓		✓
$+v$.removeFirst		✓		
$+v$.removeLast		✓		
$+v$.setValue	✓			✓
$+v$.slide/1	✓	✓		✓
$+v$.slide/2	✓	✓		✓
$+v$.span	✓			✓
$+v$.view	✓	✓		✓
$-v$.getNode	✓	✓		✓
$-v$.insertNode	✓	✓		✓
$-$Node::init	✓	✓		
$-n$.addViewB	✓	✓		✓
$-n$.addViewE	✓	✓		
$-n$.insertAfter	✓	✓		✓
$-n$.moveViewsToB	✓	✓	✓	
$-n$.moveViewsToE/1	✓	✓	✓[1]	
$-n$.moveViewsToE/2	✓	✓		✓
$-n$.remove	✓	✓	✓	
$-n$.removeViewB	✓	✓		
$-n$.removeViewE	✓	✓		

Notes:
1: Has been verified in the sense that its twin method with b's instead of e's has been verified.
2: The body of this method has been verified, but it contains calls to unverified methods.

Table 1 – Overview of what methods have been specified (S), implemented (I), verified (V), and which would make non-trivial future work (F). Public methods are prefixed with $+$, private methods are prefixed with $-$, and overloaded methods are suffixed with $/n$ to refer to the n-argument version.

observed only through a data structure's public methods, whereas the actual sharing in the heap is described by its (private) fields. In separation logic as we use it here, assertions about objects that are "observationally separate" cannot be separated by the $*$ connective, although this connective is critical for reasoning.

Therefore, a guideline for writing code to be modularly verified with separation logic is to design data structures such that observationally separate parts of the data structure are also disjoint on the heap, whenever possible.

Finding a good way to lift this limitation is likely to be crucial in reasoning about real-world code. The ideas presented in the recent work of Dinsdale-Young et al. [DYDG+10] look promising in this respect. The intuition seems to be that access to shared data is governed by a protocol, and this protocol can be as simple as requiring read-only access or perhaps as complex as required to solve this problem.

5.2 Method bodies

So far we have focused on method specifications and heap layout. To have a complete verified library of LLWV, one of course needs to write the Java code implementing each method and prove that it satisfies its specification in a dialect of separation logic that has been shown sound with respect to the programming language semantics. To get a feeling for whether the specifications are practically useful, sample client code should be verified.

Table 1 summarises which methods were specified, implemented and verified, including several private methods in class Node that we do not discuss here [Jen10]. The main accomplishment is that List::remove and everything it calls has been verified. Two sample clients have been verified: an implementation of Bubble Sort, sliding views of length 2 across the list, has been verified for safety, and a toy example that uses views to duplicate and increment every list item has been verified for correctness.

Method body proofs were done by hand and are presented as code interleaved with assertions in [Jen10]. Bex-lists and the most often used lemmas about them were formalised in the Coq proof assistant.

6 Alternative models

We chose the model and specifications in Section 4 with the following goals in mind.

- The model should admit short and clear specifications: it should be easy to see whether the intended meaning of an operation is expressed by its specification.
- Sequential composition should be straightforward: postconditions and preconditions should have the same form to make it easy to show that one operation's postcondition implies the next operation's precondition.
- The model should admit local reasoning: effects that are local in the implementation should also look local in the model.
- The specification should highlight the similarity between lists and views. Lists and views can be used interchangeably in many situations, so the reasoning in those cases should also be the same.

We believe that the bex-list model and the corresponding specifications achieve these goals. For comparison, we will here discuss some alternatives we considered before settling on bex-lists.

6.1 First Attempt

A straightforward way of modelling lists with views is to separate the model α into three components (L, B, E): a traditional cons-based list L of items, and maps B and E assigning to each view the offset in L where the view begins and ends. Formally,

$$\alpha = (L, B, E) \in \mathsf{int\ list} \times (\mathsf{View} \underset{\mathsf{fin}}{\rightharpoonup} \mathbb{N}) \times (\mathsf{View} \underset{\mathsf{fin}}{\rightharpoonup} \mathbb{N})$$

However, models that involve indices seem to lead to specifications that fail with respect to all four goals listed above. For example, the following attempt to specify remove, where $(\cdot)$ is list concatenation, leads to a postcondition that requires the

verifier to reason about subtraction.

$$\begin{aligned}
&\{\, \mathsf{L}(l, (L \cdot x{::}L', B, E)) \,\}\\
&l.\mathsf{remove}(|L|)\\
&\{\, \exists B', E'.\ \mathsf{L}(l, (L \cdot L', B', E')) \wedge\\
&\quad B' = \{[v \mapsto \text{if } j \le |L| \text{ then } j \text{ else } j-1] \mid [v \mapsto j] \in B\} \wedge\\
&\quad E' = \{[v \mapsto \text{if } j \le |L| \text{ then } j \text{ else } j-1] \mid [v \mapsto j] \in E\} \,\}
\end{aligned}$$

The specification also lacks locality: it is not clear from the specification that the update only affects the immediate vicinity of x since apparently all indices above x are decremented; however this reflects a property of the model, not the implementation.

6.2 Second Attempt

The first attempt above can be used as a basis of something better, though. First, lift the length function of cons-based lists to work for the whole model $\alpha = (L, B, E)$, defining $|\alpha| \triangleq |L|$. Then define concatenation $\alpha_1 \cdot \alpha_2$ of models:

$$\begin{aligned}
\alpha_1 \cdot \alpha_2 \triangleq \big(L_1 \cdot L_2,\ & B_1 \uplus (\mathsf{map}\ (+|L_1|)\ B_2),\\
& E_1 \uplus (\mathsf{map}\ (+|L_1|)\ E_2)\big)\\
&\text{where } (L_i, B_i, E_i) = \alpha_i \text{ for } i \in \{1, 2\}\\
&\text{and } \uplus \text{ is union of maps with disjoint domains.}
\end{aligned}$$

Note that model concatenation $\cdot$ is associative.

Finally, observe that any model $\alpha = (L, B, E)$ can be written as a concatenation of four basic building blocks:

$$\begin{aligned}
\epsilon &\triangleq (\mathsf{nil}, \{\}, \{\}) && \text{Empty list}\\
\underline{x} &\triangleq (x :: \mathsf{nil}, \{\}, \{\}) && \text{Single-item list}\\
[_v &\triangleq (\mathsf{nil}, \{[v \mapsto 0]\}, \{\}) && \text{View begins}\\
]_v &\triangleq (\mathsf{nil}, \{\}, \{[v \mapsto 0]\}) && \text{View ends}
\end{aligned}$$

With these ingredients, we can give concise specifications to most methods. In particular, remove looks much better than in the first attempt, and also more local and concise than when using bex-lists (Section 4.2):

$$\{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \gamma) \,\}\ l.\mathsf{remove}(|\alpha|)\ \{\, \mathsf{L}(l, \alpha \cdot \gamma) \,\}$$

Most other specifications resemble their bex-list cousins. For example,

$$\begin{aligned}
&\{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \gamma) \,\}\ l.\mathsf{setValue}(|\alpha|, x')\ \{\, \mathsf{L}(l, \alpha \cdot \underline{x'} \cdot \gamma) \,\}\\
&\{\, \mathsf{L}(l, \alpha \cdot \beta \cdot \gamma) \,\}\ l.\mathsf{view}(|\alpha|, |\beta|)\ \{\, \mathsf{L}(l, \alpha \cdot [_{ret} \cdot \beta \cdot]_{ret} \cdot \gamma) \,\}
\end{aligned}$$

The drawbacks of this model are due to $\alpha \cdot \beta = \beta \cdot \alpha$ when $|\alpha| = |\beta| = 0$. This equality allows us to freely re-order views analogously to how the sets b and e in the bex-list model allow it. However, to correctly specify insert we need a restrictive variant $\circ$ of concatenation in the precondition:

$$\{\, \mathsf{L}(l, \alpha \circ \beta) \,\}\ l.\mathsf{insert}(|\alpha|, x)\ \{\, \mathsf{L}(l, \alpha \cdot \underline{x} \cdot \beta) \,\}$$

Using the normal concatenation operator $\cdot$ in place of $\circ$ in the precondition would allow the verifier to "choose" whether a view that begins or ends at the insertion point would end up to the left or right of the inserted item x. But since those two outcomes are observably different, such a specification would allow deriving a logical contradiction.

Instead, we define $\circ$ to be an (even more) partial variant of concatenation. Intuitively, a list that can be written $\alpha_1 \circ \alpha_2$ must have no views that end just before the first item in α_2, and any view that begins just after the last item of α_1 must be empty. Formally,

$$
\begin{aligned}
\alpha_1 \circ \alpha_2 \triangleq\ & \alpha_1 \cdot \alpha_2 \\
& \text{if } \forall v.\ [v \mapsto 0] \notin E_2 \\
& \text{and } \big([v \mapsto |L_1|] \in B_1 \Rightarrow [v \mapsto |L_1|] \in E_1\big) \\
& \text{where } (L_i, B_i, E_i) = \alpha_i \text{ for } i \in \{1, 2\}
\end{aligned}
$$

Thus, the specification of insert is only beautiful because it hides its complexity beneath the definition of $\circ$. The verifier would have to develop a theory to establish $\circ$ before every call to insert. But the approach is not very general since $\circ$ is specific to the semantics of the insert operation. If some other variant of insert were introduced, there would have to be another restricted concatenation operator with a corresponding theory.

The same problem of herding the views onto the desired side of a concatenation appears if we want to formulate a theorem such as (1), which is crucial for independent reasoning about views. It is also no longer possible to define L to be as syntax-directed on α as for bex-lists, which makes it harder to prove the implementation correct.

It was our desire to syntactically restrict where views may begin and end that made us abandon this model in favour of the bex-lists, which make explicit the grouping of all views that begin or end at each list position. The specifications that arose from the "second attempt" model remain more elegant and intuitive, though, and it would be interesting to investigate whether it could work well if the semantics of LLWV were changed.

7 Future work

The future work specific to the LLWV data structure includes:

- The semantics of inserting new items in a LLWV [KS06] can easily lead to surprises since nearby views can be affected [Jen10]. Thus, it should be investigated whether views, and insertion in particular, could be defined differently, and if so, whether something better than the bex-list model can be found.
- In verifying both the LLWV implementation and sample clients [Jen10], proofs often required solving equations in ordered bex-lists. The solutions were often intuitively simple but somewhat laborious to prove formally. It seems likely that a decision procedure could be developed for a useful fragment of these equations.
- As discussed in Section 2, the "list with views" abstraction can be applied to both linked lists and array lists. It remains future work to formally verify the array list case. Also, C5 has a variation of LLWV that uses *hash-indexes* to implement operations such as deciding whether a given item resides within a

given view in constant expected time. Verification of this would surely be a challenge.

- It might be advantageous to replace the C5 library's current implementation of views (mentioned in Section 5.1) with that presented in Section 3, which seems to have some algorithmic advantages. To support the extension to hash-indexed lists and views mentioned above, it would need to have distinct start and end sentinels, though.
- Could some of these ideas be applied to specifying powerful iterators such as java.util.ListIterator? Java's list iterators permit more modifications to the list than iterators studied elsewhere in the separation logic literature, though they are still less powerful than views.
- It would be interesting to give a specification of LLWV in other recent logics for shared mutable data structures, e.g., region logic [BNR08], and compare with the present formulation.

The remaining points concern improvements to the logic motivated by insights gained from this case study.

- Current dialects of separation logic do not take advantage of the guarantees offered by memory-safe languages such as Java and C#. As discussed in Section 5.1, separation logic works as if all fields were public; it would be interesting to integrate reasoning about field access modifiers into the logic.
- Fractional permissions proved useful here, but they seem to be a somewhat blunt instrument when used in a sequential setting. Their read-only guarantee can only be applied at the granularity of a field, so it is impossible to express invariants such as the least significant bit of a field being read-only.
- The original C5 implementation of LLWV employs the System.WeakReference class to let lists point to their views through references that are ignored by the garbage collector. Modelling such weak references in separation logic might be interesting future work.

8 Conclusion

Several things can be done when implementing a data structure to ease verification with separation logic. When modularity is desired, data should be laid out such that heap separation coincides with lack of observable interference. Modularity and local reasoning demand more features from the logic, such as existentially-quantified predicates and fractional permissions, but in return they lead to cleaner specifications.

The bex-list model was chosen over other candidates because it better satisfied the goals listed in Section 6. It seems that there is a balance between choosing a model that is easy to verify and one that is easy to work with for clients: with a model that directly mimics the heap layout, the implementation will be easier to prove correct, but clients are likely to find the model unnatural to work with. The bex-list aims to be a compromise between the two extremes.

Further discussion and subtleties can be found in [Jen10].

References

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of POPL*, 2005.

[Blo08] Joshua Bloch. *Effective Java Programming Language Guide.* Addison-Wesley, second edition, 2008.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *Proceedings of ECOOP*, 2008.

[BRZ07] John Boyland, William Retert, and Yang Zhao. Iterators can be independent "from" their collections. In *Proceedings of ECOOP*, 2007.

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005. Lecture Notes in Computer Science, vol. 4111*, pages 342–363, 2006.

[CPV07] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *Proceedings of SAS*, 2007.

[Dij76] E.D. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of ECOOP*, 2010.

[FBL10] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *ACM Symposium on Applied Computing, March 2010, Sierre, Switzerland*, 2010.

[HH08] Christian Haack and Clément Hurlin. Resource usage protocols for iterators. In *Proceedings of IWACO*, 2008.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001.

[Jen10] Jonas B. Jensen. Specification and validation of data structures using separation logic. Master's thesis, Technical University of Denmark, February 2010.

[KAB+09] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *Proceedings of TLDI*, 2009.

[KS06] Niels Kokholm and Peter Sestoft. The C5 generic collection library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 2006.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Par05] Matthew Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[Par07] Matthew Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership in Object-Oriented programming (IWACO), Berlin, Germany*, 2007. At http://people.dsv.su.se/~tobias/iwaco/accepted.html.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of POPL*, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, 2008.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, 2002.

[Rey08] John C. Reynolds. An introduction to separation logic (preliminary draft). Course notes, October 2008.

[SBP10] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, 2010.

A Septraction

In this appendix we define the septraction operator, $-\circledast$, which was used in the definition of V.

A.1 Formal set-up

Our heap model is identical to the model of heaps with fractional permissions in [BCOP05]:

$$\mathit{Heap} \triangleq \mathit{ObjId} \times \mathit{Field} \underset{\mathsf{fin}}{\rightharpoonup} \mathit{Val} \times \{\pi \in \mathbb{Q} \mid 0 < \pi \leq 1\}$$

When $h_1, h_2 \in \mathit{Heap}$ we write $h_1 \# h_2$ to say that the composition of h_1 and h_2 is defined, and $h_1 \circ h_2$ denotes this composition. There is an ordering on heaps defined as $h_1 \sqsubseteq h$ iff $\exists h_2 \# h_1.\ h_1 \circ h_2 = h$.

The separation logic used throughout this article is intuitionistic, meaning that all formulas satisfy the *monotonicity condition*: they must continue to hold in any larger heap [IO01]. Thus, an assertion is a monotone function from heaps to Booleans, where the Booleans are ordered as $\mathsf{false} \sqsubseteq \mathsf{true}$.

A.2 The septraction operator

To get an intuitionistic septraction connective that has most of the desirable properties of its classical cousin [CPV07], we use the following definition.

Definition 1

$$(P -\circledast Q)\ h \triangleq \exists h_0 \sqsubseteq h.\ \exists h' \# h_0.\ \mathsf{Pr}\ P\ h' \wedge Q\ (h_0 \circ h') \qquad \text{where}$$

$$\mathsf{Pr}\ P\ h \triangleq P\ h \wedge \forall h' \sqsubseteq h.\ P\ h' \Rightarrow h' = h \qquad \blacksquare$$

Note that this definition satisfies the monotonicity condition since the same subheap h_0 continues to exist under any larger heap.

The operator $\mathsf{Pr} : (\mathit{Heap} \underset{\text{mon}}{\to} \mathbf{2}) \to (\mathit{Heap} \to \mathbf{2})$, where $\mathbf{2}$ denotes the Booleans, is Yang's *precising* operator (see the discussion of this operator in [Rey08]). In particular, $\mathsf{Pr}\ P\ h$ holds if h is a minimal heap such that $P\ h$.

To formulate the inference rules for septraction, we need the notion of a strongly supported assertion:

Definition 2 An assertion P is *strongly supported* if for any heap h, the set of subheaps of h that satisfy P is either empty or has a least element. ■

All the assertions in Figure 5 are strongly supported (modulo a few side conditions; see [Jen10] for details).

Lemma 1. *For strongly supported assertions P,*

$$\frac{Q \models P}{Q \models P * (P -\!\!\circledast\, Q)} \qquad \textit{and} \qquad \frac{Q \models P * Q'}{P -\!\!\circledast\, Q \models Q'}$$

where $\models$ denotes entailment among assertions; that is, $P \models Q$ iff for all h, $P\ h$ implies $Q\ h$.

Proof. Both proofs rely on the fact that if $P\ h$ then there exists $h_{\min} \sqsubseteq h$ such that $\mathsf{Pr}\ P\ h_{\min}$. □

A.3 Remark

An alternative to Definition 2 is the following:

Definition 3 An assertion P is *weakly supported* if for any heaps h_1, h_2 that are both subheaps of the same h and both satisfy P, there exists a heap h_{12} that is a subheap of both h_1 and h_2 and satisfies P. ■

Definitions 2 and 3 are usually equivalent and therefore used interchangeably [Rey08], but it turns out that they are not the same when fractional permissions are used. For example, the assertion $\exists z > 0.\ x.f \overset{z}{\mapsto} y$ is weakly but not strongly supported. This assertion is useful and natural since it represents a read-only points-to assertion that can be arbitrarily split across $*$.

B Conversions between lists and views

In this appendix we prove Equation (1) and its generalization to multiple views.

For brevity, we abbreviate a bex-list of the form $b\beta e$ as B and write B^v to mean $b^v\beta e^v$. Recall that the notation b^v means $b \cup \{v\}$ where $v \notin b$.

The following lemma captures the essence of why Equation (1) is valid.

Lemma 2. *If* $\mathsf{ord}(\alpha, \mathsf{B}, \mathsf{B})$ *and* $\mathsf{wf}(B)$*, then*

$$\mathsf{l}(n_1', n_2, \alpha B^v) \models \mathsf{V}(v, B) * \left[\mathsf{V}(v, B') -\!\!* \mathsf{l}(n_1', n_2, \alpha B'^v)\right]$$

Proof. The left part expands to $\mathsf{I}(n_1', n_1, \alpha) * \mathsf{I}(n_1, n_2, B^v)$ for some n_1, and it is the right part of this conjunction that is interesting to us. By Lemma 1, that part entails $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 * (v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 \multimap\!\circledast\ \mathsf{I}(n_1, n_2, B^v))$, where the side condition on the lemma follows from expanding the I predicate and its constituents. In that formula, the septraction part is just the definition of V, so we can contract that and get $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 * \mathsf{V}(v, B)$.

If we can now show that

$$v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2 \models \mathsf{V}(v, B') \mathrel{-\!\!*} \mathsf{I}(n_1, n_2, B'^v), \tag{5}$$

then we have altogether that

$$\mathsf{I}(n_1', n_2, \alpha B^v) \models \mathsf{I}(n_1', n_1, \alpha) * \mathsf{V}(v, B) * \big[\mathsf{V}(v, B') \mathrel{-\!\!*} \mathsf{I}(n_1, n_2, B'^v)\big],$$

and since $Q' * (P \mathrel{-\!\!*} Q) \models P \mathrel{-\!\!*} (Q * Q')$ we may then join the two I predicates by Equation (3) to finish the proof.

To show (5), first apply the fact that $Q \models P \mathrel{-\!\!*} (P * Q)$ to get $\mathsf{V}(v, B') \mathrel{-\!\!*} (\mathsf{V}(v, B') * v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2)$. On the right of the separating implication, unfold the definition of V to replace $\mathsf{V}(v, B')$ with $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e} \multimap\!\circledast\ \mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, B'^v)$ for new existentials n_b and n_e. Since there is an assertion I', too long to write out here, such that $\mathsf{I}(n_\mathrm{b}, n_\mathrm{e}, B'^v) \equiv I' * v.\mathsf{bgn} \mapsto n_\mathrm{b} * v.\mathsf{end} \mapsto n_\mathrm{e}$, and we can split the permissions on $v.\mathsf{bgn}$ and $v.\mathsf{end}$ in two halves, by the second half of Lemma 1 we can get $I' * v.\mathsf{bgn} \overset{0.5}{\mapsto} n_\mathrm{b} * v.\mathsf{end} \overset{0.5}{\mapsto} n_\mathrm{e}$. Recall that this is still in separating conjunction with $v.\mathsf{bgn} \overset{0.5}{\mapsto} n_1 * v.\mathsf{end} \overset{0.5}{\mapsto} n_2$, which lets us conclude that $n_1 = n_\mathrm{b}$ and $n_2 = n_\mathrm{e}$. Now we can join the split permissions and contract the I predicate again to get $\mathsf{I}(n_1, n_2, B'^v)$, which is what we wanted. □

In the following, let $i \geq 1$ and $m \geq 0$. The $\circledast$ operator binds tighter than $*$.

Lemma 3. *If* $\mathsf{ord}(\alpha_i, \mathsf{B}, \mathsf{B})$ *and* $\mathsf{wf}(B_i)$ *for all* i*, then*

$$\underset{i \leq m}{\circledast} \mathsf{I}(n_i, n_{i+1}, \alpha_i B_i^{v_i}) \models \underset{i \leq m}{\circledast} \mathsf{V}(v_i, B_i) * \big[\underset{i \leq m}{\circledast} \mathsf{V}(v_i, B_i') \mathrel{-\!\!*} \underset{i \leq m}{\circledast} \mathsf{I}(n_i, n_{i+1}, \alpha_i B_i'^{v_i})\big]$$

Proof. By induction on m. The base case is trivial. For the inductive case, let us first abbreviate the above formula to read

$$I(\leq m) \models V(\leq m) * [V'(\leq m) \mathrel{-\!\!*} I'(\leq m)]$$

Thus, we start out with $I(\leq m)$, which entails $I(< m) * I(m)$, and applying the induction hypothesis to the left part gives us $\big(V(< m) * [V'(< m) \mathrel{-\!\!*} I'(< m)]\big) * I(m)$. For the remaining part, since Lemma 2 gives us that

$$I(m) \models V(m) * [V'(m) \mathrel{-\!\!*} I'(m)],$$

then the whole entails $I(\leq m) \models V(\leq m) * [V'(\leq m) \mathrel{-\!\!*} I'(\leq m)]$ due to the fact that $(P \mathrel{-\!\!*} Q) * (P' \mathrel{-\!\!*} Q') \models (P * P') \mathrel{-\!\!*} (Q * Q')$. □

Theorem 1.

$$\mathsf{L}(l, \alpha_1 B_1^{v_1} \cdots \alpha_m B_m^{v_m} \gamma) \models \underset{i \leq m}{\circledast} \mathsf{V}(v_i, B_i) * \big[\underset{i \leq m}{\circledast} \mathsf{V}(v_i, B_i') \mathrel{-\!\!*} \mathsf{L}(l, \alpha_1 B_1'^{v_1} \cdots \alpha_m B_m'^{v_m} \gamma)\big]$$

Proof. By Lemma 3 and Equation (3) □

Corollary 1. *Equation* (1) *is valid; i.e.,*

$$\mathsf{L}(l, \alpha b^v \beta e^v \gamma) \implies \mathsf{V}(v, b\beta e) * \big[\mathsf{V}(v, b'\beta' e') \mathrel{-\!\!*} \mathsf{L}(l, \alpha b'^v \beta' e'^v \gamma)\big]$$